Proposer une autre formule de calcul du nombre de nombres premiers inférieurs à un nombre donné (Denise Vella-Chemla, 1.7.2017)

On voudrait ici proposer une formule légèrement différente de celle que Bernhard Riemann a élaborée pour compter le nombre de nombres premiers inférieurs à un nombre donné.

Bien que moins performante que la formule de Riemann, le but de cette nouvelle formule, basée sur une idée de Gauss, pourrait être de faciliter l'appréhension des processus à l'oeuvre dans les petites fluctuations qui interviennent pour le calcul des nombres de nombres premiers, pour le calcul des ratios aux logarithmes, ou bien pour le calcul des tailles des différents intervalles qui sont utilisés dans l'article que Riemann a consacré à la fonction $\pi(x)$.

On ne s'intéresse ici qu'aux formules suivantes de l'article de Riemann, soit que ces formules possèdent une propriété de symétrie, soit qu'elles permettent (c'est le cas pour la dernière) d'obtenir une formule exacte de décompte des nombres premiers. Il est dit que la volonté d'obtenir une formule exacte était l'une des motivations de Riemann lorsqu'il a écrit l'article qui énonce sa célèbre hypothèse.

1) $f(x) = \pi(x) + \frac{1}{2}\pi\left(x^{\frac{1}{2}}\right) + \frac{1}{3}\pi\left(x^{\frac{1}{3}}\right) + \frac{1}{4}\pi\left(x^{\frac{1}{4}}\right) + \frac{1}{5}\pi\left(x^{\frac{1}{5}}\right) + \ldots$

2) $\pi(x) = \frac{\mu(2)}{2} f\left(x^{\frac{1}{2}}\right) + \frac{\mu(3)}{3} f\left(x^{\frac{1}{3}}\right) + \frac{\mu(4)}{4} f\left(x^{\frac{1}{4}}\right) + \frac{\mu(5)}{5} f\left(x^{\frac{1}{5}}\right) + \ldots = \sum \frac{\mu(k)}{k} f\left(x^{\frac{1}{k}}\right)$

3) $\pi(x) = Li(x) + \frac{\mu(2)}{2} Li\left(x^{\frac{1}{2}}\right) + \frac{\mu(3)}{3} Li\left(x^{\frac{1}{3}}\right) + \frac{\mu(4)}{4} Li\left(x^{\frac{1}{4}}\right) + \frac{\mu(5)}{5} Li\left(x^{\frac{1}{5}}\right) + \ldots = \sum \frac{\mu(k)}{k} Li\left(x^{\frac{1}{k}}\right)$

4) $f(x) = Li(x) - \sum_{\rho}(Li(x^{\rho}) + Li(x^{\overline{\rho}})) + \int_x^{\infty} \frac{du}{u(u^2-1)ln\ u} + ln\ 2$

(avec $\rho = \frac{1}{2} + \alpha\ i$ et $\overline{\rho} = \frac{1}{2} - \alpha\ i$)

Les formules 2) et 3) utilisent la fonction de Moebius qui a comme propriété de permettre d'inverser certains calculs (par l'inversion de Moebius), i.e. elle permet d'obtenir l'image par une fonction d'un nombre en agrégeant les images de ses diviseurs. La fonction de Moebius "élimine les carrés" et compte la parité du nombre de diviseurs des produits purs (elle vaut -1 pour les nombres produits purs d'un nombre impair de diviseurs (ex : $30 = 2.3.5$ a 3 diviseurs) et 1 pour les nombres produits purs d'un nombre pair de diviseurs (ex : $210 = 2.3.5.7$ a 4 diviseurs). La fonction de Moebius "équilibre" à peu près les ajouts et les soustractions (jusqu'à 10 racines, 4 signes - pour 2 signes +, jusqu'à 100, 30 signes + et 30 signes - à égalité, jusqu'à 1000, 304 signes + pour 303 signes -, jusqu'à 10000, 3029 signes + pour 3053 signes -, jusqu'à 10^5, 30372 signes + pour 30421 signes -, etc.).

A la place de la formule de Moebius, on propose de compter, comme Gauss le préconise dans le chapitre des Recherches arithmétiques consacré à la loi de réciprocité quadratique, la parité du nombre de facteurs de la forme $4k-1$ intervenant dans les factorisations des nombres et de remplacer les valeurs fournies par la fonction de Moebius par celles fournies par ce nouveau comptage. On appelle le résultat de notre fonction $D(x)$.

$$D(x) = \pi(x) + \sum_k \frac{4kmoins1(k)}{k} \frac{x^{\frac{1}{k}}}{log\left(x^{\frac{1}{k}}\right)}.$$

On peut calculer les résultats obtenus pour cette nouvelle formule par le programme ci-après.

Ce programme n'est pas efficace du tout. De plus, on a mal initialisé le cumul : il aurait fallu l'initialiser à $\frac{x}{log(x)}$ plutôt qu'à $\pi(x)$. Toujours est-il que la différence entre la valeur finale de $D(x)$ et $\pi(x)$ est tout de même suffisamment petite pour présenter un intérêt ($D(x) - \pi(x) = 58$ pour 80 000). Il faudrait comprendre comment les différentes variables évoluent en analysant les premières valeurs (pour x jusqu'à 100). Il faudrait sûrement réfléchir dans une direction qui considère qu'un $4n+1$ peut être mis sous la forme $(-2\sqrt{n}+i)(-2\sqrt{n}-i)$ (les deux facteurs du produit étant deux points conjugués du plan complexe situés de part et d'autre du point réel $-2\sqrt{n}$, l'un en haut, l'autre en bas, sur la droite des complexes de partie réelle $-2\sqrt{n}$) tandis qu'un nombre de la forme $4n-1$ peut être mis sous la forme $(-2\sqrt{n}+1)(-2\sqrt{n}-1)$ (les deux facteurs étant deux réels situés de part et d'autre à distance 1 d'un zéro trivial). Mais ces idées sont pour l'instant trop spéculatives pour pouvoir être exploitées.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cmath>

int tabfacteurs[1000005], tabpuiss[1000005], tabexpo[1000005], tabcpte4kmoins1[1000005] ;

int prime(int atester) {
  bool pastrouve=true;
  unsigned long k = 2;
  if (atester == 1) return 0;
  if (atester == 2) return 1;
  if (atester == 3) return 1;
  if (atester == 5) return 1;
  if (atester == 7) return 1;
  while (pastrouve) {
    if ((k * k) > atester) return 1;
    else if ((atester % k) == 0) {
      return 0 ;
    }
    else k++;
  }
}

int arrondi(float nombre) {
  return nombre + 0.5;
}

int main(int argc,char **argv) {
  int  x, k, i, d, pix, p, nbdiv, tempo, expo ;
  float res, cumul ;
  int compte4kmoins1 ;

  compte4kmoins1 = 1 ;
  for (i = 1 ; i <= 1000000 ; ++i) {
    std::cout << "\n" << i << " -> " ;
    tabfacteurs[i] = 1 ;
    tabpuiss[i] = 1 ;
    tabexpo[i] = 1 ;
    tempo = i ;
    p = i/2 ;
    nbdiv = 1 ;
    if (prime(tempo)) {
      tabfacteurs[1] = tempo ;
      tabpuiss[1] = tempo ;
      tabexpo[1] = 1 ;
    }
    else while ((tempo > 1) && (p > 1)) {
      if ((prime(p)) && ((tempo%p) == 0)) {
        tabfacteurs[nbdiv] = p ;
        nbdiv = nbdiv+1 ;
        tempo = tempo/p ;
      }
      p=p-1 ;
    }
```

```
if (not(prime(i))) nbdiv=nbdiv-1 ;
    if ((nbdiv == 1) && (prime(i))) {
      tabpuiss[1] = i ;
      tabexpo[1] = 1 ;
    }
    else if ((nbdiv == 1) && (not(prime(i)))) {
      tempo = tabfacteurs[1] ;
      tabpuiss[1] = i ;
      expo = 1 ;
      while (tempo < i) {
        tempo=tempo*tabfacteurs[1] ;
        expo = expo+1 ;
      }
      tabexpo[1] = expo ;
    }
    else if (nbdiv > 1) {
      for (k = 1 ; k <= nbdiv ; ++k){
        tempo = tabfacteurs[k] ;
        expo = 1 ;
        while (((i % tempo) == 0) && (tempo < i)) {
          tempo=tempo*tabfacteurs[k] ;
          expo = expo+1 ;
  }
  tabpuiss[k] = tempo/tabfacteurs[k] ;
  tabexpo[k] = expo-1 ;
      }
    }
    for (k = 1 ; k <= nbdiv ; ++k) {
      std::cout << tabfacteurs[k] << "^" ;
      std::cout << tabexpo[k] << "." ;
      if ((tabfacteurs[k] % 4) == 3)
      if ((tabexpo[k] %2 ) == 1)
      compte4kmoins1 = compte4kmoins1 * (-1) ;
    }
    tabcpte4kmoins1[i] = compte4kmoins1 ;
  }
  std::cout << "\n" ;
  for (x = 2 ; x <= 1000000 ; ++x) {
    std::cout << x << " -->    " ;
    pix = 0 ;
    for (i = 2 ; i <= x ; ++i) if (prime(i)) pix = pix+1 ;
    std::cout << "pi("<< x << ")=" << pix ;
    std::cout << "  compte4kmoins1 " << tabcpte4kmoins1[x] << "\n" ;

    cumul = (float) pix ;
    for (k = 2 ; k <= sqrt(x) ; ++k) {
      res = pow(x,1./k) ;
      cumul = cumul+(tabcpte4kmoins1[k]/(float) k)*(res/log(res)) ;
    }
    std::cout << "res final " << arrondi(cumul) << "\n\n" ;
  }
}
```

Le tableau ci-dessous fournit les différences entre le nombre calculé par la fonction proposée ici $D(x)$ et la valeur effective de $\pi(x)$. Il fournit également les différences entre la fonction $Li(x)$ et la fonction $\pi(x)$. Pour 10^6, on voit que l'estimation la plus juste proposée par Riemann ($R(x)$ est bien meilleure que l'approximation proposée ici (30 (valeur trouvée dans la littérature) $\ll$ 60). En ce qui concerne le calcul de la formule 3), il n'est qu'approximatif : on a approximé les logarithmes intégrals en faisant calculer par Python les logarithmes intégrals de nombres entiers jusqu'à 1000000. Malgré cela, la formule 3) est la meilleure. Ne disposant pas d'un logiciel de calcul formel permettant de calculer les logarithmes intégrals, on ne peut pas pour l'instant voir l'efficacité de la formule 4).

x	$\pi(x)$	$Li(x)$	$Li(x)-\pi(x)$	$D(x)$	$D(x)-\pi(x)$	$formule\ 1)$	$formule\ 3)$
500	95	101	6	95	0	101	94
1 000	168	177	9	168	0	176	168
2 000	303	314	11	304	1	313	303
5 000	669	684	15	672	3	683	669
10 000	1 229	1 245	16	1 235	6	1 247	1 227
20 000	2 262	2 288	26	2 271	9	2 285	2 264
50 000	5 133	5 165	33	5 149	16	5 164	5 133
100 000	9 592	9 629	37	9 614	22	9 633	9 587
200 000	17 984	18 035	51	18 014	30	18 037	17 981
500 000	41 638	41 606	32	41 584	46	41 614	41 529
995 907	78 262	78 332	70	78 199	63	78 298	78 231
1 000 000	78 498	78 627	130	78 558	60	78 726	78 528
10 000 000	664 579	664 917	338			664 829	664 668
100 000 000	5 761 455	5 762 208	753			5 761 554	5 761 551
1 000 000 000	50 847 534	50 849 233	1 699			50 847 633	50 847 455

Ci-dessous, le tableau des écarts à $\pi(x)$, plus parlant :

x	$\pi(x)$	$D(x)-\pi(x)$	$formule\ 1)-\pi(x)$	$formule\ 3)-\pi(x)$
500	95	0	6	1
1 000	168	0	8	0
2 000	303	1	10	0
5 000	669	3	14	0
10 000	1 229	6	18	2
20 000	2 262	9	23	2
50 000	5 133	16	31	0
100 000	9 592	22	41	5
200 000	17 984	30	53	3
500 000	41 638	−46	−24	9
995 907	78 262	−63	36	31
1 000 000	78 498	60	228	30
10 000 000	664 579		250	89
100 000 000	5 761 455		99	96
1 000 000 000	50 847 534		99	79

En annexe sont fournies les valeurs calculées au fur et à mesure de l'application des formules 1 et 3 pour le nombre 995907, on voit bien les fluctuations autour de la valeur finalement atteinte.

Fournissons les valeurs qui ont été calculées pour estimer $D(1\ 000\ 000) = \pi(10^6) + \sum_k \frac{4kmoins1(k)}{k} \frac{x^{\frac{1}{k}}}{log\left(x^{\frac{1}{k}}\right)}$ et qui font intervenir les racines n-ièmes de 1000000 (les racines n-ièmes au-delà de la $20^{ème}$ sont trop petites (i.e.

inférieures strictement à 2, $\sqrt[20]{1\ 000\ 000} = 1.99526$), leur $\pi(k)$ est nul, elles n'ont pas à être prises en compte :

$\frac{1}{2}\frac{1000}{log(1000)}$	72.3824	$\frac{1}{11}\frac{3.51119}{log(3.51119)}$	0.254149
$\frac{-1}{3}\frac{100}{log(100)}$	−7.23824	$\frac{-1}{12}\frac{3.16228}{log(3.16228)}$	−0.228893
$\frac{-1}{4}\frac{31.6228}{log(31.6228)}$	−2.28893	$\frac{-1}{13}\frac{2.89427}{log(2.89427)}$	−0.209494
$\frac{-1}{5}\frac{15.8429}{log(15.8429)}$	−1.14718	$\frac{1}{14}\frac{2.6827}{log(2.6827)}$	0.19418
$\frac{1}{6}\frac{10}{log(10)}$	0.723824	$\frac{-1}{15}\frac{2.51189}{log(2.51189)}$	−0.181816
$\frac{-1}{7}\frac{7.19686}{log(7.19686)}$	−0.520926	$\frac{-1}{16}\frac{2.37137}{log(2.37137)}$	−0.171646
$\frac{-1}{8}\frac{5.62341}{log(5.62341)}$	−0.407036	$\frac{-1}{17}\frac{2.25393}{log(2.25393)}$	−0.163145
$\frac{-1}{9}\frac{4.64159}{log(4.64159)}$	−0.335969	$\frac{-1}{18}\frac{2.15443}{log(2.15443)}$	−0.155943
$\frac{-1}{10}\frac{3.98107}{log(3.98107)}$	−0.28816	$\frac{1}{19}\frac{2.06914}{log(2.06914)}$	0.149769

Annexe 1 : sources permettant de tester les fonctions moins performantes

Le programme ci-dessous a permis de trouver les valeurs de $f(x)$ jusqu'à 1 000 000 000 selon la première formule fournie (formule proposée en 1) dans la première page).

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cmath>

int prime(int atester) {
  bool pastrouve=true;
  unsigned long k = 2;

  if (atester == 1) return 0;
  if (atester == 2) return 1;
  if (atester == 3) return 1;
  if (atester == 5) return 1;
  if (atester == 7) return 1;
  while (pastrouve) {
    if ((k * k) > atester) return 1;
    else if ((atester % k) == 0) {
      return 0 ;
    }
    else k++;
  }
}

int moebius(int n) {
  int moebius, p, nbdiv, carre ;

  moebius = -2 ;
  nbdiv = 0 ;
  for (p=2 ; p <= n ; p++) {
    if (prime (p)) {
      carre = p*p ;
      if ((n % carre) == 0) moebius = 0 ;
      if ((n % p) == 0) nbdiv++ ;
    }
  }
  if (moebius == 0) return 0 ;
  else {
    if ((nbdiv % 2) == 1) return -1 ;
    else return 1 ;
  }
}

int arrondi(float nombre) {
  return nombre + 0.5;
}
```

```
int main(int argc,char **argv) {
  int  x, y, tempo ;
  float somme, pix, restempo ;
  int stocke[100005] ;

  for (x = 2 ; x <= 100000 ; ++x) stocke[x] = 0 ;
  for (x = 2 ; x <= 100000 ; ++x) {
    pix = 0 ;
    for (y = 1 ; y <= x ; ++y)
      if (prime(y)) pix = pix+1 ;
    stocke[x] = pix ;
  }
  for (x = 2 ; x <= 100000 ; ++x) std::cout << x << " -> pi(x) " << stocke[x] << " moebius(x) " <<
      moebius(x) << "\n" ;

  for (x = 2 ; x <= 100000 ; ++x) {
    std::cout << x << " --> " ;
    somme = stocke[x] ;
    tempo = 2 ;
    while (pow(x, 1./tempo) >= 2) {
      restempo = (1.0/(float)tempo)*((float)stocke[(int)pow(x, 1./tempo)]) ;
      std::cout << "rs " << restempo << "\n" ;
      somme = somme+restempo ;
      tempo = tempo+1 ;
   }
   std::cout << (int) somme << "\n" ;
  }
  std::cout << "\n\n" ;
}
```

Le programme ci-dessous a permis de trouver les valeurs de $\pi(x)$ jusqu'à 1 000 000 000 selon la troisième formule fournie (formule proposée en 3) dans la première page). Les logarithmes intégrals ont été calculés avec gnu-octave et stockés préalablement dans un fichier dans lequel on va les chercher.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cmath>
#include <fstream>

int prime(int atester) {
  bool pastrouve=true;
  unsigned long k = 2;

  if (atester == 1) return 0;
  if (atester == 2) return 1;
  if (atester == 3) return 1;
  if (atester == 5) return 1;
  if (atester == 7) return 1;
  while (pastrouve) {
    if ((k * k) > atester) return 1;
    else if ((atester % k) == 0) {
      return 0 ;
    }
    else k++;
  }
}
```

```
1
2  int moebius(int n) {
3    int moebius, p, racine, nbdiv, carre ;
4
5    moebius = -2 ;
6    nbdiv = 0 ;
7    racine = sqrt(n) ;
8    for (p=2 ; p <= n ; p++) {
9      if (prime (p)) {
10       carre = p*p ;
11       if ((n % carre) == 0) moebius = 0 ;
12       if ((n % p) == 0) nbdiv++ ;
13     }
14   }
15   if (moebius == 0) return 0 ;
16   else {
17     if ((nbdiv % 2) == 1) return -1 ;
18     else return 1 ;
19   }
20 }
21
22 int arrondi(float nombre) {
23   return nombre + 0.5;
24 }
25
26 int main(int argc,char **argv) {
27   int  x, y, pix, mu, k, i ;
28   float res, cumul ;
29   int stocke[10005] ;
30   float logainte[10005] ;
31
32   for (x = 2 ; x <= 10000 ; ++x) stocke[x] = 0 ;
33   for (x = 2 ; x <= 10000 ; ++x) {
34     pix = 0 ;
35     for (y = 1 ; y <= x ; ++y)
36       if (prime(y)) pix = pix+1 ;
37     stocke[x] = pix ;
38   }
39   for (x = 2 ; x <= 10000 ; ++x) {
40     std::cout << x << " -> pi(x) " << stocke[x] ;
41     std::cout << "  moebius(x) " << moebius(x) << "\n" ;
42   }
43   std::ifstream fichier("Li10000", std::ios::in);
44   if (fichier) {
45     float flotte ;
46
47     i = 2 ;
48     while (not fichier.eof()) {
49       fichier >> flotte ;
50       logainte[i] = flotte ;
51       std::cout << "li(" << i << ") = " << logainte[i] << "\n" ;
52       i = i+1 ;
53     }
54     fichier.close();
55   }
56   else std::cerr << "Impossible dâouvrir le fichier !" << std::endl ;
57   for (x = 2 ; x <= 10000 ; ++x) {
58      std::cout << x << " --> " ;
59      cumul = (float) logainte[x] ;
60      for (k = 2 ; k <= x ; ++k)
61        cumul = cumul+((float) moebius(k) /(float) k)*(logainte[(int) pow(x, 1./k)]) ;
62      std::cout << cumul << "\n" ;
63   }
64 }
```

Annexe 2 : quelques valeurs de racines n-ièmes

x	500	1000	2000	5000	10000	20000	50000	100000	200000	500000	995907	10^6	10^9
$\sqrt[2]{x}$	22.3607	31.6228	44.7214	70.7107	100	141.421	223.607	316.228	447.214	707.107	997.951	1000	31622.8
$\sqrt[3]{x}$	7.93701	10	12.5992	17.0998	21.5443	27.1442	36.8403	46.4159	58.4804	79.3701	99.8634	100	1000
$\sqrt[4]{x}$	4.72871	5.62341	6.6874	8.40896	10	11.8921	14.9535	17.7828	21.1474	26.5915	31.5904	31.6228	177.828
$\sqrt[5]{x}$	3.46572	3.98107	4.57305	5.4928	6.30957	7.2478	8.70551	10	11.487	13.7973	15.8359	15.8489	63.0957
$\sqrt[6]{x}$	2.81727	3.16228	3.54954	4.13519	4.64159	5.21001	6.06962	6.81292	7.64724	8.90899	9.99317	10	31.6228
$\sqrt[7]{x}$	2.42978	2.6827	2.96194	3.37617	3.72759	4.1156	4.69117	5.17947	5.7186	6.51836	7.19264	7.19686	19.307
$\sqrt[8]{x}$	2.17456	2.37137	2.586	2.89982	3.16228	3.44849	3.86697	4.21697	4.59863	5.15669	5.62053	5.62341	13.3352
$\sqrt[9]{x}$	1.99474	2.15443	2.32692	2.5763	2.78256	3.00533	3.32742	3.59381	3.88153	4.29753	4.63947	4.64159	10
$\sqrt[10]{x}$		1.99526	2.13847	2.34367	2.51189	2.69217	2.95051	3.16228	3.38925	3.71447	3.97944	3.98107	7.94328
$\sqrt[11]{x}$			1.99569	2.16905	2.31013	2.46038	2.67411	2.84804	3.03328	3.29677	3.50988	3.51119	6.57933
$\sqrt[12]{x}$				2.03352	2.15443	2.28254	2.46366	2.61016	2.76537	2.98479	3.1612	3.16228	5.62341
$\sqrt[13]{x}$				1.92547	2.03092	2.14214	2.29858	2.42446	2.55724	2.74399	2.89335	2.89427	4.92388
$\sqrt[14]{x}$					1.9307	2.02869	2.16591	2.27585	2.39136	2.55311	2.68191	2.6827	4.39397
$\sqrt[15]{x}$						1.93524	2.05714	2.15443	2.25633	2.39845	2.5112	2.51189	3.98107
$\sqrt[16]{x}$							1.96646	2.05353	2.14444	2.27084	2.37077	2.37137	3.65174
$\sqrt[17]{x}$								1.96842	2.05034	2.16388	2.25339	2.25393	3.38386
$\sqrt[18]{x}$									1.97016	2.07305	2.15394	2.15443	3.16228
$\sqrt[19]{x}$										1.99501	2.06869	2.06914	2.97635
$\sqrt[20]{x}$											1.99485	1.99526	2.81838
$\sqrt[21]{x}$													2.6827
$\sqrt[22]{x}$													2.56502
$\sqrt[23]{x}$													2.46209
$\sqrt[24]{x}$													2.37137
$\sqrt[25]{x}$													2.29087
$\sqrt[26]{x}$													2.21898
$\sqrt[27]{x}$													2.15443
$\sqrt[28]{x}$													2.09618
$\sqrt[29]{x}$													2.04336
$\sqrt[30]{x}$													1.99526

Annexe 3 : Valeurs intermédiaires dans les calculs des formules 1 et 3 pour 995907

res = 78199.0.
+(1.0/2.0)*168.0 → 78283.00000
+ (1.0/3.0) ∗ 25.0 → 78291.33333
+ (1.0/4.0) ∗ 11.0 → 78294.08333
+ (1.0/5.0) ∗ 6.0 → 78295.28333
+ (1.0/6.0) ∗ 4.0 → 78295.95000
+ (1.0/7.0) ∗ 4.0 → 78296.52143
+ (1.0/8.0) ∗ 3.0 → 78296.89643
+ (1.0/9.0) ∗ 2.0 → 78297.11865
+ (1.0/10.0) ∗ 2.0 → 78297.31865
+ (1.0/11.0) ∗ 2.0 → 78297.50047
+ (1.0/12.0) ∗ 2.0 → 78297.66714
+ (1.0/13.0) ∗ 1.0 → 78297.74406
+ (1.0/14.0) ∗ 1.0 → 78297.81549
+ (1.0/15.0) ∗ 1.0 → 78297.88215
+ (1.0/16.0) ∗ 1.0 → 78297.94465
+ (1.0/17.0) ∗ 1.0 → 78298.00348
+ (1.0/18.0) ∗ 1.0 → 78298.05903
+ (1.0/19.0) ∗ 1.0 → 78298.11166
$formule\ 1\ \rightarrow\ 78298.11166$

res = 78330.198822562036 (=li(995907)-li(2)).
−(1.0/2.0)*176.268 → 78242.06482
− (1.0/3.0) ∗ 29.0513 → 78232.38106
− (1.0/5.0) ∗ 7.41526 → 78230.89800
+ (1.0/6.0) ∗ 5.11747 → 78231.75092
− (1.0/7.0) ∗ 3.8102 → 78231.20660
+ (1.0/10.0) ∗ 1.90756 → 78231.39736
− (1.0/11.0) ∗ 1.55148 → 78231.25631
− (1.0/13.0) ∗ 1.101972 → 78231.17155
+ (1.0/14.0) ∗ 0.813336 → 78231.22964
+ (1.0/15.0) ∗ 0.634324 → 78231.27193
− (1.0/17.0) ∗ 0.336776 → 78231.25212
− (1.0/19.0) ∗ 0.0967475 → 78231.24703
$formule\ 3\ \rightarrow\ 78231.24703$

Annexe 4 : Valeurs intermédiaires dans les calculs des formules 1 et 3 pour 10^6

res = 78627.549159.
+(1.0/2.0)*168.0 → 78711.54916
+ (1.0/3.0) ∗ 25.0 → 78719.88249
+ (1.0/4.0) ∗ 11.0 → 78722.63249
+ (1.0/5.0) ∗ 6.0 → 78723.83249
+ (1.0/6.0) ∗ 4.0 → 78724.49916
+ (1.0/7.0) ∗ 4.0 → 78725.07059
+ (1.0/8.0) ∗ 3.0 → 78725.44559
+ (1.0/9.0) ∗ 2.0 → 78725.66781
+ (1.0/10.0) ∗ 2.0 → 78725.86781
+ (1.0/11.0) ∗ 2.0 → 78726.04963
+ (1.0/12.0) ∗ 2.0 → 78726.21629
+ (1.0/13.0) ∗ 1.0 → 78726.29322
+ (1.0/14.0) ∗ 1.0 → 78726.36465
+ (1.0/15.0) ∗ 1.0 → 78726.43131
+ (1.0/16.0) ∗ 1.0 → 78726.49381
+ (1.0/17.0) ∗ 1.0 → 78726.55264
+ (1.0/18.0) ∗ 1.0 → 78726.60819
+ (1.0/19.0) ∗ 1.0 → 78726.66082
$formule\ 1\ \rightarrow\ 78726.66082$

res = 78627.549159 (=li(1000000)-li(2)).
$-(1.0/2.0)$*176.56449 $\rightarrow$ 78539.26691
$-\ (1.0/3.0) * 29.080977 \rightarrow 78529.57326$
$-\ (1.0/5.0) * 7.41996 \rightarrow 78528.08926$
$+\ (1.0/6.0) * 5.120435 \rightarrow 78528.94267$
$-\ (1.0/7.0) * 3.81234 \rightarrow 78528.39805$
$+\ (1.0/10.0) * 1.90874 \rightarrow 78528.58892$
$-\ (1.0/11.0) * 1.55252 \rightarrow 78528.44778$
$-\ (1.0/13.0) * 1.02059 \rightarrow 78528.36928$
$+\ (1.0/14.0) * 0.814136 \rightarrow 78528.42743$
$+\ (1.0/15.0) * 0.635074 \rightarrow 78528.46977$
$-\ (1.0/17.0) * 0.33744 \rightarrow 78528.44992$
$-\ (1.0/19.0) * 0.0973665 \rightarrow 78528.44479$
$formule\ 3\ \rightarrow\ 78528.44479$

Annexe 5 : Valeurs intermédiaires dans les calculs des formules 1 et 3 pour 10^7

res = 664579.0.
$+(1.0/2.0)$*446.0 $\rightarrow$ 664802.00000
$+\ (1.0/3.0) * 47.0 \rightarrow 664817.66667$
$+\ (1.0/4.0) * 25.0 \rightarrow 664823.91667$
$+\ (1.0/5.0) * 9.0 \rightarrow 664825.71667$
$+\ (1.0/6.0) * 6.0 \rightarrow 664826.71667$
$+\ (1.0/7.0) * 4.0 \rightarrow 664827.28810$
$+\ (1.0/8.0) * 4.0 \rightarrow 664827.78810$
$+\ (1.0/9.0) * 4.0 \rightarrow 664828.23254$
$+\ (1.0/10.0) * 3.0 \rightarrow 664828.53254$
$+\ (1.0/11.0) * 2.0 \rightarrow 664828.71436$
$+\ (1.0/12.0) * 2.0 \rightarrow 664828.88102$
$+\ (1.0/13.0) * 2.0 \rightarrow 664829.03487$
$+\ (1.0/14.0) * 2.0 \rightarrow 664829.17773$
$+\ (1.0/15.0) * 1.0 \rightarrow 664829.24439$
$+\ (1.0/16.0) * 1.0 \rightarrow 664829.30689$
$+\ (1.0/17.0) * 1.0 \rightarrow 664829.36572$
$+\ (1.0/18.0) * 1.0 \rightarrow 664829.42127$
$+\ (1.0/19.0) * 1.0 \rightarrow 664829.47391$
$+\ (1.0/20.0) * 1.0 \rightarrow 664829.52391$
$+\ (1.0/21.0) * 1.0 \rightarrow 664829.57152$
$+\ (1.0/22.0) * 1.0 \rightarrow 664829.61698$
$+\ (1.0/23.0) * 1.0 \rightarrow 664829.66046$
$formule\ 1\ \rightarrow\ 664829.66046$

res = 664917.359884 (=li(10000000)-li(2)).
$-(1.0/2.0)$*461.916 $\rightarrow$ 664686.40188
$-\ (1.0/3.0) * 52.0412 \rightarrow 664669.05482$
$-\ (1.0/5.0) * 10.5047 \rightarrow 664666.95388$
$+\ (1.0/6.0) * 6.99028 \rightarrow 664668.11892$
$-\ (1.0/7.0) * 5.12043572 \rightarrow 664667.38743$
$+\ (1.0/10.0) * 2.59679 \rightarrow 664667.64711$
$-\ (1.0/11.0) * 2.15298 \rightarrow 664667.45139$
$-\ (1.0/13.0) * 1.50758 \rightarrow 664667.33542$
$+\ (1.0/14.0) * 1.26268 \rightarrow 664667.42561$
$+\ (1.0/15.0) * 1.05275 \rightarrow 664667.49579$
$-\ (1.0/17.0) * 0.708872 \rightarrow 664667.45410$
$-\ (1.0/19.0) * 0.435926 \rightarrow 664667.43115$
$+\ (1.0/21.0) * 0.2115 \rightarrow 664667.44122$
$+\ (1.0/22.0) * 0.113026 \rightarrow 664667.44636$
$-\ (1.0/23.0) * 0.0220097 \rightarrow 664667.44540$
$formule\ 3\ \rightarrow\ 664667.44540$

Annexe 6 : Valeurs intermédiaires dans les calculs des formules 1 et 3 pour 10^8

res = 5761455.0.
+(1.0/2.0)*168.0 → 5761539.00000
+ (1.0/3.0) ∗ 25.0 → 5761547.33333
+ (1.0/4.0) ∗ 11.0 → 5761550.08333
+ (1.0/5.0) ∗ 6.0 → 5761551.28333
+ (1.0/6.0) ∗ 4.0 → 5761551.95000
+ (1.0/7.0) ∗ 4.0 → 5761552.52143
+ (1.0/8.0) ∗ 3.0 → 5761552.89643
+ (1.0/9.0) ∗ 2.0 → 5761553.11865
+ (1.0/10.0) ∗ 2.0 → 5761553.31865
+ (1.0/11.0) ∗ 2.0 → 5761553.50047
+ (1.0/12.0) ∗ 2.0 → 5761553.66714
+ (1.0/13.0) ∗ 1.0 → 5761553.74406
+ (1.0/14.0) ∗ 1.0 → 5761553.81549
+ (1.0/15.0) ∗ 1.0 → 5761553.88215
+ (1.0/16.0) ∗ 1.0 → 5761553.94465
+ (1.0/17.0) ∗ 1.0 → 5761554.00348
+ (1.0/18.0) ∗ 1.0 → 5761554.05903
+ (1.0/19.0) ∗ 1.0 → 5761554.11166
$formule\ 1\ \rightarrow\ 5761554.11166$

res = 5762208.330284 (=li(100000000)-li(2)).
−(1.0/2.0)*1245.0920 → 5761585.78428
− (1.0/3.0) ∗ 94.9471 → 5761554.13525
− (1.0/5.0) ∗ 14.743 → 5761551.18665
+ (1.0/6.0) ∗ 9.36926 → 5761552.74819
− (1.0/7.0) ∗ 6.69582 → 5761551.79165
+ (1.0/10.0) ∗ 3.34743 → 5761552.12639
− (1.0/11.0) ∗ 2.79446 → 5761551.87235
− (1.0/13.0) ∗ 2.01134 → 5761551.71763
+ (1.0/14.0) ∗ 1.72081 → 5761551.84055
+ (1.0/15.0) ∗ 1.47471 → 5761551.93886
− (1.0/17.0) ∗ 1.07737 → 5761551.87549
− (1.0/19.0) ∗ 0.767058 → 5761551.83511
− (1.0/21.0) ∗ 0.515185 → 5761551.81058
− (1.0/22.0) ∗ 0.405563 → 5761551.79215
− (1.0/23.0) ∗ 0.304730 → 5761551.77890
− (1.0/26.0) ∗ 0.0441204 → 5761551.77720
$formule\ 3\ \rightarrow\ 5761551.77720$

Annexe 7 : Valeurs intermédiaires dans les calculs des formules 1 et 3 pour 10^9

res = 50847534.0.
+(1.0/2.0)*168.0 → 50847618.00000
+ (1.0/3.0) ∗ 25.0 → 50847626.33333
+ (1.0/4.0) ∗ 11.0 → 50847629.08333
+ (1.0/5.0) ∗ 6.0 → 50847630.28333
+ (1.0/6.0) ∗ 4.0 → 50847630.95000
+ (1.0/7.0) ∗ 4.0 → 50847631.52143
+ (1.0/8.0) ∗ 3.0 → 50847631.89643
+ (1.0/9.0) ∗ 2.0 → 50847632.11865
+ (1.0/10.0) ∗ 2.0 → 50847632.31865
+ (1.0/11.0) ∗ 2.0 → 50847632.50047
+ (1.0/12.0) ∗ 2.0 → 50847632.66714
+ (1.0/13.0) ∗ 1.0 → 50847632.74406
+ (1.0/14.0) ∗ 1.0 → 50847632.81549
+ (1.0/15.0) ∗ 1.0 → 50847632.88215
+ (1.0/16.0) ∗ 1.0 → 50847632.94465
+ (1.0/17.0) ∗ 1.0 → 50847633.00348
+ (1.0/18.0) ∗ 1.0 → 50847633.05903

$+ (1.0/19.0) * 1.0 \rightarrow 50847633.11166$
$formule\ 1\ \rightarrow\ 50847633.11166$

res = 50849233.911838 (=li(1000000000)-li(2)).
$-(1.0/2.0)$*3432.87 $\rightarrow$ 50847517.47684
$- (1.0/3.0) * 176.5644942 \rightarrow 50847458.62201$
$- (1.0/5.0) * 20.6717 \rightarrow 50847454.48767$
$+ (1.0/6.0) * 12.4509 \rightarrow 50847456.56282$
$- (1.0/7.0) * 8.62744 \rightarrow 50847455.33033$
$+ (1.0/10.0) * 4.18123 \rightarrow 50847455.74845$
$- (1.0/11.0) * 3.49223 \rightarrow 50847455.43097$
$- (1.0/13.0) * 2.5419 \rightarrow 50847455.23544$
$+ (1.0/14.0) * 2.19725 \rightarrow 50847455.39239$
$+ (1.0/15.0) * 1.90874 =\rightarrow 50847455.51964$
$- (1.0/17.0) * 1.44963 \rightarrow 50847455.43437$
$- (1.0/19.0) * 1.09682 \rightarrow 50847455.37664$
$- (1.0/21.0) * 0.814136 \rightarrow 50847455.33787$
$- (1.0/22.0) * 0.692111 \rightarrow 50847455.30641$
$- (1.0/23.0) * 0.58041 \rightarrow 50847455.28117$
$- (1.0/26.0) * 0.294016 \rightarrow 50847455.26987$
$- (1.0/29.0) * 0.0616036 \rightarrow 50847455.26774$
$formule\ 3\ \rightarrow\ 50847455.26774$