Statistiques et arithmétique avec CoffeeScript

I/ Répartition des nombres premiers

> On choisit un nombre entier au hasard ; quelle est la probabilité qu'il soit premier ?

Cette question est bien plus compliquée qu'elle n'en a l'air : Que signifie, tout d'abord, « choisir un entier au hasard » ? Comment modéliser ce choix au hasard ? En effet si « au hasard » veut dire « avec équirépartition », c'est impossible, car il n'existe pas de loi de probabilité uniforme sur tous les entiers, ceux-ci étant en nombre infini. On peut penser alors à choisir un nombre au hasard suivant une loi de Poisson mais avec quel paramètre ? En plus les lois de Poisson favorisent les petits nombres et 0 et 1 ne pouvant être premiers doivent être exclus de ce choix au hasard. On propose alors un énoncé plus raisonnable :

> On choisit un nombre entier au hasard, entre 2 et 1002 ; quelle est la probabilité qu'il soit premier ?

Mais pourquoi 1002 et pas un autre nombre ? Dans cette partie on va explorer algorithmiquement mais surtout statistiquement ce qui se passe lorsqu'on fait varier la taille de l'échantillon (ci-dessus, 1000) et notamment le comportement asymptotique de la répartition des nombres premiers. L'algorithmique intervient lorsqu'on cherche à faire l'étude en un temps raisonnable...

1. Obtention des nombres premiers

Un algorithme simple pour avoir la liste des nombres premiers consiste à effectuer une boucle sur les nombres entiers allant de 2 à 1002 et, dans cette boucle, de tester la primalité de l'indice. C'est d'ailleurs l'algorithme décrit dans l'Encyclopédie[1] :

> Pour trouver la suite des nombres *premiers*, il n'y a qu'à parcourir tous les nombres depuis 1 jusqu'à l'infini ; examiner ceux qui n'ont point d'autre diviseur que l'unité ou qu'eux-mêmes, les ranger par ordre, & l'on aura par ce moyen autant de nombres *premiers* que l'on voudra.

Mais pour tester la primalité d'un entier, on va typiquement tester la divisibilité par les entiers inférieurs à sa racine carrée. On va alors boucler (sur les diviseurs) dans une boucle (sur les entiers de 2 à 1002) et il faut s'attendre à ce que ce soit long si on veut la liste des nombres premiers jusqu'à 1002 :

1 Tome 2 du volume « mathématiques » de l'Encyclopédie de d'Alembert, paru en 1783, article *premier (nombre)*.

```
boucles = 0
for entier in [2..1002]
    boucles += troncature(racine(entier))
affiche  boucles
```

L'exécution de ce script révèle que la fonction « modulo » est calculée 20676 fois ce qui est tout de même beaucoup !

On va donc dans la suite, pour ne pas passer trop de temps à attendre que l'ordinateur ait fini, utiliser un algorithme rapide donnant la liste des nombres premiers : Celui d'Eratosthène. Rappelons son principe :

- on part de la liste de tous les entiers inférieurs ou égaux à 1002, en commençant par 2 ;
- on cherche le plus petit entier de la liste non encore testé (au début, 2) ;
- on enlève tous ses multiples de la liste ;
- on boucle : chercher le plus petit entier non encore testé, etc.

Comme la racine carrée de 1002 vaut environ 39, on sait que si un nombre supérieur à 39 n'a pas été éliminé de la liste, c'est qu'il est premier parce que sinon il aurait nécessairement un diviseur inférieur à 39. L'algorithme d'Eratosthène[2] se rédige ainsi en CoffeeScript (avec l'outil alcoffeethmique pour l'affichage) :

```
premiers = [2..1002]
for d in [2..39]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
```

Cette version enrichie pour calculer le nombre de fois que la fonction « modulo » est calculée :

```
premiers = [2..1002]
boucles = 0

for d in [2..39]
    boucles += (x for x in premiers when x>d).length
    premiers = (x for x in premiers when x<=d or x%d isnt 0)

affiche boucles
```

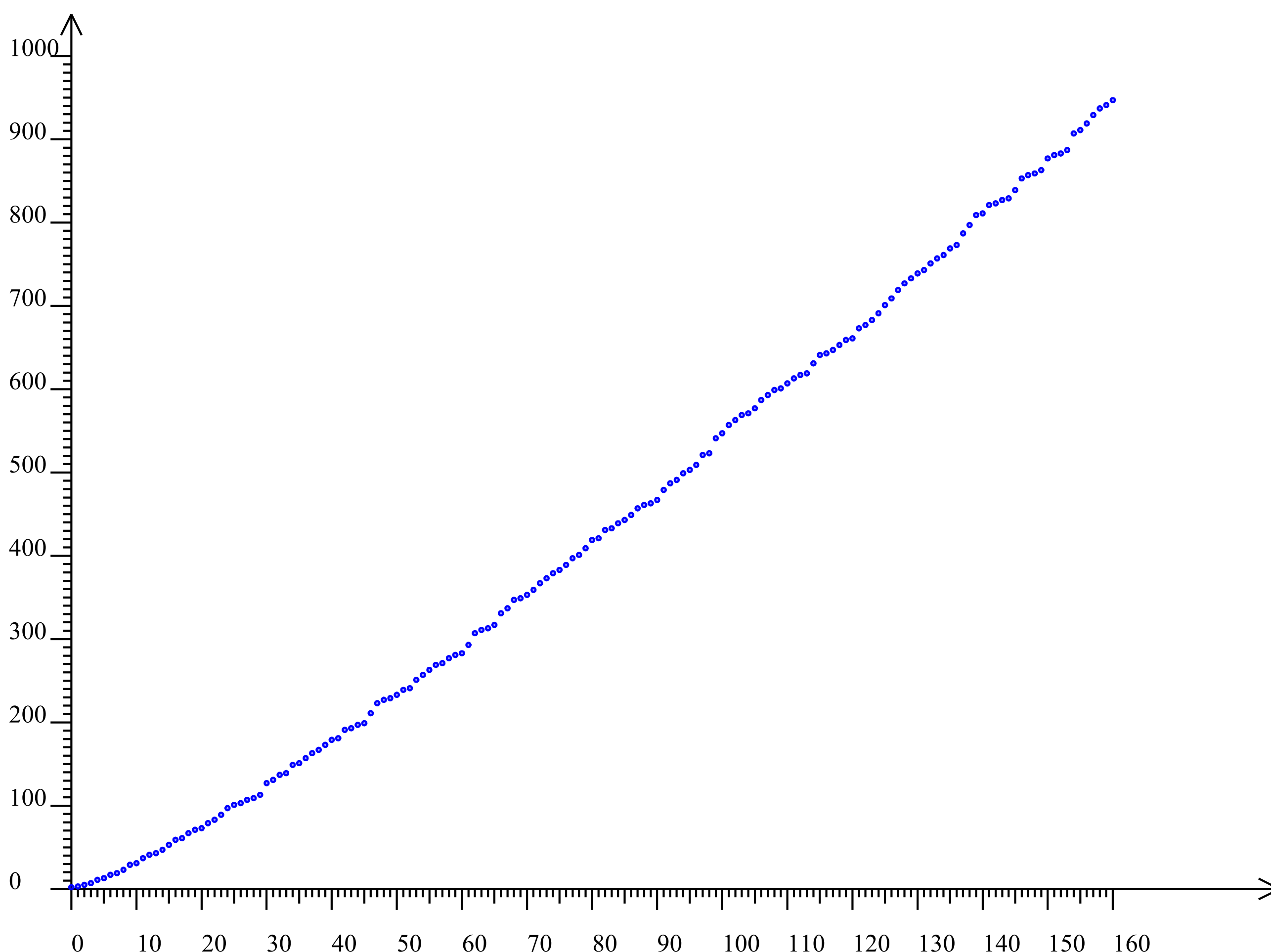
L'exécution de ce script révèle que cette fois-ci la fonction « modulo » n'est plus calculée que 8339 fois, soit nettement moins que la méthode « naïve ». Merci Eratosthène !

2 Qui avait déjà été utilisé pour cet exerciciel.

Un ajout de commande d'affichage permet de représenter la suite des nombres premiers à laquelle l'Encyclopédie faisait allusion :

```
premiers = [2..1002]
for d in [2..39]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
dessineSuite premiers, 160, 0, 1000, 1, "blue"
```

L'exécution du script donne presque instantanément le graphique suivant, qui illustre le fait que les nombres premiers constituent une suite croissante d'entiers[3] :



Noter que la répartition des nombres premiers est symétrique de ce nuage de points par rapport à la première bissectrice : Le fait que le quatrième nombre premier est 7 est lié à ce que $\pi(7)=4$. Voir les effectifs cumulés ci-dessous, à ce sujet.

2. Statistiques sur les nombres premiers

Une fois qu'on a la liste des nombres premiers, on peut faire des calculs statistiques comme la médiane, les quartiles ou la moyenne. Et la répartition des nombres premiers peut se faire par une

3 On dit que l'ensemble des nombres premiers est récursivement énumérable ; en fait, on a même mieux : C'est un ensemble récursif parce que l'ensemble des nombres composés est lui aussi récursivement énumérable.

juxtaposition de boîtes à moustaches. Par exemple, pour afficher la moyenne des nombres premiers jusqu'à 1002, on modifie légèrement le script ci-dessus :

```
premiers = [2..1002]
for d in [2..39]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
affiche laMoyenneDe premiers
```

On apprend que le nombre premier moyen est environ 453 :

```
Algorithme lancé
453.13690476190476

Algorithme exécuté en 74 millisecondes
```

De même, *alcoffeethmique* peut afficherl'écart-type de cette série statistique :

```
premiers = [2..1002]
for d in [2..39]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
affiche lEcartTypeDe premiers
```

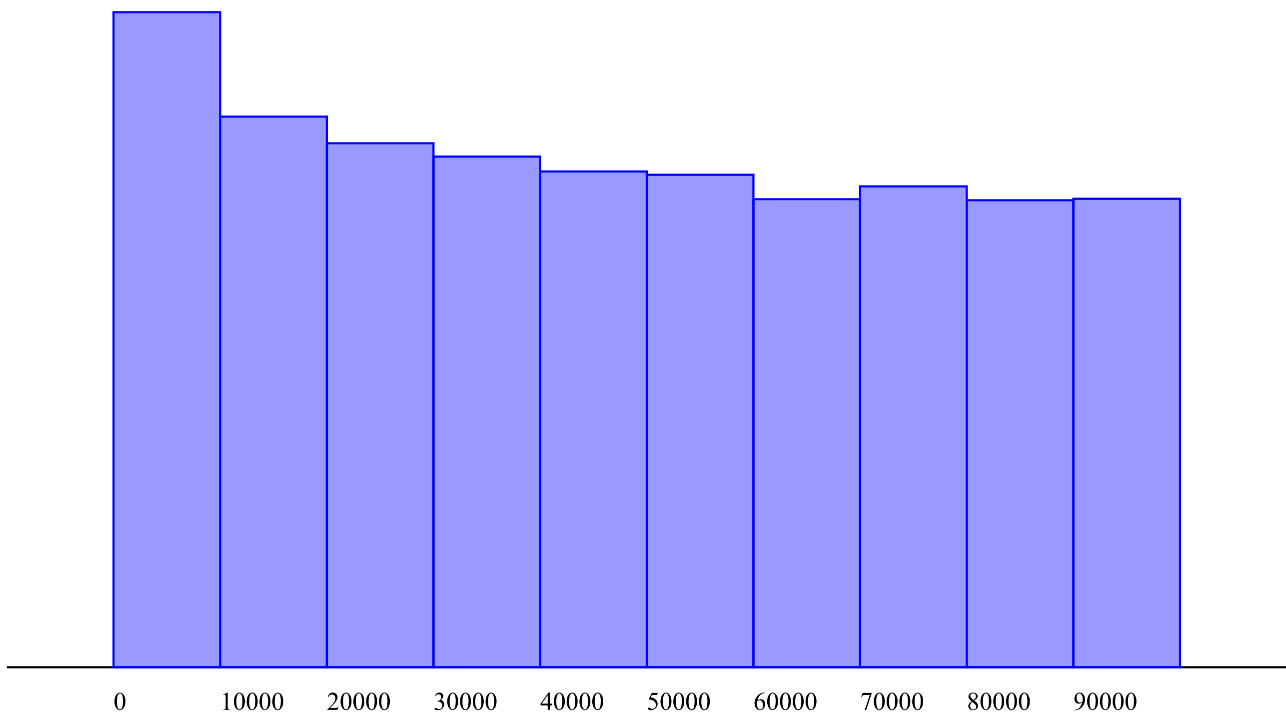
Le résultat :

```
Algorithme lancé
297.3036151654055

Algorithme exécuté en 79 millisecondes
```

On peut aussi préciser un peu la répartition des nombres premiers entre 2 et 100000 par un histogramme. La liste des nombres premiers sera plus longue à construire puisqu'il y a plus de nombres premiers :

```
premiers = [2..100000]
for d in [2..350]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
histogramme premiers, 0, 100000, 10, 1600
```

L'histogramme montre qu'il y a plus de petits nombres premiers que de grands nombres premiers. C'est l'idée directrice de ce qui va suivre :



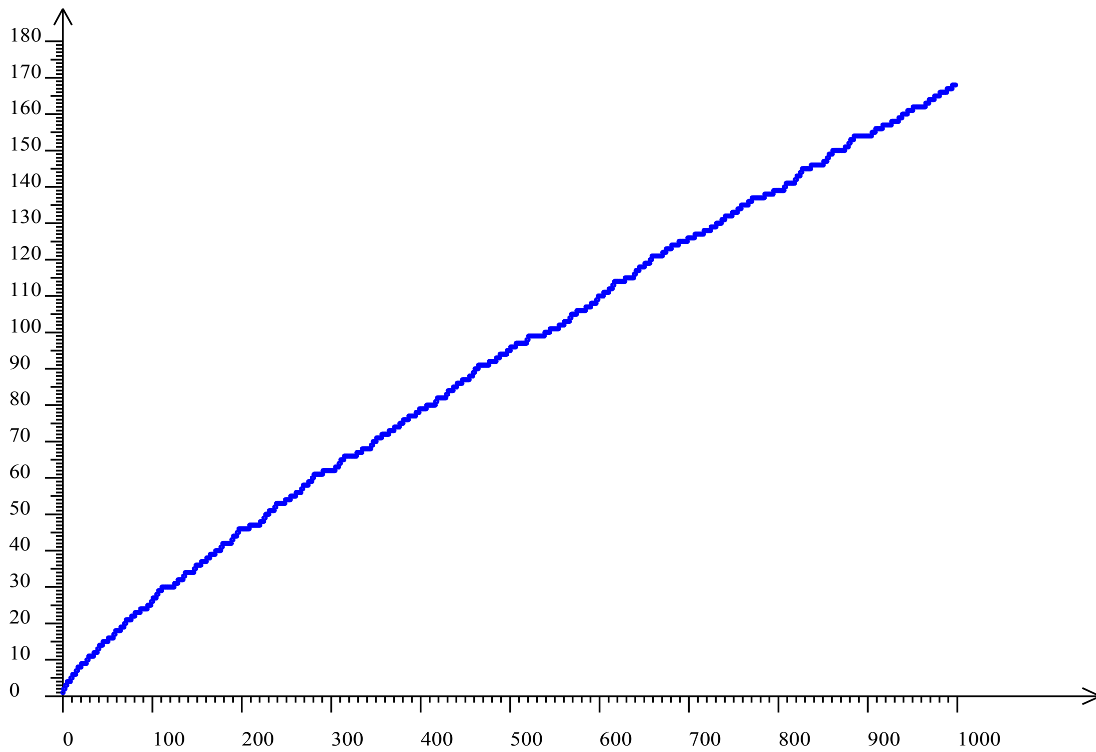
En fait, plus N est grand, et plus la proportion de nombres premiers inférieurs à N est petite. Cette proportion tend vers 0 d'ailleurs, comme on va le voir ci-dessous. Alors on ne va pas se concentrer sur les fréquences cumulées mais sur les effectifs cumulés : Le nombre de nombres premiers inférieurs ou égaux à N, que l'on note π(N) :

3. Effectifs cumulés

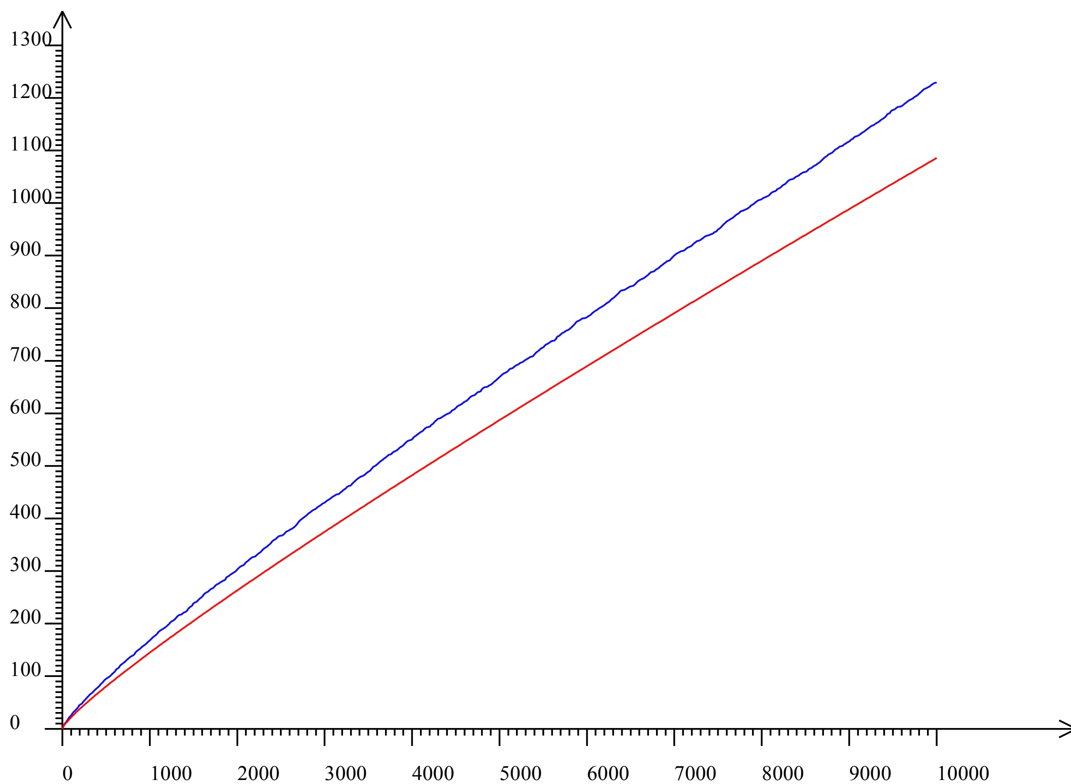
Pour calculer π(n), on peut juste compter les nombres premiers dans la liste des n premiers nombres entiers. Mais comme cette liste est incluse dans la liste des entiers jusqu'à 1002, on peut se contenter d'extraire cette sous-liste avant de compter son effectif (« length »), qui est π(n). Cet algorithme est défini comme fonction de n :

```
premiers = [2..1002]
for d in [2..39]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)

pi = (n) ->
    liste = (x for x in premiers when x<=n)
    liste.length

dessineSuite (pi(x) for x in [2..1000]), 1000, 0, 180, 1, "blue"
```

La variable liste est locale : Elle n'est connue que de la fonction pi, qui ne fait que retourner sa valeur. L'exécution de ce script représente graphiquement la fonction π(n) :



Cette fonction semble suivre une tendance que Gauss et Legendre ont conjecturée proportionnelle à $\frac{x}{\ln(x)}$. C'est en fait l'énoncé du théorème des nombres premiers[4], qu'on peut vérifier expérimentalement en représentant graphiquement cette fonction (en rouge, superposée au graphique) :

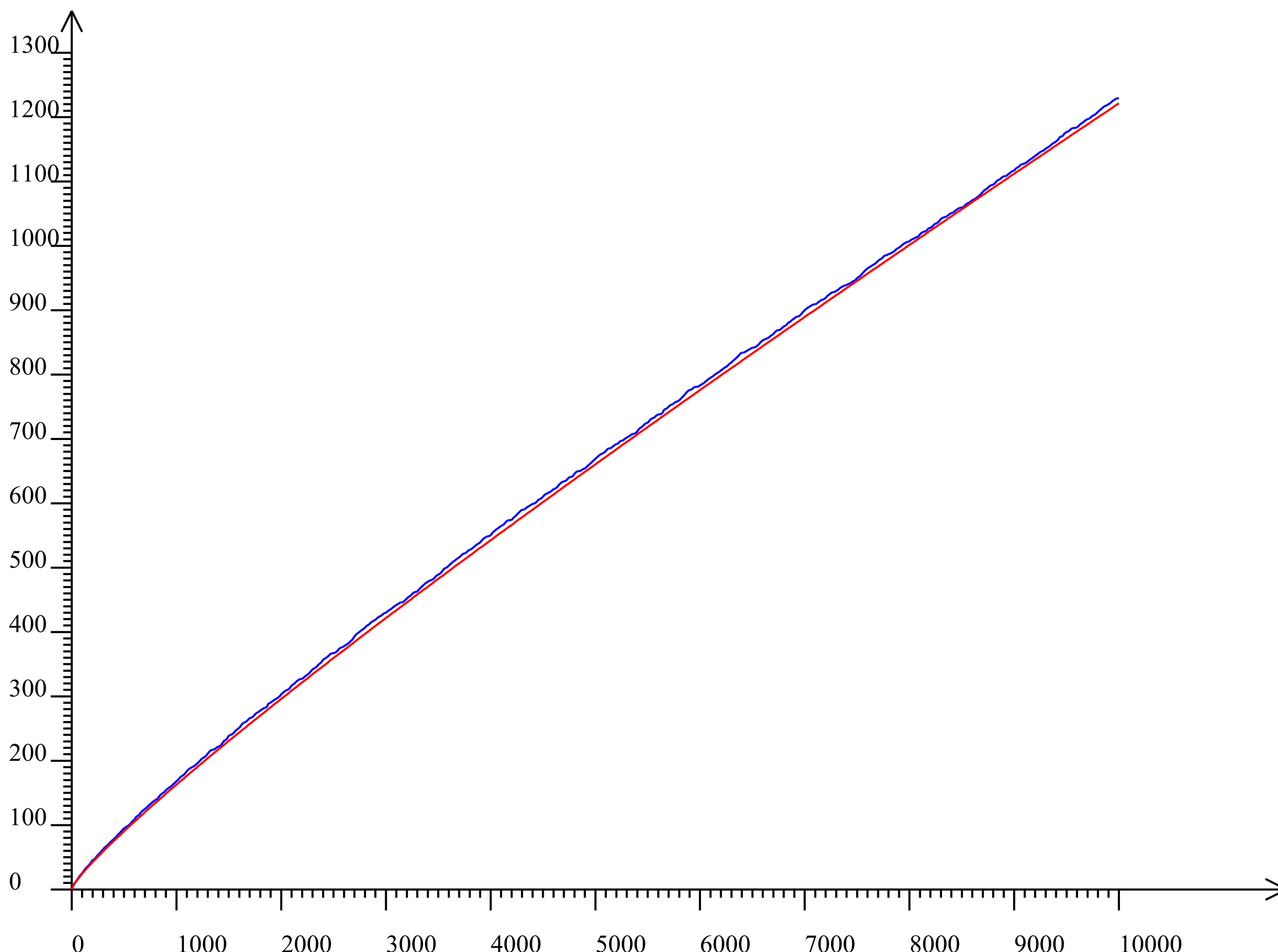


4 Démontré en même temps par Hadamard et De la Vallée Poussin, en 1896

En fait, le tracé bleu ci-dessus est plus fin que la représentation graphique de la suite, parce que c'est une représentation graphique de fonction qui est faite. On constate aussi qu'on est allé jusqu'à 10000 :

```
premiers = [2..10002]
for d in [2..100]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
pi = (n) ->
    liste = (x for x in premiers when x<=n)
    liste.length

dessineFonction pi, 0, 10000, 0, 1300, 'blue'

dessineFonction ((x)->x/ln(x)), 0, 10000, 0, 1300, 'red'
```

On peut aussi comparer avec la fonction $1{,}125\frac{x}{\ln(x)}$ en représentant celle-ci en rouge, superposée à la représentation graphique de π(x) :



Le théorème des nombres premiers donne une meilleure approximation de π(x) : $\int_2^x \frac{dt}{\ln(t)}$. C'est en cherchant à affiner cette approximation que Riemann a émis la célèbre conjecture qui porte son nom.

II/Sommes des carrés

Autre étude de cas, la répartition des sommes de deux carrés : Comme il existe des nombres qui sont premiers et d'autres qui sont composés, il existe des nombres qui sont sommes de deux carrés et d'autres qui ne le sont pas. La question de la proportion relative des deux catégories de nombres se pose alors comme pour les nombres premiers.

1. Liste des sommes de deux carrés

2 n'est pas seulement un nombre premier, c'est aussi la somme de deux carrés : $2=1^2+1^2$. Par contre 3 n'est pas somme de deux carrés, pas plus que 4 (on ne compte pas 2^2+0^2 parce que, par analogie avec les nombres premiers, on se restreint aux carrés non nuls). Mais 5 lui, est la somme de deux carrés : $5=4^2+1^2$. On peut vérifier que 6 et 7 ne sont pas sommes de carrés mais $8=2^2+2^2$, si.

On va noter $\sigma(n)$ le nombre de sommes de deux carrés inférieures ou égales à n, ce qui fait que $\sigma(10)=3$ (les trois nombres en question sont 2, 5 et 8).

Pour construire la liste des sommes de deux carrés, on boucle sur le premier carré et, dans la boucle, sur le second carré, et à chaque passage dans la boucle intérieure, on ajoute à un tableau initialement vide la somme des deux carrés. Mais cela produit une liste de nombres, qui est inexploitable car

- les nombres sont dans le désordre
- et il y a des doublons[5].

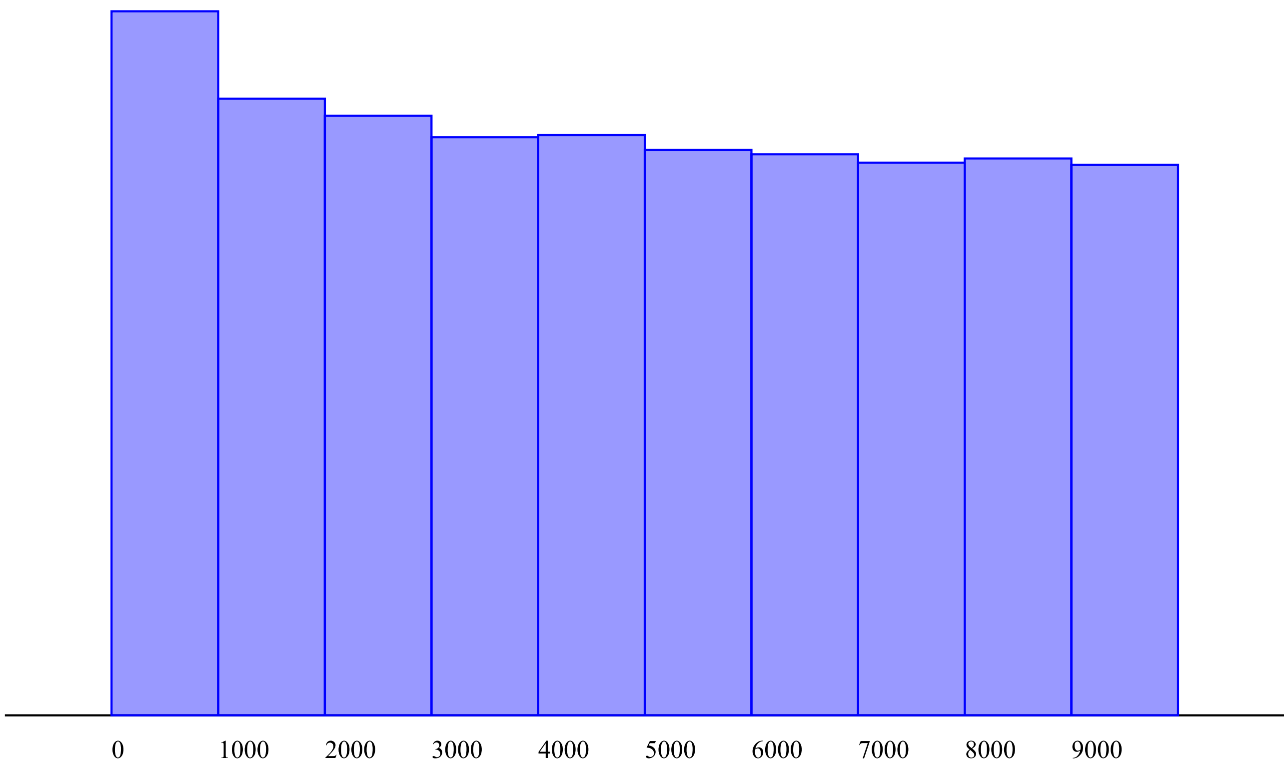
Alors, une fois la liste construite, on la trie avec « sort[6] » (en anglais, to sort veut dire trier), puis on enlève les doublons avec « unique » :

```
somcar = []
for n in [1..100]
    for m in [1..100]
        somcar.push m*m+n*n
somcar.sort (x,y)->x-y
somcar = somcar.unique()
histogramme somcar, 0, 10000, 10, 400
```

L'histogramme montre qu'il y a proportionnellement plus de petites sommes de deux carrés que de grandes sommes de deux carrés, comme avec les nombres premiers. La ressemblance entre les deux histogrammes est à l'origine des investigations qui suivent. Comparer l'histogramme obtenu page suivante avec celui de la page 5 (nombres premiers jusqu'à 10000) : Même la légère surabondance entre 8000 et 9000 est là !

5 Par exemple $50=5^2+5^2=7^2+1^2$ ou $65=7^2+4^2=8^2+1^2$...

6 Pour que tous les navigateurs internet trient correctement les entiers, il est nécessaire de fournir une fonction de tri, qui, à deux nombres x et y, associe leur différence. Aucune variable n'étant affectée par cette fonction, il s'agit d'une λ-fonction, notée en CoffeeScript `(x,y)->x-y`

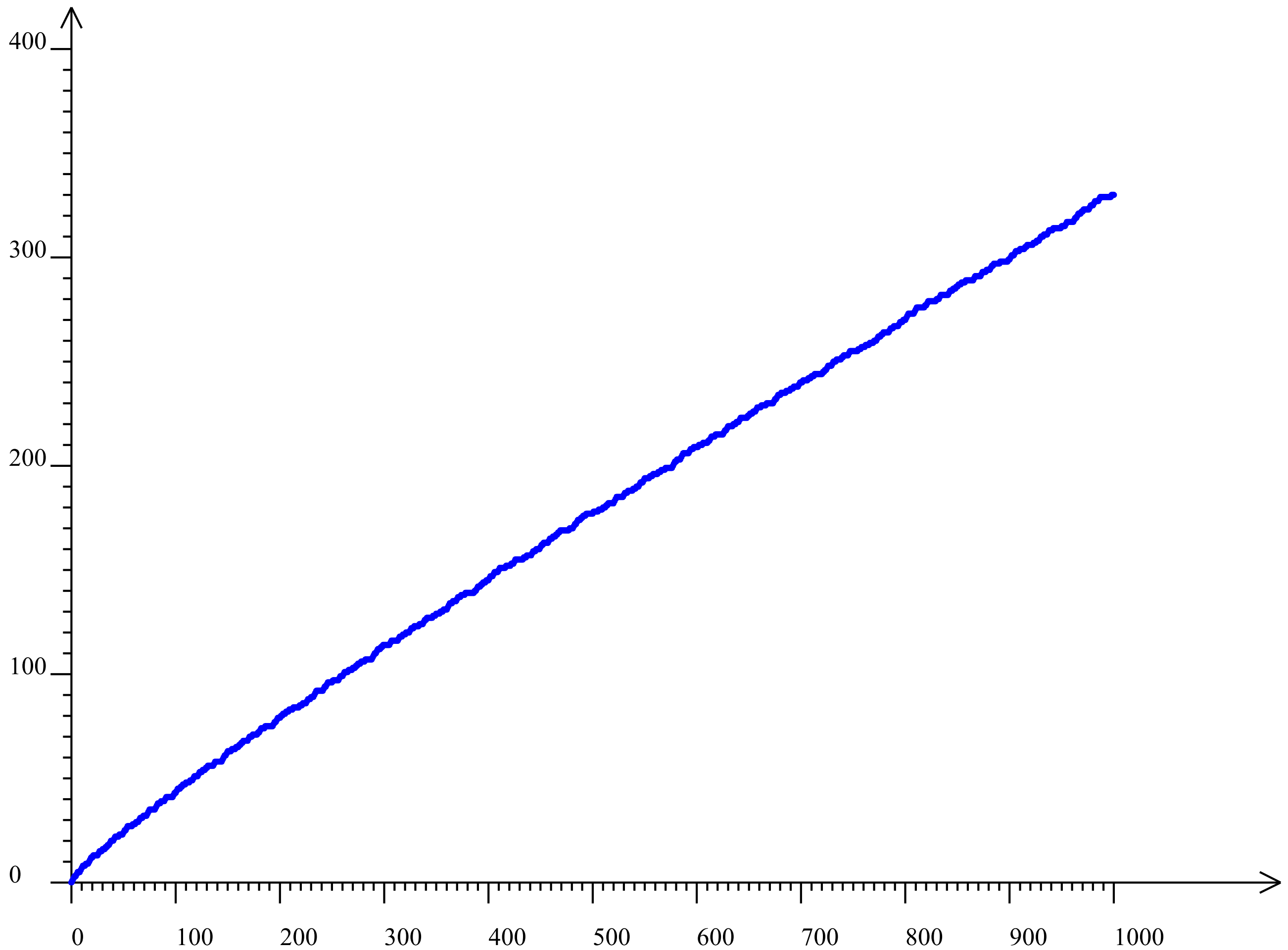


2. Répartition des sommes de deux carrés

Pour calculer la fonction σ(n), on fait comme avec les nombres premiers : On constitue une liste formée des sommes de deux carrés qui sont inférieures à n, et la fonction retourne la taille de cette liste (« length »). On peut gagner un facteur 2 sur le temps d'exécution en ne bouclant que jusqu'à n au lieu de 100. Comme σ(n) est une suite, on peut la représenter comme telle. Le script est plus long que l'algorithme d'Eratosthène :

```
somcar = []
for n in [1..100]
   for m in [1..n]
      somcar.push m*m+n*n
somcar.sort (x,y)->x-y
somcar = somcar.unique()

sigma = (n) ->
    liste = (x for x in somcar when x<n)
    liste.length
dessineSuite (sigma(x) for x in [0..1000]), 1000, 0, 400, 1, "blue"
```

La représentation graphique de la suite ressemble beaucoup à celle de la répartition des nombres premiers :

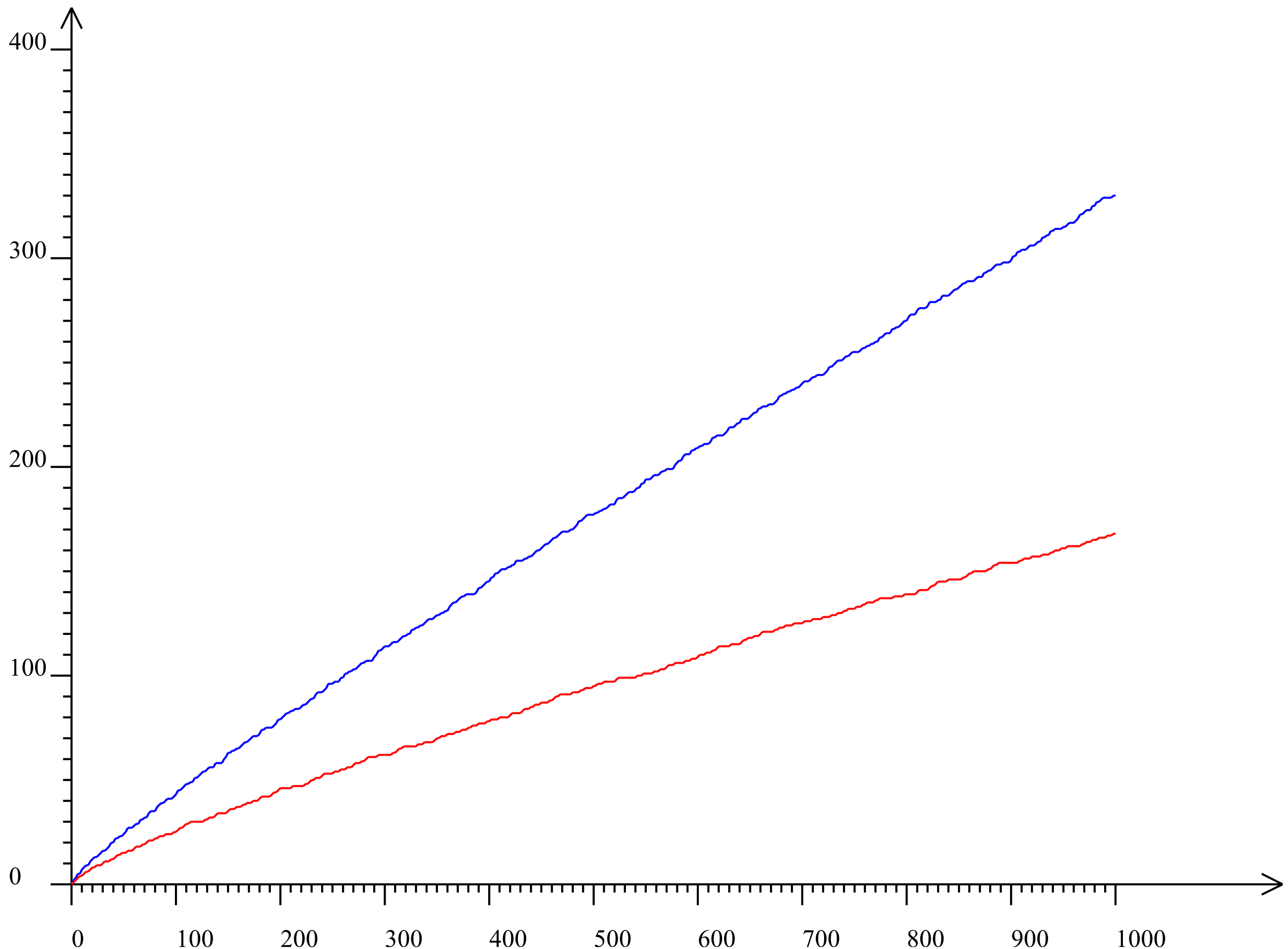


La représentation graphique de la fonction σ(x) où x est réel, accompagnée de celle de la fonction π(x) (en rouge) montre ce qui ressemble à une équivalence.

Pour représenter graphiquement la fonction, on peut faire

```
somcar = []
for n in [1..100]
   for m in [1..n]
      somcar.push m*m+n*n
somcar.sort (x,y)->x-y
somcar = somcar.unique()


#sigma est en fait une fonction de la variable réelle :
sigma = (n) ->
    liste = (x for x in somcar when x<n)
    liste.length

#dessin de la courbe bleue :
dessineFonction sigma, 0, 1000, 0, 400, "blue"
```

Le graphique a un air de déjà vu :

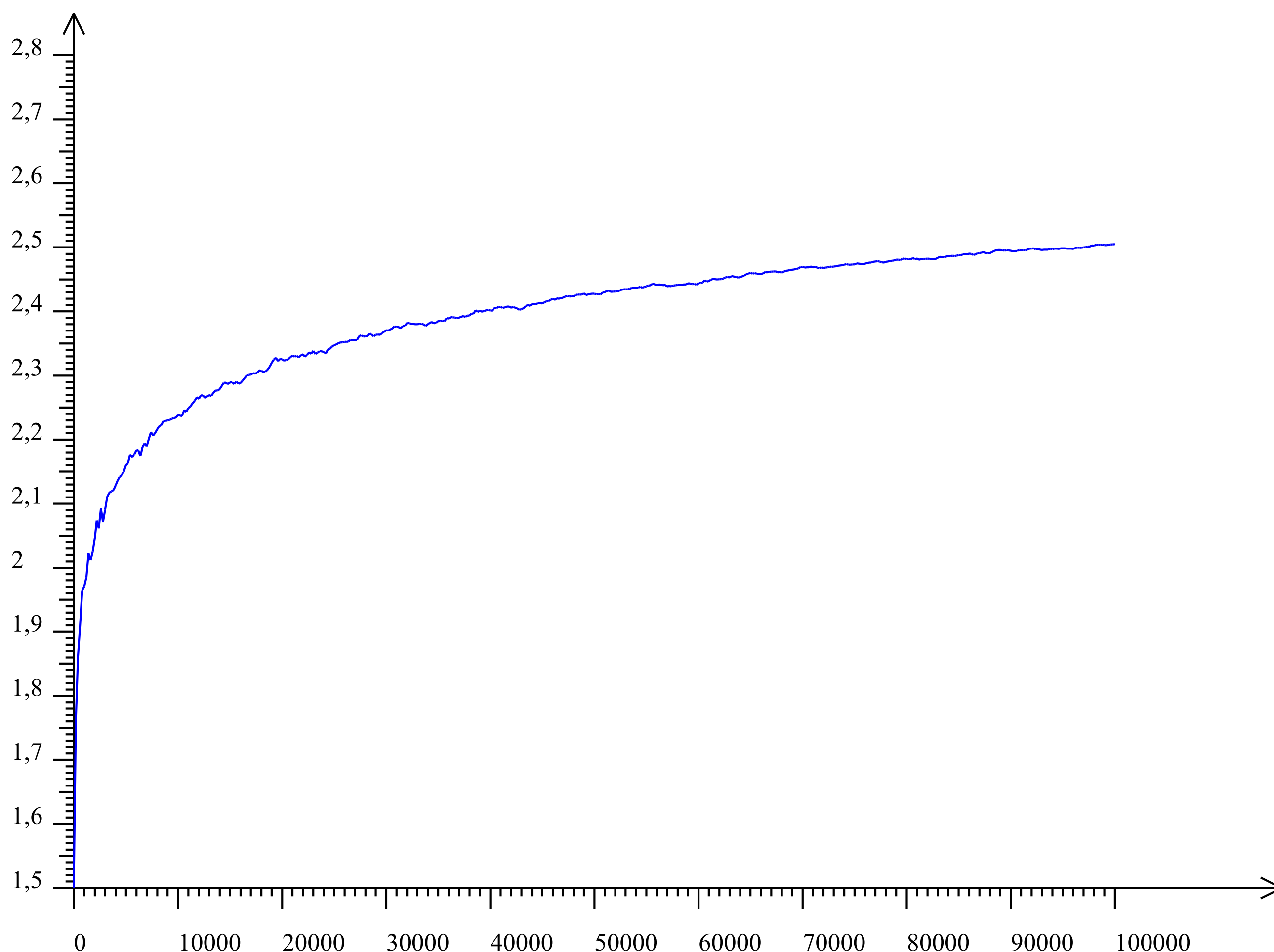


On voit qu'il y a plus de sommes de deux carrés que de nombres premiers. Mais combien de fois plus ?

3. Comparaison avec les nombres premiers

L'idée suivante consiste à représenter graphiquement le quotient de la fonction $\sigma(x)$ par la fonction $\pi(x)$. Le script permettant cela est essentiellement du copié-collé des scripts précédents :

```
somcar = []
for n in [0..316]
   for m in [0..316]
      somcar.push m*m+n*n
somcar.sort (x,y)->x-y
somcar = somcar.unique()
sigma = (n) ->
    liste = (x for x in somcar when x<n)
    liste.length
premiers = [2..100000]
for d in [2..316]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
pi = (n) ->
    liste = (x for x in premiers when x<=n)
    liste.length
f = (x) -> sigma(x)/pi(x)
dessineFonction f, 3, 100000, 0, 4, "blue"
```



Il semble[7] qu'il y ait moins de trois fois plus de sommes de deux carrés que de nombres premiers. La courbe ci-dessus est tracée à partir des sommes de carrés éventuellement nuls, qui se comportent visiblement comme les sommes de carrés non nuls.

III/ Nombres chanceux d'Ulam

La ressemblance entre la répartition statistique des nombres premiers et celle des nombres chanceux d'Ulam est importante parce que les nombres chanceux sont définis par un algorithme de crible, et que cela a amené Ulam à se demander si la répartition des nombres premiers ne serait pas la conséquence de l'algorithme d'Eratosthène plutôt que celle de la propriété de primalité. Autrement dit, la conjecture de Riemann pourrait être due à la statistique plutôt qu'à l'arithmétique.

1. Calcul des nombres chanceux

L'algorithme définissant les nombres chanceux est le suivant :

- On commence par lister tous les entiers (comme d'Alembert, on en prend beaucoup pour faire croire qu'on va jusqu'à l'infini) à partir de 2 ;
- On enlève à cette longue liste un nombre sur 2 ;

7 On serait même tenté de conjecturer que la fonction représentée graphiquement ci-dessus tende vers e à l'infini. Cette conjecture émise numériquement reste bien entendu à prouver (ou infirmer)...

- Comme le plus petit entier rescapé est 3, on enlève un nombre sur 3 de la liste restante ;
- Comme le plus petit entier rescapé est 7, on enlève à la liste restante, un entier sur 7 ;
- etc.

Une fois qu'il ne reste plus que les rescapés, la liste qui reste est celle des nombres chanceux d'Ulam. Pour calculer cette liste on fait donc un peu comme avec les nombres premiers :

```
crible = (2*k+1 for k in [0..500])
rang = 1
while crible[rang] < crible.length
    crible = (crible[k] for k in [0..crible.length] when (k+1)%crible[rang] isnt 0)
    rang++
```

La variable « rang » est l'indice du pas de la progression selon laquelle on élimine les nombres malchanceux. Cet indice est incrémenté par rang++ pour ne pas toucher à un rescapé, et crible[rang] est donc le pas. Lorsque k+1 n'est pas divisible par ce pas, on garde crible[k].

Il subsiste un problème : Malgré les précautions que l'on a prises, certains éléments du tableau « crible » sont vides. On ne garde alors que les éléments différents de « undefined » pour avoir un tableau vraiment numérique :

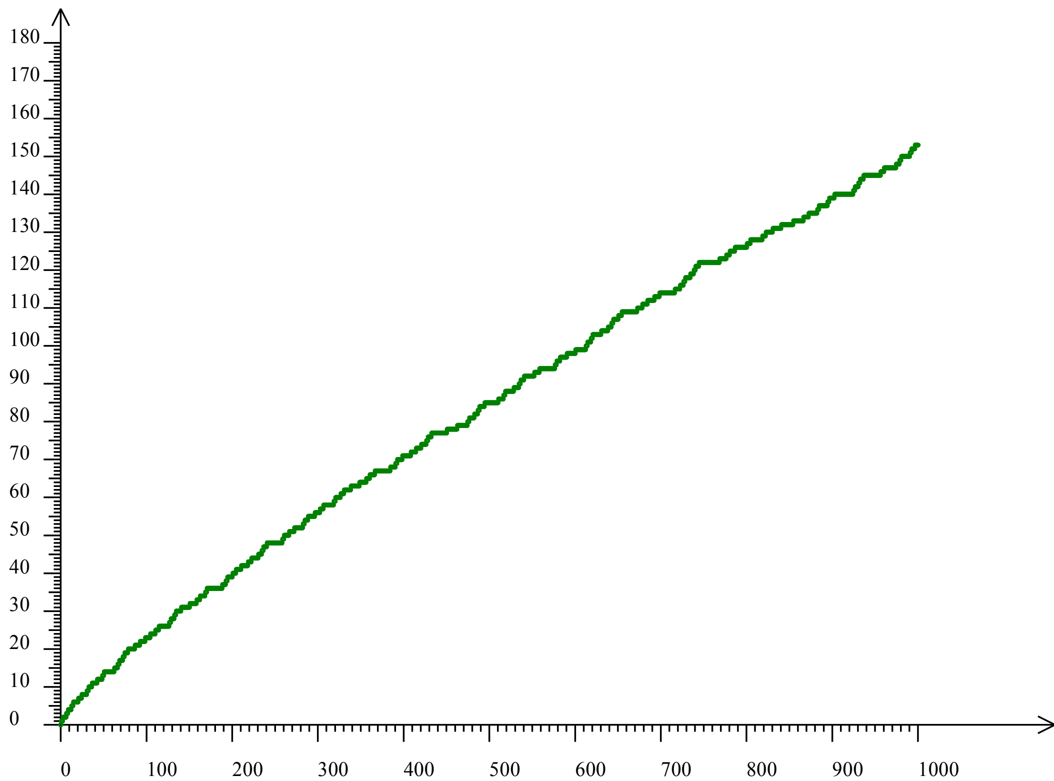
```
crible = (2*k+1 for k in [0..500])
rang = 1
while crible[rang] < crible.length
    crible = (crible[k] for k in [0..crible.length] when (k+1)%crible[rang] isnt 0)
    rang++
crible = [x for x in crible when x isnt undefined][0]
```

2. Répartition des nombres chanceux

Ensuite on définit une fonction similaire à $\pi(x)$ que l'on peut représenter graphiquement comme une suite, en vert pour la distinguer de celle des nombres premiers :

```
crible = (2*k+1 for k in [0..500])
rang = 1
while crible[rang] < crible.length
    crible = (crible[k] for k in [0..crible.length] when (k+1)%crible[rang] isnt 0)
    rang++
crible = [x for x in crible when x isnt undefined][0]
pi = (n) ->
    liste = (x for x in crible when x<=n)
    liste.length
dessineSuite (pi(n) for n in [0..1000]), 1000, 0, 180, 1, 'green'
```

La représentation graphique a, elle aussi, un air de déjà-vu. Il faut dire que cette fonction est, elle aussi, équivalente à $\frac{x}{\ln(x)}$. Voici la représentation graphique :



3. Comparaison avec les nombres premiers

Et voici, comme une synthèse, les représentations graphiques du nombre de nombres premiers inférieurs à x (en bleu), du nombre de nombres chanceux inférieurs à x (en vert) et du nombre de sommes de deux carrés inférieurs à x (en rouge) :

400
300
200
100
0
0 100 200 300 400 500 600 700 800 900 1000

IV/ Diviseurs

Maintenant on va faire des statistiques non pas sur des ensembles de nombres entiers, mais sur les diviseurs d'un nombre entier. On a alors une infinité de « petites » séries statistiques.

1. Liste des diviseurs

La liste des diviseurs est calculée au sein d'une fonction, qui, à un nombre entier non nul, associe un tableau JavaScript, qui est la liste des diviseurs :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)

affiche diviseurs 28
```

Cette liste de diviseurs étant une série statistique, on peut donc calculer

- le diviseur moyen avec `laMoyenneDe` (le diviseur moyen de 28 est 9+1/3, le saviez-vous?) ;
- l'écart-type avec `lEcartTypeDe` (pour 28, l'écart-type est proche de la moyenne) ;
- le diviseur médian avec `laMédianeDe` (le diviseur médian de 28 est 5,5) ;
- les quartiles avec `lePremierQuartileDe` et `leDernierQuartileDe` (pour 28, ce sont 2 et 14 ; ces renseignements permettent d'ailleurs de représenter les diviseurs de 28 – ou d'autres entiers – par des boîtes à moustaches)

Voici par exemple comment on calcule la somme des diviseurs de 28 (voir plus bas) :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)

affiche laSommeDe diviseurs 28
```

La réponse donnée par alcoffeethmique est intéressante :

```
Algorithme lancé
56

Algorithme exécuté en 24 millisecondes
```

La somme des diviseurs de 28 est égale au double de 28 : Cela signifie que 28 est un nombre parfait, et ces nombres sont plutôt rares (le suivant est 8 128).

2. Fonctions de diviseurs

On note $\sigma_k(n)$ la somme des puissances k-ièmes des diviseurs de n. On définit ainsi une infinité de fonctions, une pour chaque valeur de k. Particulièrement remarquables, mais non étudiées ici, sont :

- $\sigma_5(n)$ (somme des puissances cinquièmes de n)
- $\sigma_{11}(n)$ (somme des puissances onzièmes de n)

qui servent à calculer la fonction tau de Ramanujan[8]. Mais pour avoir des séries statistiques portant sur des nombres de taille raisonnable, on ira moins loin, et on ne regardera ici que

- $\sigma_0(n)$ (nombre de diviseurs de n),
- $\sigma_1(n)$ (somme des diviseurs de n), et
- $\sigma_2(n)$ (somme des carrés des diviseurs de n)

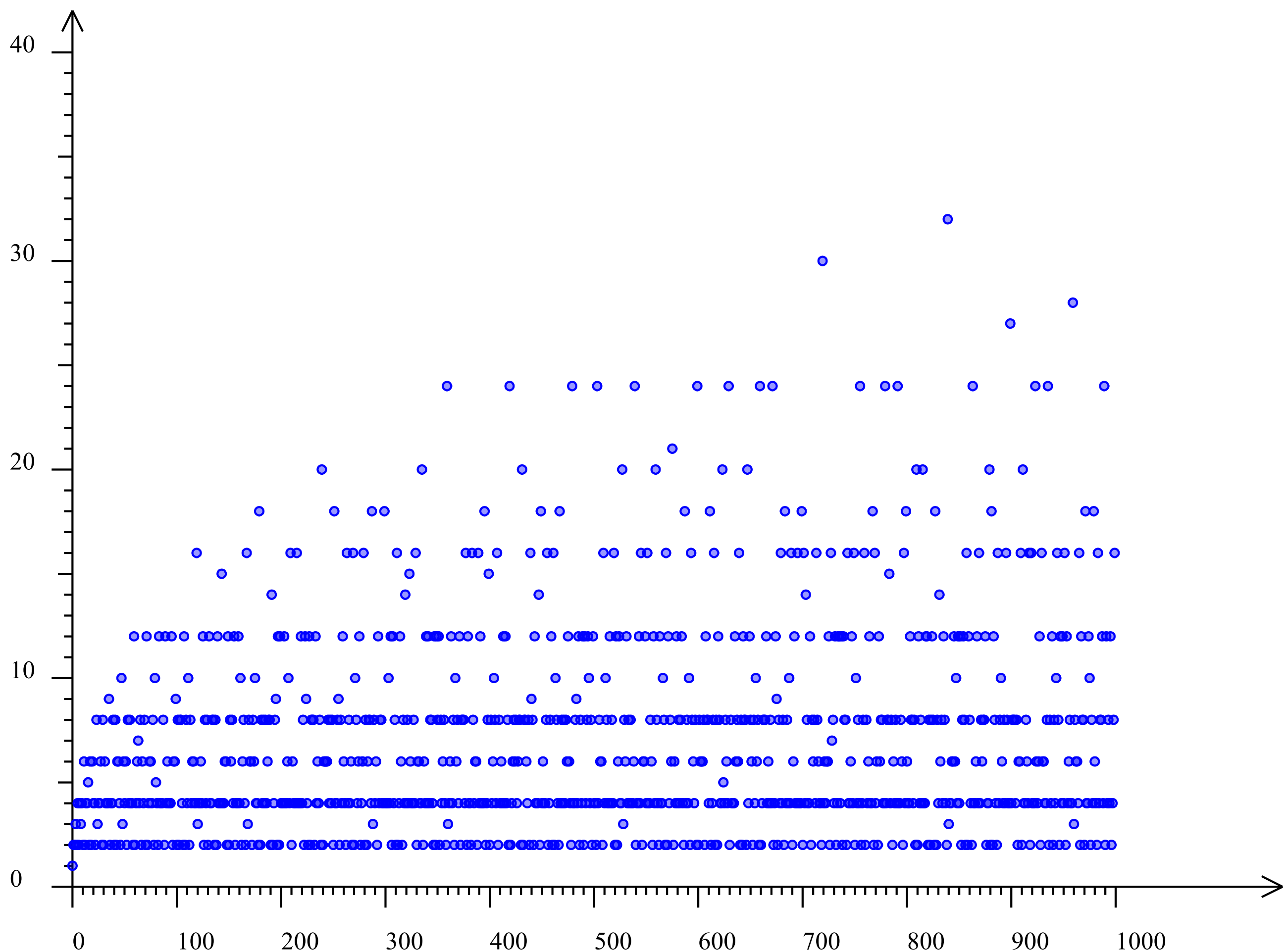
3. Nombre de diviseurs

Comme $k^0=1$, la somme des puissances zéro-ièmes des diviseurs de n n'est autre que le nombre de diviseurs de n. En CoffeeScript cela se fait avec « length », appliqué à la liste des diviseurs vue précédemment :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
tau = (n) ->
    (diviseurs n).length

dessineSuite (tau(n) for n in [1..1000]), 999, 0, 40, 2, 'blue'
```

Ci-dessus la fonction s'appelle « tau » au lieu de σ_0 pour éviter d'avoir à donner un nom de fonction portant un indice, mais aussi parce que c'est souvent ainsi qu'on la note. Attention toutefois à ne pas confondre cette fonction avec la fonction tau de Ramanujan évoquée ci-dessus. Le script ci-dessus représente graphiquement, en fonction de n, le nombre de diviseurs de n :

8 Cette fonction, à ne pas confondre avec le nombre de diviseurs parfois aussi appelé tau, joue un rôle particulièrement tentaculaire en mathématiques, puisque, définie comme une transformée de Fourier de fonction modulaire dans le demi-plan de Poincaré, elle intervient naturellement en géométrie hyperbolique plane. Mais elle a servi également à démontrer l'existence du réseau de Leech en dimension 24, qui est à l'origine d'un empilement particulièrement compact d'hypersphères dans cet espace, mais aussi dans la démonstration du théorème de Wiles. On la trouve même en théorie de la gravitation, via les algèbres de Kac-Moody qui relient les courbes elliptiques au plus grand groupe simple sporadique : Le Monstre de Fischer. Cependant les valeurs prises par cette fonction deviennent vite très grandes ce qui gêne une étude expérimentale de celle-ci. Mais avec Big.js on pourrait, et qui sait, ainsi, gagner le million de dollars promis au premier qui démontrera la conjecture de Birch et Swinnerton-Dyer...



En bas du graphique, on aperçoit des nombres qui n'ont que deux diviseurs : Les nombres premiers. Bien plus rares sont les carrés de nombres premiers, qui ont trois diviseurs. Les nombres ayant 4 diviseurs par contre, sont nettement plus nombreux : Ce sont les produits de deux nombres premiers comme par exemple 15, qui est divisible par 1, 3, 5 et lui-même.

Surprenant : Il y a relativement beaucoup de nombres ayant exactement 24 diviseurs : On en compte 16 parmi les entiers inférieurs ou égaux à 1000 :

```
Algorithme lancé
360,420,480,504,540,600,630,660,672,756,780,7
92,864,924,936,990

Algorithme exécuté en 86 millisecondes
```

On les a trouvés et affichés avec ce script :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
tau = (n) ->
    (diviseurs n).length

affiche (n for n in [1..1000] when tau(n) is 24)
```

On peut donc considérer comme particulièrement exceptionnels en nombres de diviseurs, les entiers ayant plus de 24 diviseurs. On les trouve avec ce script :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
tau = (n) ->
    (diviseurs n).length

affiche (n for n in [1..1000] when tau(n)>24)
```

La liste est formée de 4 nombres :

```
Algorithme lancé
720,840,900,960

Algorithme exécuté en 34 millisecondes
```

Ces nombres ont respectivement 30, 32, 27 et 28 diviseurs. Il faut dire qu'un nombre inférieur à 1000 a, en moyenne, 7,069 diviseurs (le nombre médian de diviseurs étant par contre, 5). Quételet aurait eu sans doute des choses intéressantes à dire sur le nombre moyen, les seuls nombres inférieurs ou égaux à 1000 ayant 7 diviseurs étant

- 64 (les diviseurs étant 1,2,4,8,16,32,64)
- 729 (les diviseurs étant 1,3,9,27,81,243,729)

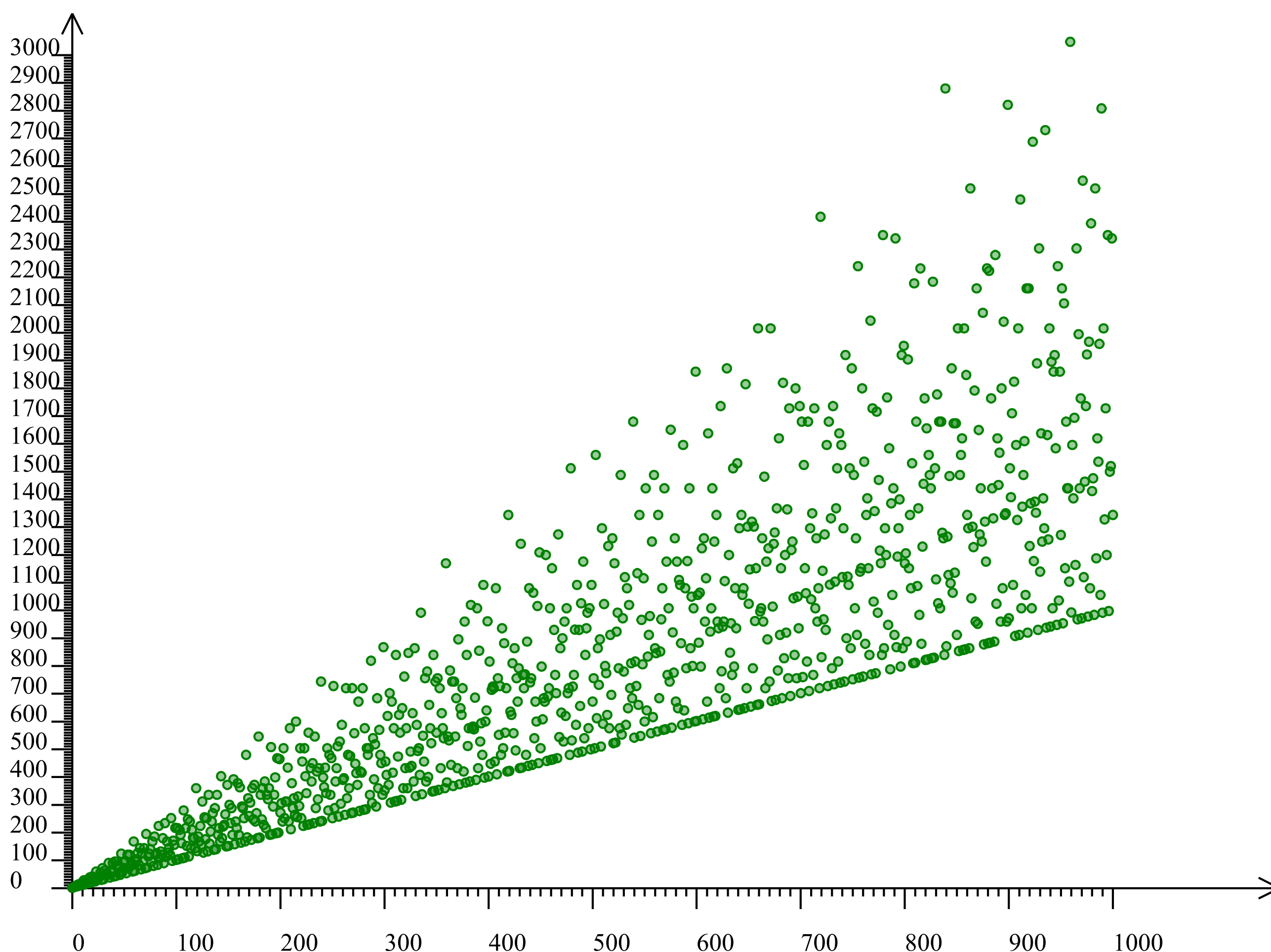
En effet, un nombre ayant 7 diviseurs est la puissance sixième d'un nombre premier p, les diviseurs étant les puissances de p ayant un exposant égal à 0, 1, 2, 3, 4, 5 ou 6.

4. Somme des diviseurs

La somme des puissances unièmes des diviseurs de n est la somme des diviseurs de n, on appellera donc « sigma » (en omettant l'indice 1) ladite somme. On a déjà vu avec l'exemple de 28 comment on peut calculer cette fonction :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
sigma = (n) ->
    laSommeDe (diviseurs n)

dessineSuite (sigma(n) for n in [1..1001]), 1000, 0, 3000, 2,
'green'
```

Le script ci-dessus représente graphiquement la suite des sommes des diviseurs de n, fonction de n, en vert :



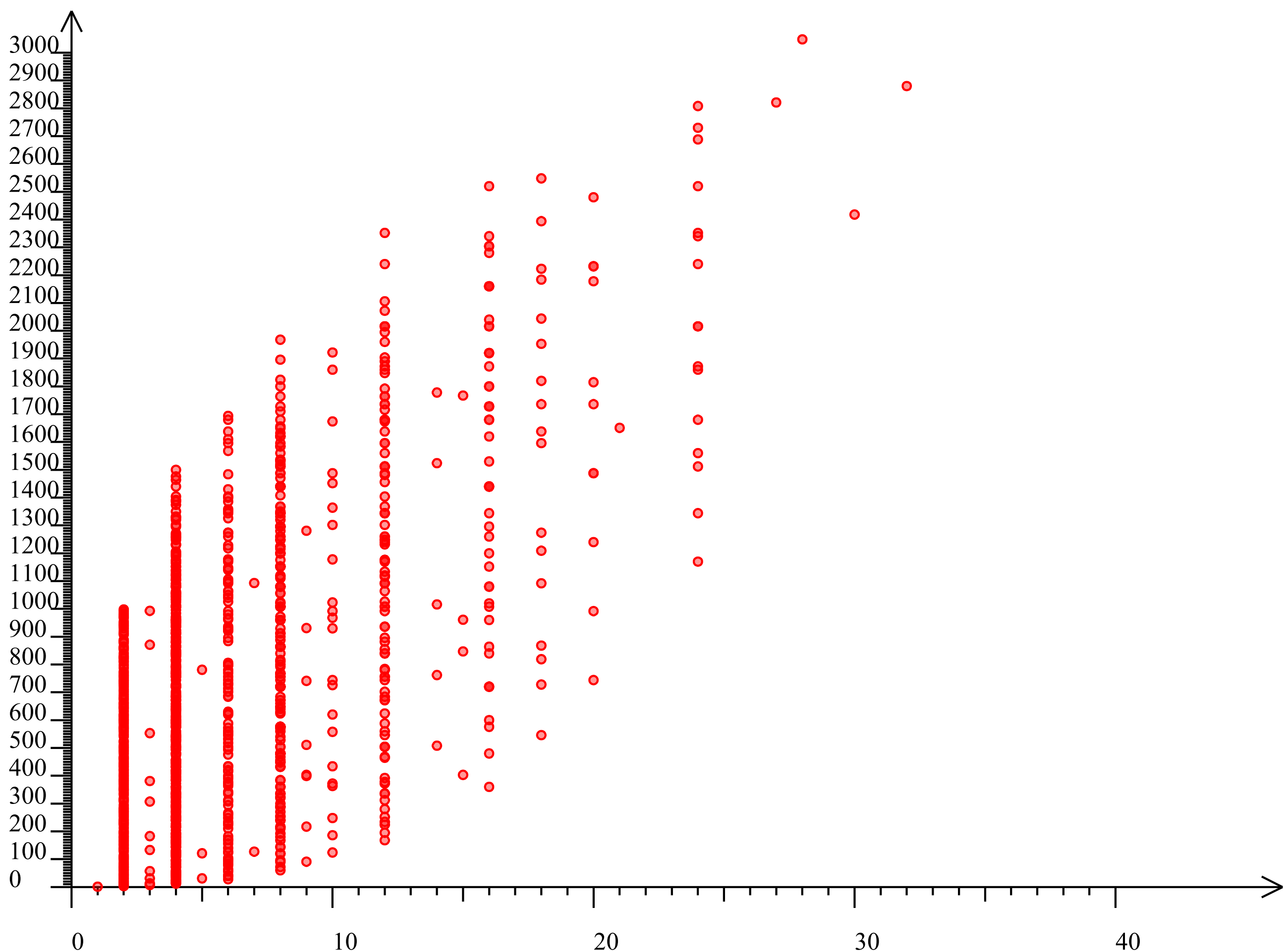
Comme 1 et n sont des diviseurs de n, $\sigma_1(n)$ est au moins aussi grand que n+1. L'égalité est bien entendu atteinte lorsque n est premier et on distingue là encore les nombres premiers en bas du nuage de points, alignés sur la droite d'équation y=x+1. On peut aussi se servir de ce graphique pour estimer les quantités respectives

- de ***nombres déficients*** (les points situés en-dessous de la droite d'équation y=2x)
- de ***nombres parfaits*** (les points situés sur la droite d'équation y=2x ; seuls 6 et 28 sont inférieurs à 1000 parmi eux)
- de ***nombres abondants*** (les points situés au-dessus de la droite d'équation y=2x)

Mais on va plutôt élaborer sur l'idée intuitive selon laquelle, « plus un nombre a de diviseurs, plus on a de chances que leur somme soit élevée ». La statistique peut y aider en représentant un nuage de points ayant $\sigma_0(n)$ pour abscisse et $\sigma_1(n)$ pour ordonnée. Comme il n'y a pas de fonction « nuage de points » dans *alcoffeethmique*, on fait ça dans une boucle :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
tau = (n) ->
    (diviseurs n).length
sigma = (n) ->
    laSommeDe (diviseurs n)

effaceDessin()
dessineAxes 0, 40, 0, 3000
for n in [1..1000]
    dessineCercle 40+tau(n)/40*500, 440-sigma(n)/3000*400, 2, 'red'
```

Le nuage, représenté en rouge, montre qu'il y a finalement assez peu de corrélation entre le nombre de diviseurs et la somme des diviseurs :



Ceci dit, les points sont quand même disposés dans une ellipse et on peut deviner une ébauche de corrélation (mais avec un faible coefficient de corrélation).

5. Somme des carrés de diviseurs

De même on peut définir la fonction $\sigma_2(n)$, ci-dessous appelée « sigma2 », qui à un entier n, associe la somme des carrés des ses diviseurs. Par exemple $\sigma_2(6)=1^2+2^2+3^2+6^2=1+4+9+36=50$.

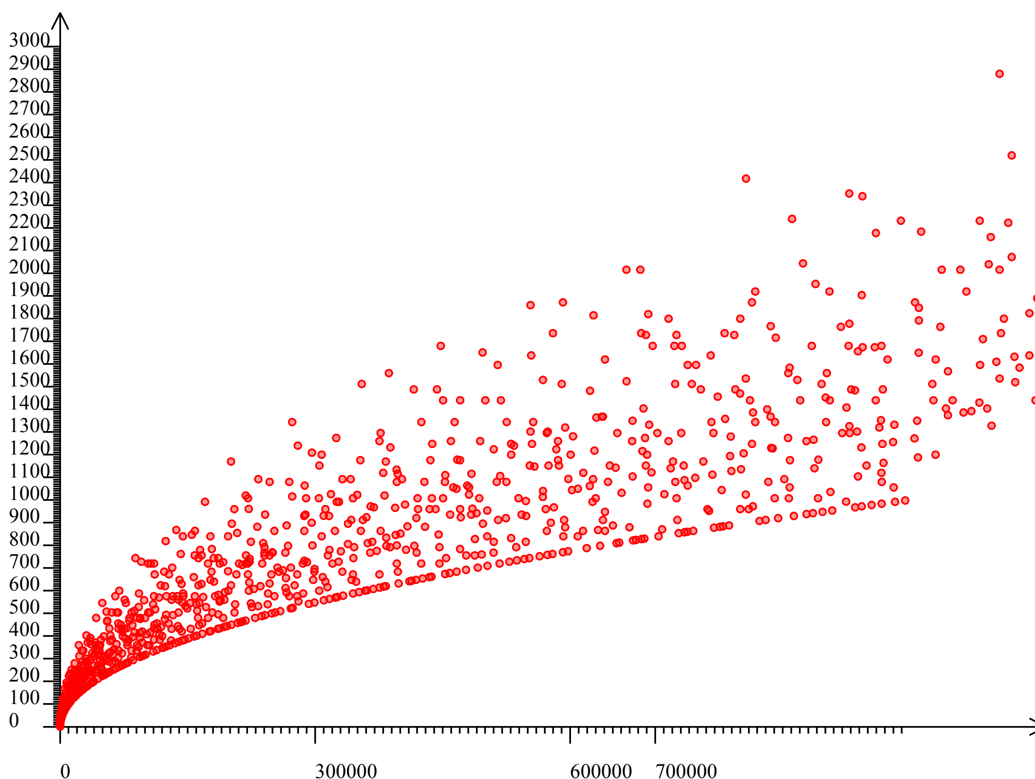
On peut calculer cette fonction ainsi :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
sigma2 = (n) ->
    laSommeDe (x*x for x in diviseurs n)
```

Le nuage de points, si on le représentait, ressemblerait à celui représentant la suite $\sigma_1(n)$, à ceci près que les alignements suivraient des paraboles plutôt que des droites. On va alors plutôt représenter un nuage de points ayant pour abscisse $\sigma_2(n)$ et pour ordonnée $\sigma_1(n)$, avec un script similaire au précédent :

```
diviseurs = (n) ->
    (d for d in [1..n] when n%d is 0)
sigma2 = (n) ->
    laSommeDe (x*x for x in diviseurs n)
sigma = (n) ->
    laSommeDe (diviseurs n)

effaceDessin()
dessineAxes 0, 1000000, 0, 3000
for n in [1..1000]
    dessineCercle 40+sigma2(n)/40*.02, 440-sigma(n)/3000*400, 2,
'red'
```

Voici le nuage de points :



Là encore, on peut voir une faible corrélation, mais cette fois-ci elle n'est plus affine.

V/ Avec les nombres complexes

1. Entiers de Gauss

La notion de nombres premiers s'étend à toute structure algébrique ayant une division, en (re)définissant les nombres premiers comme ceux qui ne sont divisibles que par eux-mêmes et les « unités » de la structure. C'est le cas pour les entiers de Gauss qui sont de la forme m+in avec $i^2=-1$. Le produit d'un entier de Gauss z par son conjugué s'appelle la norme de z. Or il se trouve que cette norme est multiplicative ce qui fait que si elle est première alors z est premier aussi. Chez les entiers de Gauss, les unités (entiers de norme 1) sont 1, i, -1 et -i. Un entier de Gauss est premier si et seulement si son produit par une unité l'est. Cela a pour conséquence que les entiers de Gauss premiers ont une symétrie d'ordre 4, et simplifie les calculs puisque seuls ceux pour lesquels m et n sont positifs, ont besoin d'être testés.

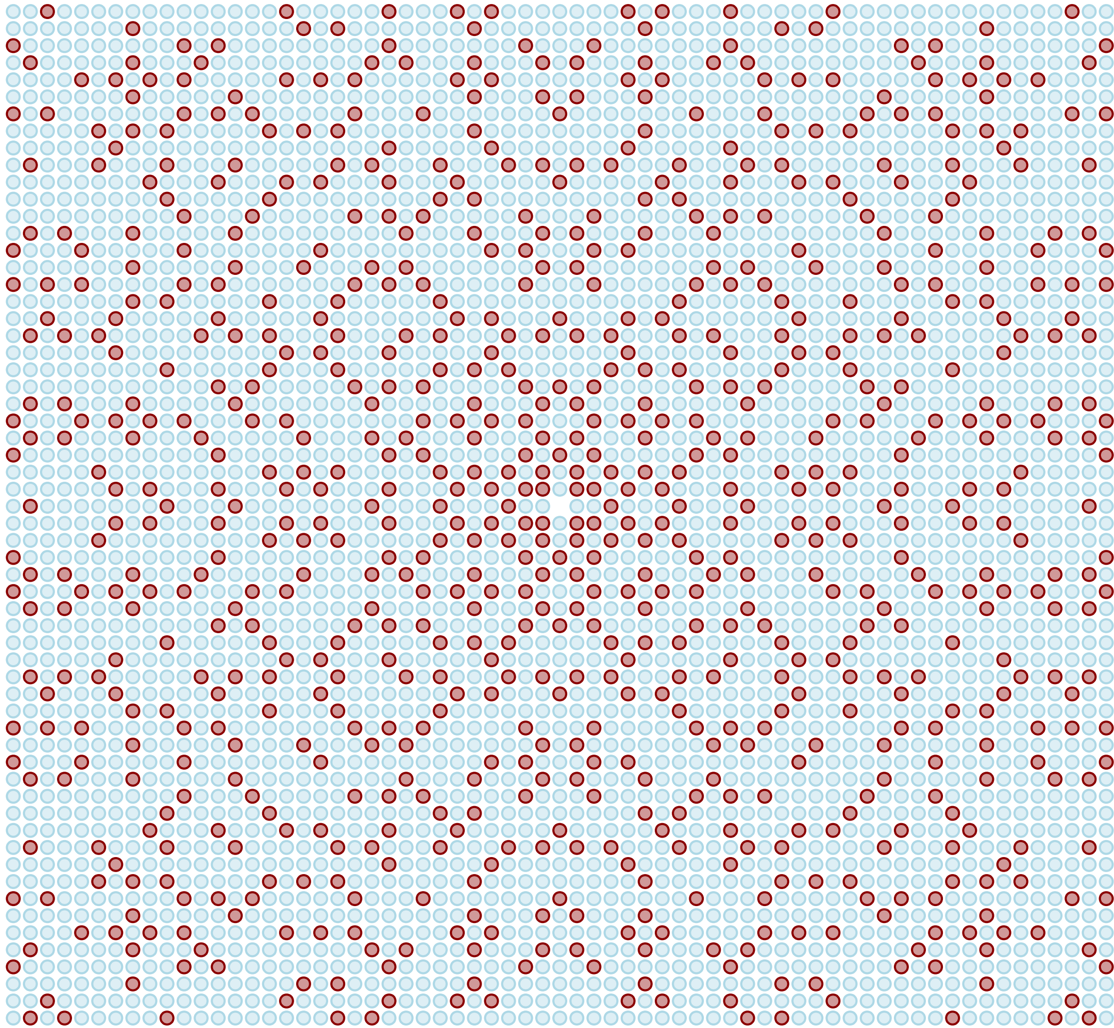
Les entiers de Gauss premiers appartiennent à l'une des catégories suivantes :

- Les entiers naturels premiers et congrus à 3 modulo 4 (n=0).
- Les entiers de Gauss imaginaires dont la norme est première.

Le calcul de la norme de m+in est aisé puisqu'elle est égale à m^2+n^2. Alors pour peu qu'on aie le crible d'Eratosthène sous la main, on peut dessiner les entiers de Gauss non nuls en coloriant sélectivement ceux qui sont premiers. Le script, un peu long, est formé de fonctions plus simples que lui, et juxtaposées :

```
premiers = [2..10000]
for d in [2..100]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
norme = (z) -> z[0]*z[0]+z[1]*z[1]
isPrime = (a) ->
    if a[1] is 0
        a[0] in premiers and a[0]%4 is 3
    else
        norme(a) in premiers

effaceDessin()
e = 8
r = 3
for m in [1..33]
    for n in [0...33]
        if isPrime [m,n]
            couleur = 'darkred'
        else
            couleur = 'lightblue'
        dessineCercle 320+e*m,240-e*n, r, couleur
        dessineCercle 320-e*n,240-e*m, r, couleur
        dessineCercle 320-e*m,240+e*n, r, couleur
        dessineCercle 320+e*n,240+e*m, r, couleur
```

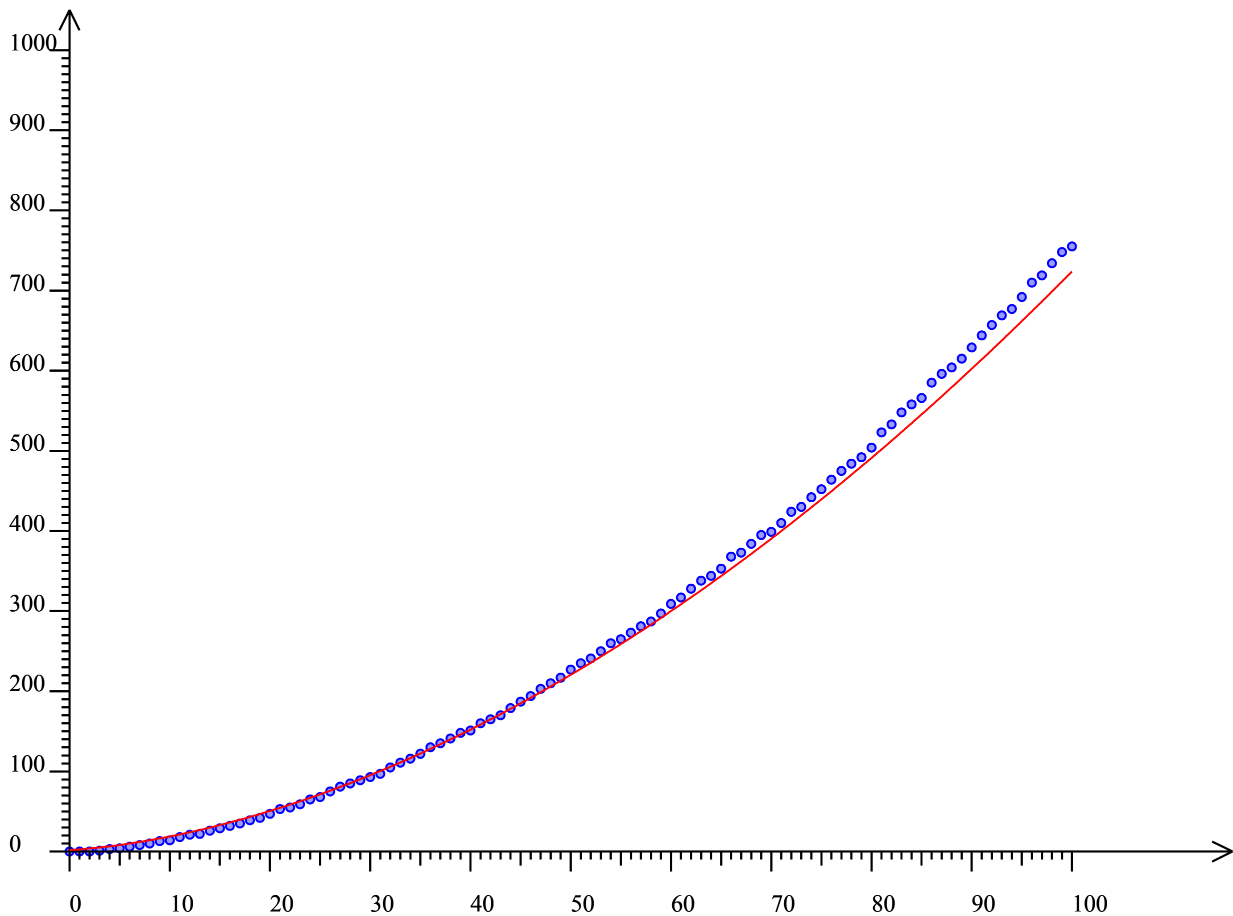
Le dessin obtenu montre que finalement il y a pas mal d'entiers de Gauss premiers, bien que par exemple 2 ne le soit pas puisque 2=(1+i)(1-i) :



Comment généraliser la notion de « nombres premiers inférieurs à x » ? On pourrait imaginer de regarder le nombre de nombres premiers de Gauss inférieurs à une distance donnée. Pour simplifier les calculs on va plutôt regarder le nombre de nombres premiers dont la norme est inférieure à x. On peut la représenter graphiquement par le script suivant :

```
premiers = [2..100000]
for d in [2..400]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)
norme = (z) -> z[0]*z[0]+z[1]*z[1]
isPrime = (a) ->
    if a[1] is 0
        a[0] in premiers and a[0]%4 is 3
    else
        norme(a) in premiers

suite = [0]
for m in [0..100]
    total = 0
    for n in [0...m]
        total++ if isPrime [m,n]
    suite.push suite[suite.length-1] + total

dessineSuite suite, 100, 0, 700, 2, 'blue'
```

La représentation de la suite montre qu'il y a plus de nombres premiers en dimension 2 qu'en dimension 1 :



La courbe représentée en rouge est la représentation graphique de la fonction $\frac{x^2}{3\ln(x)}$:

```
dessineFonction ((x)->x*x/ln(x)/3), 2, 100, 0, 1000, 'red'
```

On voit donc la possibilité de généraliser la conjecture des nombres premiers à la dimension 2. Voire à la dimension 4 puisqu'on peut aussi considérer les « entiers » de la forme a+bi+cj+dk où a, b, c et d sont entiers relatifs et i, j et k sont les quaternions unités. Une autre généralisation des entiers de Gauss est donnée par les entiers d'Eisenstein, où au lieu de considérer les racines quatrièmes de 1, on ne regarde que les racines cubiques :

2. Entiers d'Eisenstein

On pose $\omega=e^{2i\pi/3}$: L'une des racines cubiques de 1 (l'autre est à la fois le conjugué et le carré d'ω). Alors un nombre de la forme m+nω est *un entier d'Eisenstein*. Mais bien qu'il y ait moins de racines cubiques de 1 que de racines quatrièmes de 1, les entiers d'Eisenstein suivent une symétrie d'ordre 6, parce que les racines sixièmes de ω sont des entiers d'Eisenstein. Par exemple $(1+\omega)^2=\omega$ et 1+ω est une racine sixième de 1. Ainsi, il y a 6 entiers d'Eisenstein de norme[9] 1 (unités) : 1, 1+ω, ω, -1, -1-ω et -ω. Comme on peut définir une division sur les entiers d'Eisenstein, on appellera « premier » un entier d'Eisenstein qui n'est divisible que par lui-même et les 6 unités ci-dessus. Alors les entiers premiers d'Eisenstein appartiennent à l'une des deux catégories suivantes :

- les entiers naturels premiers congrus à 2 modulo 3.
- les entiers d'Eisenstein imaginaires dont la norme est première.

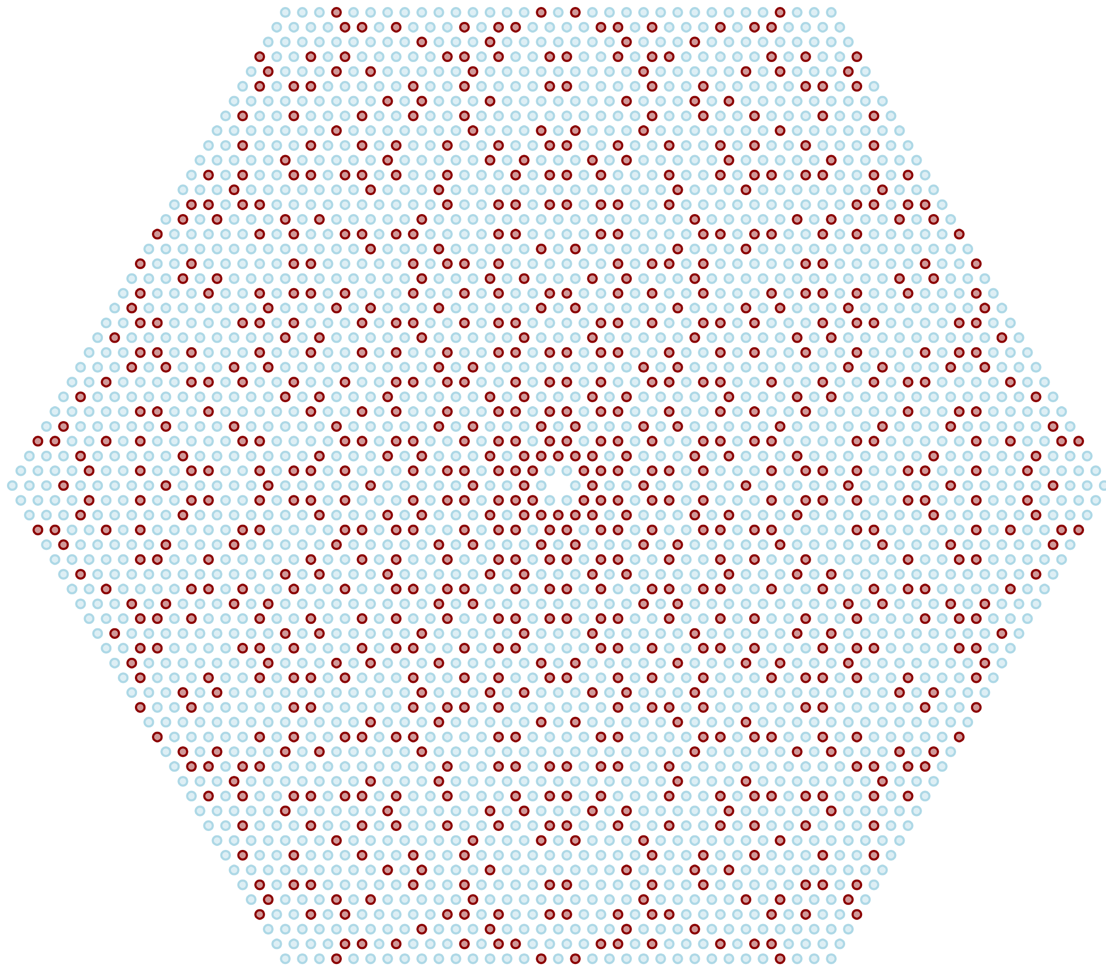
On constate que, vu comme entier d'Eisenstein, 2 est à nouveau premier. Par contre 3 ne l'est plus puisque $3=-(1+2\omega)^2$. Le script dessinant les entiers d'Eisenstein et coloriant en rouge ceux d'entre eux qui sont premiers, utilise la symétrie d'ordre 6 du problème :

```
premiers = [2..10000]
for d in [2..100]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)

isPrime = (a) ->
    if a[1] is 0
        a[0] in premiers and a[0]%3 is 2
    else
        a[0]*a[0]-a[0]*a[1]+a[1]*a[1] in premiers

effaceDessin()
e = 4
r = 2
for m in [1..32]
    for n in [0...m]
        if isPrime [m,n]
            couleur = 'darkred'
        else
            couleur = 'lightblue'
        dessineCercle 320+m*2*e-n*e,240-n*e*racine(3), r, couleur
        dessineCercle 320-m*2*e+n*e,240+n*e*racine(3), r, couleur
        dessineCercle 320-m*e-n*e,240-(m-n)*e*racine(3), r, couleur
        dessineCercle 320+m*e+n*e,240+(m-n)*e*racine(3), r, couleur
        dessineCercle 320+m*e-n*2*e,240-m*e*racine(3), r, couleur
        dessineCercle 320-m*e+n*2*e,240+m*e*racine(3), r, couleur
```

On ne dessine ainsi que les entiers d'Eisenstein qui sont dans un hexagone (partie réelle limitée).

Comme pour les entiers de Gauss premiers, il y a abondance des entiers d'Eisenstein premiers (il y a beaucoup de rouge dans la figure) :

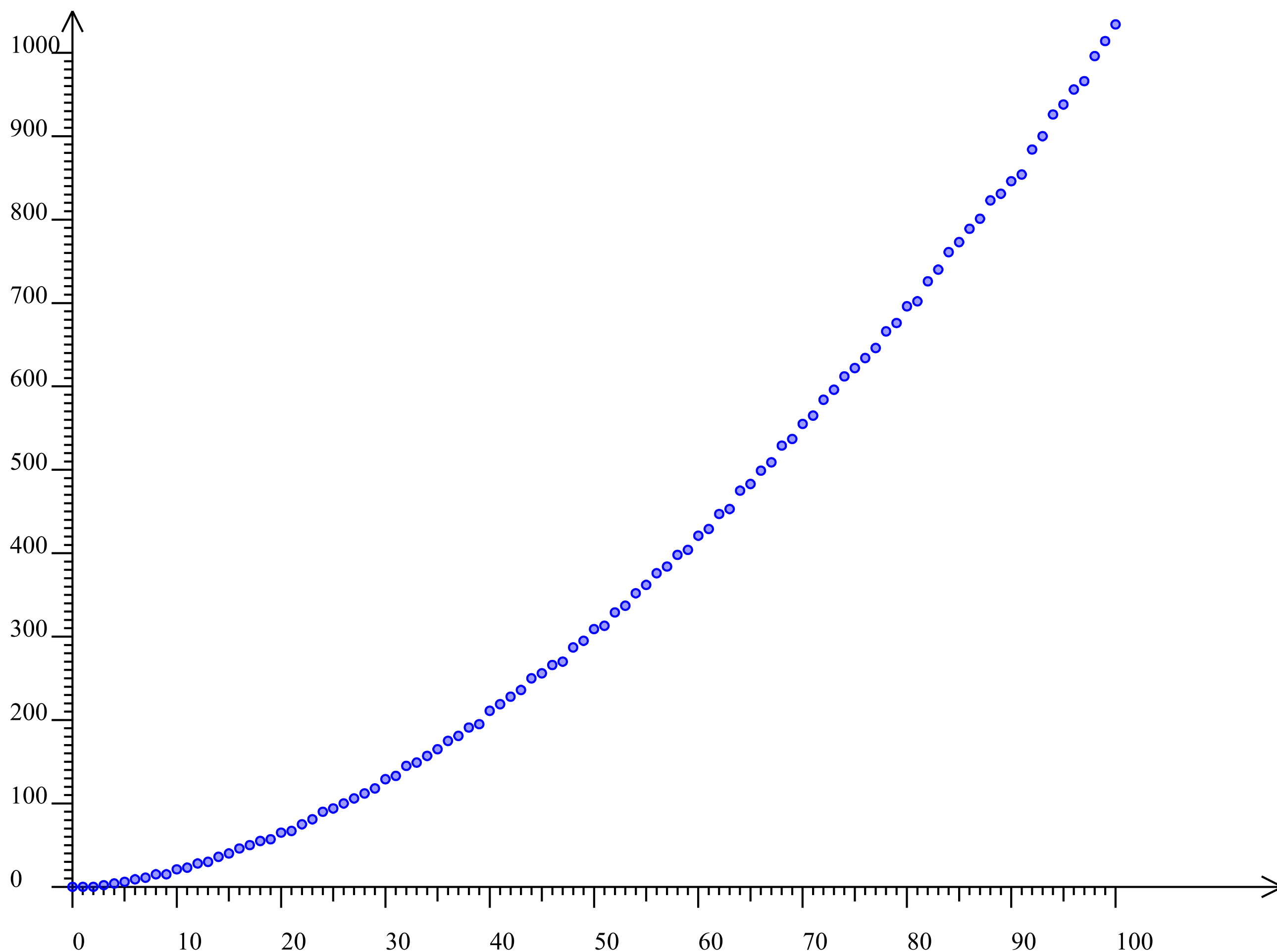
9 En appelant conjugué de m+ωn le nombre m-ωn, la norme de m+ωn est m^2+n^2-mn



Comme il est difficile de compter les entiers d'Eisenstein premiers par norme croissante, on va ensuite approcher les cercles par des hexagones réguliers et compter les nombres premiers d'Eisenstein par abscisse. Et ne compter que le sixième d'entre eux :

```
premiers = [2..10000]
for d in [2..100]
     premiers = (x for x in premiers when x<=d or x%d isnt 0)

isPrime = (a) ->
    if a[1] is 0
        a[0] in premiers and a[0]%3 is 2
    else
        a[0]*a[0]-a[0]*a[1]+a[1]*a[1] in premiers
suite = [0]

for m in [0..100]
    total = 0
    for n in [0...m]
        total++ if isPrime [m,n]
    suite.push suite[suite.length-1] + total

dessineSuite suite, 100, 0, 1000, 2, 'blue'
```

Comme les premiers de Gauss, les premiers d'Eisenstein semblent suivre une répartition en $\frac{x^2}{\ln(x)}$:



Sauf qu'il y a quand même plus de premiers d'Eisenstein, que de premiers de Gauss.

Remarque : Au début des années 1960, Bryan Birch, lors de la rédaction d'une thèse, a constaté avec Peter Swinnerton-Dyer, sur un ordinateur EDSAC, que même modulo un entier premier, le nombre de points à coordonnées rationnelles sur une courbe elliptique est plutôt faible. Cette remarque de nature statistique a mené à la conjecture de Birch et Swinnerton-Dyer, qui est l'un des « problèmes à un million de dollars » de la fondation Clay…

Alain Busser
LPO Roland-Garros
Le Tampon