KLAFYVEL

Zeste de savoir

Approximer rapidement le carré d'un nombre flottant

7 mars 2022

Table des matières

Introduction

Je cherchais récemment un moyen de calculer rapidement des carrés de nombres flottants sur mon Arduino. J'avais en tête quelque chose dans le style du fameux calcul de l'inverse de la racine carrée , mais après quelques minutes de recherche je n'ai pas réussi à mettre la main dessus. Du coup j'ai passé une vingtaine de minutes à bidouiller une solution.

1. Représentation des nombres flottants et calcul de multiplications

Le type `float` sur mon Arduino suit le standard IEEE754 , ce qui signifie qu'il est représenté sous la forme d'une mantisse, un exposant biaisé et un bit de signe. Le layout en mémoire est le suivant:

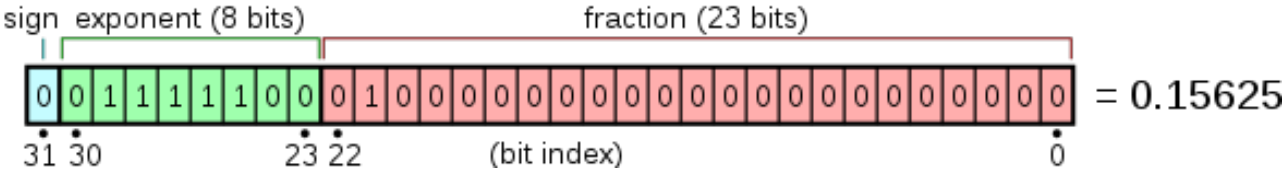


FIGURE 1.1. – Layout mémoire d'un flottant—Vectorization : Stannered, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/ , via Wikimedia Commons

Plus mathématiquement, le nombre flottant n est représenté par $\text{signe}(n) \times \text{mantisse} \times 2^{\text{exposant}-127}$. Donc si on veut multiplier deux nombres m et n, on peut en gros procéder comme suit:

— Ajouter les exposants ;
— Multiplier les mantisses ;
— Ajuster l'exposant final ;
— Mettre le bon signe.

En partant de cette idée on peut chercher une approximation du calcul du carré d'un nombre.

2. Calcul du carré, version 1

Calculer le carré d'un nombre n, c'est calculer $n \times n$. Puisque je cherche à produire une opération optimisée pour la vitesse, on peut se dire que l'on va uniquement traiter la partie qui consiste à additionner les exposants. En effet, c'est ce qui va jouer sur l'ordre de grandeur du résultat. Il suffit donc d'ajouter les deux exposants biaisés puis de soustraire le biais à l'exposant qui en résulte. Le problème de procéder ainsi est qu'il faut isoler l'exposant, le doubler puis le remettre à sa place. J'ai décidé de procéder avec moins de subtilité:

— faire un décalage à gauche de toute la représentation binaire (ce qui revient à peu près à doubler la partie exposant) ;
— Soustraire le biais ;
— Forcer le bit de signe à 0 (car un carré est positif).

Ce qui donne un code ressemblant à cela:

```
1  uint32_t k;
2  float x = 8.0;
3
4  /* Pour travailler sur la représentation binaire de x on force le
       compilateur
5     à l'interpréter comme un entier non signé
6     */
7  k = *(uint32_t *) & x;
8  /* On double (à peu près) l'exposant */
9  k <<= 1;
10 /* On soustrait le biais */
11 k -= 0x3f800000;
12 /* On force le bit de signe à zéro */
13 k &= 0x7fffffff;
```

Ce qui donne ce genre de résultat:

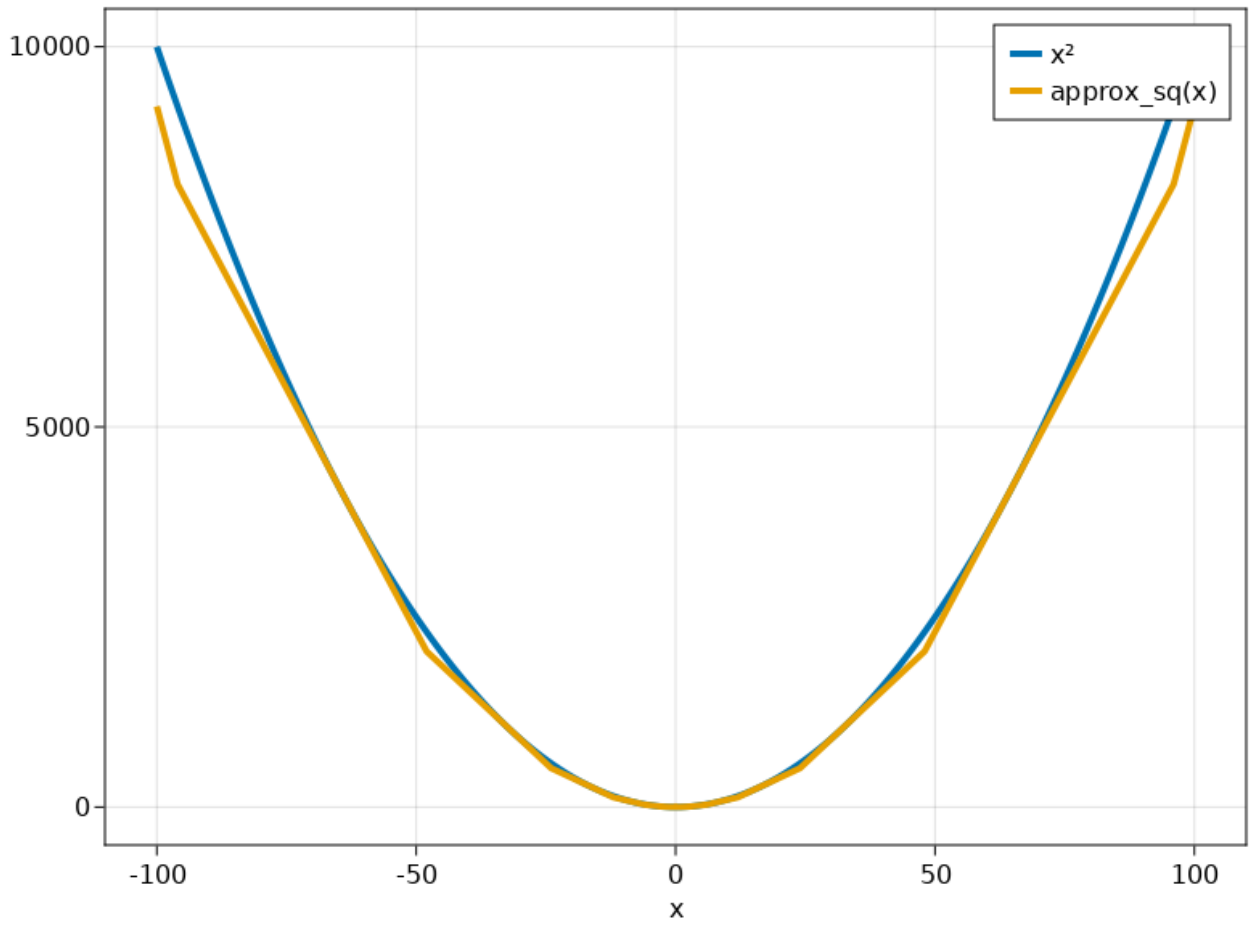


FIGURE 2.2. – Première approximation du carré

C'est assez correct, mais on remarque que l'approximation est systématiquement inférieure au vrai résultat. On peut faire mieux!

3. Calcul du carré, version 2 : la constante magique

Une technique courante pour ce genre de calcul est d'ajouter une constante "magique" sur la mantisse pour être au plus près du vrai résultat. Une personne subtile pourrait utiliser une méthode d'optimisation avancée pour la trouver. Mais si vous avez lu jusqu'ici, vous avez compris que pour moi la délicatesse n'a jamais été une option. Il n'y a que 8388608 valeurs possibles pour la mantisse, alors autant les essayer toutes et prendre la meilleure. 🍊

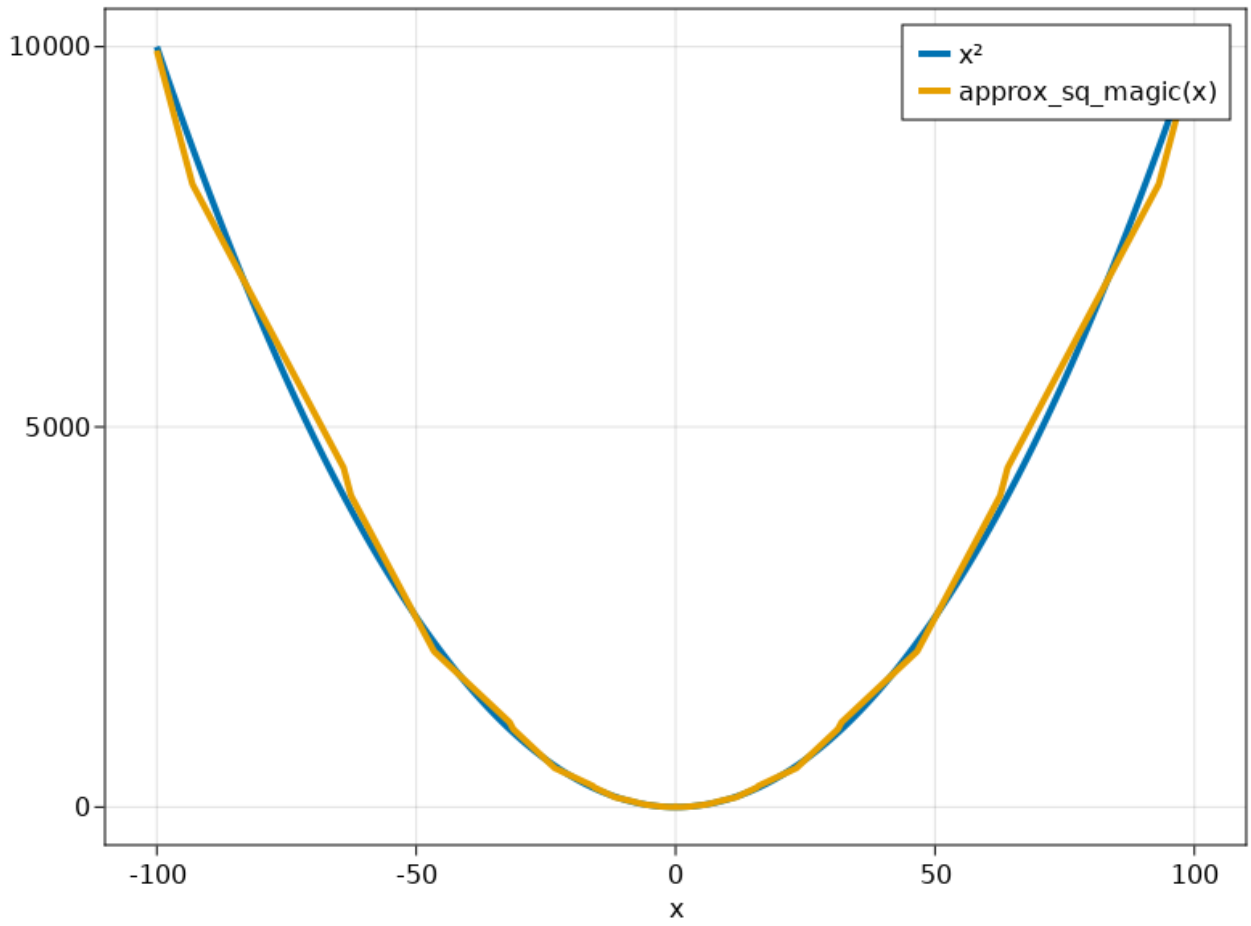
En commettant ces quelques lignes de Julia:

```
using CairoMakie

x = Float32.(-100:0.1:100)
true_value = x.^2

```

```
approx_sq(x::Float32, magic::UInt32) = begin
  i = reinterpret(UInt32, x)
  i <<= 1
  i -= UInt32(127<<23) - magic
  i &= UInt32((1<<31) - 1)
  reinterpret(Float32, i)
end

lines(x, true_value, label="x²", linewidth=4, axis=(xlabel="x",))
lines!(x, approx_sq.(x, zero(UInt32)), label="approx_sq(x)",
    linewidth=4)
axislegend(current_axis())
save("x_sq.png", current_figure())

function to_minimize(magic)
  approx_value = approx_sq.(x, magic)
  sum(abs2.(approx_value .- true_value))
end

possible_values = 0x00000000:0x007fffff
differences = zeros(length(possible_values))
Threads.@threads for (i,magic)  collect(enumerate(possible_values))
  differences[i] = to_minimize(magic)
end
i,i_mini = findmin(differences)

differences = nothing


magic = possible_values[i_mini]
lines(x, true_value, label="x²", axis=(xlabel="x",), linewidth=4)
lines!(x, approx_sq.(x, magic), label="approx_sq_magic(x)",
    linewidth=4)
axislegend(current_axis())
save("x_sq_magic.png", current_figure())
```

On peut trouver la constante magique `0x000b7798` et produire les deux figures de ce billet. Cela donne le résultat présenté à la figure suivante:

3. Calcul du carré, version 2: la constante magique



Le code C++ pour le calcul est ainsi:

```
1  uint32_t k;
2  float x = 8.0;
3
4  /* Pour travailler sur la représentation binaire de x on force le
       compilateur
5     à l'interpréter comme un entier non signé
6     */
7  k = *(uint32_t *) & x;
8  /* On double (à peu près) l'exposant */
9  k <<= 1;
10 /* On soustrait le biais et on ajoute la constante magique*/
11 k -= 0x3f748868;
12 /* On force le bit de signe à zéro */
13 k &= 0x7fffffff;
```

4. Est-ce que c'est plus rapide ?

Oui.

```
float x;
uint32_t k;
int i;

unsigned long timestart;
unsigned long timestop;
unsigned long totaltime;

void setup() {
  Serial.begin(115200);
  while(!Serial) {;}
  Serial.println("Float multiply");
  x = 2;
  timestart=micros();
  for(i=0; i<1000; i++) {
    x = x*x;
  }
  timestop=micros();
  totaltime = (timestop - timestart);
  Serial.print(totaltime, DEC);
  Serial.println(" /1000 µs");
  Serial.println("Forbidden magic");
  x = 2;
  timestart=micros();
  for(i=0; i<1000; i++) {
    k = *(uint32_t *) & x;
    k <<= 1;
    k -= 0x3f748868;
    k &= 0x7fffffff;
    x = *(float*)& k;
  }
  timestop=micros();
  totaltime = (timestop - timestart);
  Serial.print(totaltime, DEC);
  Serial.println(" /1000 µs");

}

void loop() {

}
```

```
Float multiply
7 µs
Forbidden magic
2 µs
```

Conclusion

Voilà, un billet rédigé rapidement avant que j'oublie. C'est surtout pour que je puisse venir recopier ma solution facilement dans le futur. 🍊