Energy efficient calculation of simple functions

Abdulhamid Han

***Abstract*—Energy efficiency depends also from the algorithm. For example for sorting you can use bubble sort or quick sort but the complexity of the two algorithm varies enormously. The complexity of the bubble sort is $O(n^2)$ and the complexity of the quick sort is $O(n \cdot \log(n))$. For $n = 10^6$ the relative deviation is about 10^5. It means one need for the same result more calculation steps and so more power.**

***Index Terms*—Fast inverse square root, Median without sorting, Bit counting**

I. Introduction

TO accelerate the calculation time one needs efficient algorithms which do the same work in a shorter time with an acceptable accuracy. In this paper I will introduce energy efficient algorithms to get the inverse square root, median and the bit counting faster.

March 20, 2016

II. Fast inverse square root [1]

In video games the inverse square root is necessary for lighting and reflection calculations. The inverse square root in this case is used to normalization of vectors. Often the speed is more importantly than the accuracy. So that one is satisfied with an accuracy of 1%. The main idea of this algorithm is to calculate approximately the inverse square root in one calculation step.

Consider a floating point number y like in IEEE 754 Single precision format which is introduced in Appendix B and call it i. One can calculate the inverse square root approximately with equation 1.

$$i = 0x5f3759df - (i >> 1) \tag{1}$$

Note that equation 1 should be calculated bitwise. Let's call 0x5f3759df the magic number. In figure 1 one can see the relative error in % (blue line). One can see also that the maximum relative error is below 4 %. To get a more accurate result one can use Newton's Method. If the initial guess is very close to the zero so the newton step will provide a more accurate solution. Together with the approximately value one can get a better result. To get a more accurate solution one can do more newton steps.

A. Newton's method

An appropriate formula to calculate the inverse square root is

$$f(y) = \frac{1}{y^2} - x \tag{2}$$

By inserting this formula in Newton's method (equation 3) one gets equation 4.

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} n \geq 0 \tag{3}$$

$$y_{n+1} = \frac{1}{2} y_n (3 - x y_n^2) \tag{4}$$

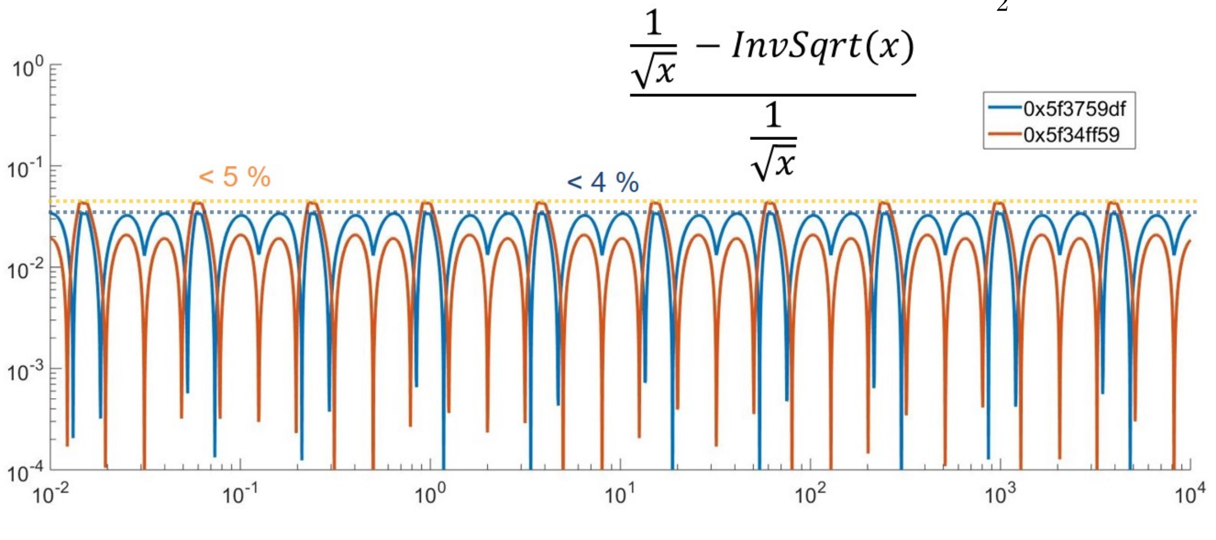


Fig. 1. Relative error of inverse square root as a function of the argument x for two different magic numbers 0x5f3759df and 0x5f34ff59 calculated with Code 1 without newton's method

With Code 1 one can calculate the inverse square root faster with a maximum relative error $< 1\%$ by using of only shift, addition, subtraction and multiplication operators.

Code 1. C Code to calculate the inverse square

```
 1 float InvSqrt( float number ) {
 2     long i;
 3     float x2, y;
 4     const float threehalfs = 1.5F;
 5          x2   = number * 0.5F;
 6          y    = number;
 7          i    = * ( long * ) &y;
 8         // store floating-point bits in long
 9          i    = 0x5f3759df - ( i >> 1 );
10          // initial guess for Newton's method
11          y    = * ( float * ) &i;
12          // convert new bits into float
13          y    = y * ( threehalfs - ( x2 * y * y ) );
14          // 1st iteration
15     return y;
16 }
```

With the command in line 8 the number is stored as IEEE 754 single precision format . In the next line one calculates a initial value with the magic number $0x5f3759df$. In the next step one converts it back in a floating number. In line 14 one calculates one newton step (see also equation 4).

B. 0x5f3759df

A floating number in IEEE 754 format can be written as $x = (1+m)\cdot 2^e$. Where m is the Mantissa and $e = Exponent - 127$. The corresponding integer interpretation is M+LE. For 32 bit floats L is 10^{23} and B is 127. By the following derivation the sign will be omitted.

$$y = x^{-\frac{1}{2}} \tag{5}$$

$$\log_2 y = -\frac{1}{2}\log_2 x \tag{6}$$

$$\log_2((1+m_y)2^{e_y}) = -\frac{1}{2}\log_2((1+m_x)2^{e_x}) \tag{7}$$

$$\log_2(1+m_y) + e_y = -\frac{1}{2}[\log_2(1+m_x) + e_x] \tag{8}$$

$$\log_2(1+m) \approx m + \sigma, m \in [0,1), \sigma \ small \tag{9}$$

$$\rightarrow m_y + \sigma + e_y = -\frac{1}{2}(m_x + \sigma + e_x) \tag{10}$$

Integer view:

$$\frac{M_y}{L} + \sigma + E_y - B = -\frac{1}{2}(\frac{M_x}{L} + \sigma + E_x - B) \tag{11}$$

$$\frac{M_y}{L} + E_y = -\frac{1}{2}(\frac{M_x}{L} + \sigma + E_x - B) + B - \sigma \tag{12}$$

$$\frac{M_y}{L} + E_y = -\frac{1}{2}(\frac{M_x}{L} + E_x) + \frac{3}{2}B - \frac{3}{2}\sigma \tag{13}$$

$$\rightarrow M_y + E_y L = \frac{3}{2}L(B-\sigma) - \frac{1}{2}(M_x + LE_x) \tag{14}$$

Integer representation:

$$\rightarrow I_y = \frac{3}{2}L(B-\sigma) - \frac{1}{2}I_x \tag{15}$$

It follows:

$$i = K - (i >> 1) \tag{16}$$

with a constant K and $-\frac{1}{2}I_x$ can be calculated with $-(i >> 1)$. Equation 16 is identical to line 10 in C Code.Now one have to choose a good value for σ which is introduce in equation 9. In the original implementation σ is 0.0450465 and so one get:

$$\frac{3}{2}L(B-\sigma) = \frac{3}{2}2^{23}(127 - 0.0450465) \tag{17}$$

$$= 1597463007 \tag{18}$$

$$= 0x5f3759df \tag{19}$$

C. Own magic number

In 9 there are an approximation with a small number σ. The mantissa is $\in [0,1)$ and with the assumption that all possible values for m equally distributed so one can say that with equation 20 the statistical error is 0.

$$\int_0^1 [\log_2(1+x) - x - \sigma]dx = 0 \tag{20}$$

$$\rightarrow \sigma = \int_0^1 [\log_2(1+x) - x]dx \tag{21}$$

$$\sigma \approx 0.057305 \tag{22}$$

So my own magic number is:

$$\frac{3}{2}L(B-\sigma) = \frac{3}{2}2^{23}(127 - 0.0.057305) \tag{23}$$

$$= 1597308761 \tag{24}$$

$$= 0x5f34ff59 \tag{25}$$

In figure 1 one can see the relative errors with this magic numbers. One can see that the maximum relative error of the original implementation is slightly better.

D. Magic number for another exponents

Consider the function $f(x) = x^p$
For p = 0.5 (square root) you can use

$$i = 0x1fbd1df5 + (i >> 1) \tag{26}$$

There is also a general formula for $-1 \leq p \leq 1$

$$i = (1-p)\cdot 0x3f7a3bea + (p \cdot i) \tag{27}$$

In figure 2 one can see the relative error in $\%$ calculated with equation 27 with $p = \frac{1}{3}$. One can see also that the maximum relative error is below 3 %.

III. Finding the Median without Sorting [2]

One can determine the median for example with sorting. As the best sorting algorithm has a average complexity of $O(n\cdot\log(n))$, determine the median with sorting has a complexity of $O(n\cdot\log(n))$. In the following an algorithm to determine the median with a linear complexity is introduced.

A. A simple algorithm to find the median

A recursive algorithm to find the median is explained in Algorithm 1. The main idea is to pick an arbitrary pivot element to partition the input array in 3 sections which contain elements less, equal and greater than the pivot element. This recursive algorithm is repeated until the median is choosen randomly as pivot element or the returned section contains only one element.

Algorithm 1: A simple algorithm to find the median

```
Input  : Array $a_0, a_1, \dots, a_{n-1}$ with length n
Output: median $a_m$
1. If $n = 1$
    return $a_0$
else
2.    Choose an arbitrary element x
3.    Partition the array in three sections
          $a_0$, ... , $a_{q-1}$ with elements less than x
          $a_q$, ... , $a_{g-1}$ with elements equal x
          $a_g$, ... , $a_{n-1}$ with elements greater than x
4.    If $k < q$
          return $a_m$ in first section
      If $k < g$
          return x
      else
          return $a_m$ in third section
```

The overall complexity depends from the pivot element. In the best case the partitioned sections are of equal length and the overall complexity is:

$$n + \frac{n}{3} + \frac{n}{9} + \ldots = O(n)$$

In the worst case the returned section is smaller by 1 for example by choosing the minimum or the maximum as pivotelement and so the overall complexity is:

$$n + (n-1) + (n-2) + \ldots = O(n^2)$$

Therefore the pivot element should choosen carefully.

B. Improved version

The choice of the pivot element in the improved version (Algorithm 2) is done more carefully.

Algorithm 2: An improved algorithm to find the median

```
Input  : Array $a_0, a_1, \dots, a_{n-1}$ with length n
Output: median $a_m$
1. If $n < 15$
    sort the array and return $a_k$
else
2.    Partition the array in $\frac{n}{5}$ sections with 5 elements
   and calculate their median
3.    Calculate recursively the median m' of this medians
4.    Partition the array in three sections
          $a_0$, ... , $a_{q-1}$ with elements less than m'
          $a_q$, ... , $a_{g-1}$ with elements equal m'
          $a_g$, ... , $a_{n-1}$ with elements greater than m'
5.    If $k < q$
          return $a_m$ in first section
      If $k < g$
          return m'
      else
          return $a_m$ in third section
```

If $n < 15$ this extra effort is not worth it and so the median can be calculated by sorting. Otherwise a good pivot element is found by partitioning the array in $\frac{n}{5}$ sections and calculate their median for example by sorting. In the next step one calculates the median of this medians. This value is used as the pivot element. This is illustrated in the figure 3.

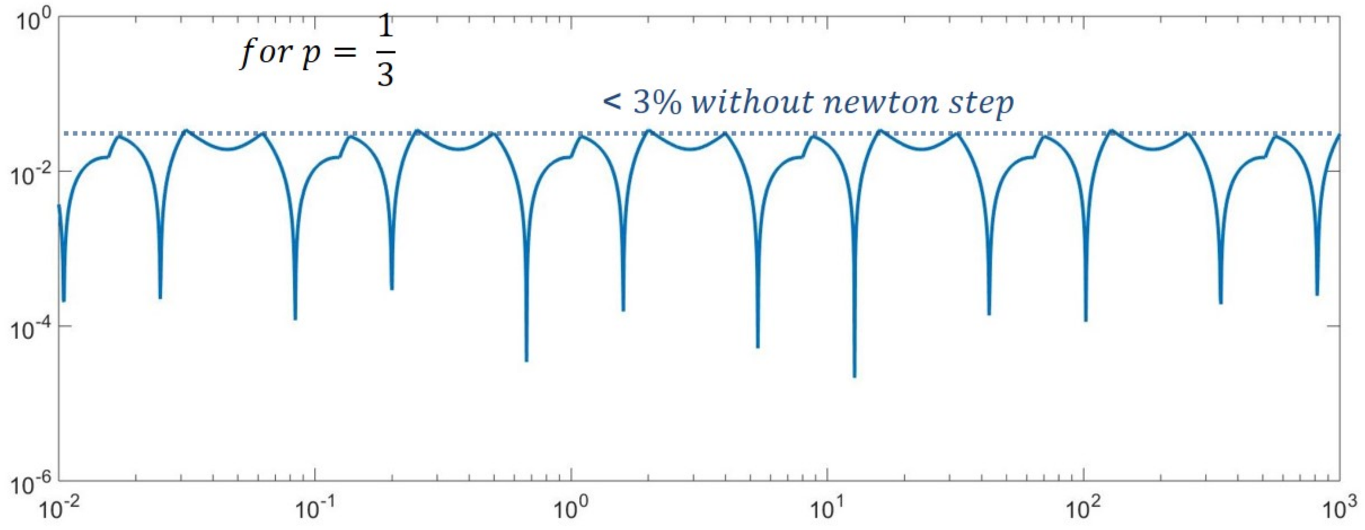


Fig. 2. Relative error of inverse cubic root which is calculated with equation 27 for $p = \frac{1}{3}$ without newton's method

The green circle in the middle represent the median of medians m'. The numbers in the green box are less than m' and the numbers in the red box are greater than m'. Therefore at least 25% of the array is less or greater than m'. One can also say that maximum 75% of the array is less or greater than m'. In consequence the returned array was at most 75%size of the remaining array and so overall complexity is linear.

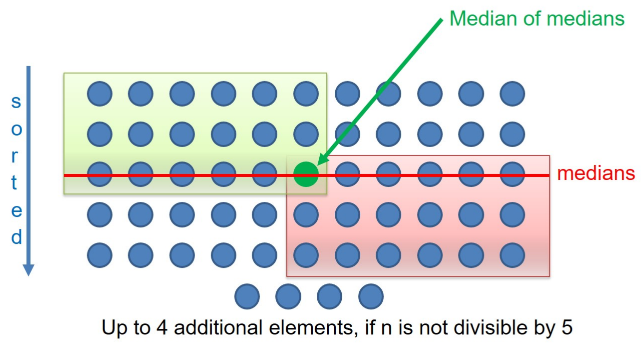


Fig. 3. $\frac{n}{5}$ sections with 5 elements with up to 4 additional elements, if n is not divisible by 5. In the third line the medians of all sections are stored. The green circle in the middle represent the median of medians m'. The numbers in the green box are less than m' and the numbers in the red box are greater than m'

The complexity of first step is constant. Step 2 to 4 can also calculated with linear complexity. As explained above the worst case complexity of last step is

$$n + \frac{3}{4}n + \frac{3^2}{4^2}n + \ldots = O(n)$$

Therefore the overall complexity of the improved version is linear.

In the next two sections implementations will be introduced.

C. Sorting

In the 3rd line of Algortihm 2 one has to determine the median of $\frac{n}{5}$ sections with 5 elements with up to 4 additional elements. This can be done for example with insertion sort (Code 2) which is well suited for small amounts of data.

Code 2. Insertion sort to determine the median

```
void sort5(int lo, int n)
{
    int i, j, h, t;
    h=n/5;
    for (i=lo+h; i<lo+n; i++)
    {
        j=i;
        t=a[j];
        while (j>=lo+h && a[j-h]>t)
        {
            a[j]=a[j-h];
            j=j-h;
        }
        a[j]=t;
    }
}
```

D. Partitioning

In the partition algorithm one distinguishes 4 cases. As in figure 4 first 3 cases are sections which is less(orange) , equal(green) and greater(blue) than x. The last one contains untreated numbers (white).

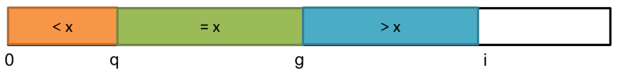


Fig. 4. In the partition algorithm one distinguishes 4 cases. The first 3 cases are sections which is less(orange) , equal(green) and greater(blue) than x. The last one contains untreated numbers (white)

In each iteration one considers an element from the untreated section. If this element is greater than x so one have to increment index i (see also figure 5).

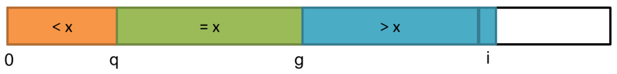


Fig. 5. If the new element is greater than x so one have to increment index i

In case that the new element is equal to x so one has to exchange the indices g and i and increment g (see also figure 6).

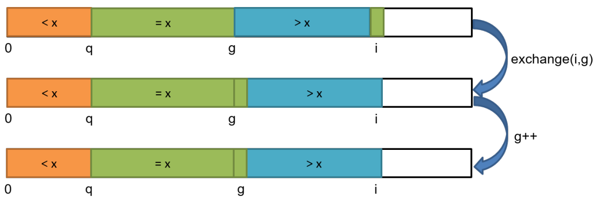


Fig. 6. If the new element is equal x so one have to first exchange the indices g and i and increment g

If the new element is smaller than x so one have to exchange the indices g and i and increment g (see also figure 7). In the next step one have to exchange the indices (g-1) and i and increment q.

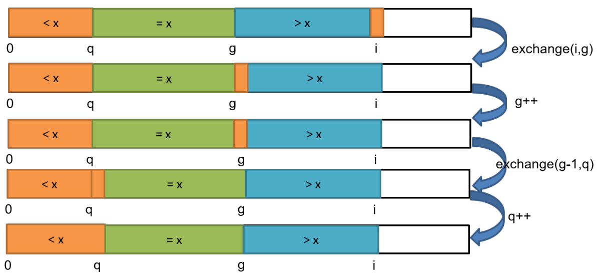


Fig. 7. If the new element is smaller than x so one have to exchange the indices g and i and increment g. In the next step one have to exchange the indices (g-1) and i and increment q

In Code 3 one can see the summary of the above explained steps.

This algorithm returns the indices q and g which seperate sections which less, equal and greater than x. Further n represents the length of the array and lo is the start index.

Code 3. Partition algorithm

```
Pair partition (int lo, int n, int x)
{
    int q=lo, g=lo, i, y;
    for (i=lo; i<lo+n; i++)
    {
        y=a[i];
        if (y<=x)
        {
            exchange(i, g++);
            if (y<x)
                exchange(g-1, q++);
        }
    }
    return new Pair(q, g);}
```

IV. Bit counting [3]

In the following several algorithms to bit counting are introduced.

A. Simple solution

A simple algorithm in Code 4 checks in each step the last significant bit and increments a counter if the bit at this position is 1. By shifting v in line 4 one can compare all bits in 32 steps.

Code 4. Simple solution

```
1 unsigned int c = 0;
  // counter
2 for (unsigned int ibit = 0; ibit < 32; ibit++) {
  // 32 loops !
3   c += v & 1;
  // check LSB and increase c
  if v & 1 is true
4   v >>= 1;   }
  // shift v
```

B. First improvement

In the first improvement one introduces a mask with an initial value of $0x1$. In each step one pushes the mask to the left and increases the counter if at this position is a 1. By introducing the mask one saves one instruction per loop.

Code 5. First improvement

```
unsigned int c = 0;
 for (unsigned int mask = 0x1; mask; mask<<=1) {
// 32 loops! Repeat until mask == 0
   if (v & mask) c++; }
```

Both algorithms always need 32 loops independent of the input vector always 32 loops.

C. Second improvement

In the second improvement one pushes the input vector v to the right until v$\neq$ 0 and increments the counter if there is a 1 in the most significant bit. The algorithm stops when the highest set bit has been found.

Code 6. Second improvement

```
unsigned int c;
for (c = 0; v; v >>= 1) {
// shift while v!=0
  c+= v & 1; }
// increase counter
```

The disadvantage of this algorithm is that one needs as many loops as the highest set bit. For example if v=0x1 one needs one loop but for v=0x80000000 one needs 32 loops.

D. Third improvement

In the third improvement one compares v and (v-1).

$$v = \ldots xyz10\ldots0 \tag{28}$$

$$v-1 = \ldots xyz01\ldots1 \tag{29}$$

$$\rightarrow v\&(v-1) = \ldots xyz00\ldots0 \tag{30}$$

In equation 28 one can see the last set bit. By calculating $v\&(v-1)$ this bit gets eliminated. The position of the 1 isn't important.

Code 7. Third improvement

```
unsigned int c;
for (c = 0; v; c++) { // repeat until v == 0
  v &= v - 1; }        // delete lowest set bit
```

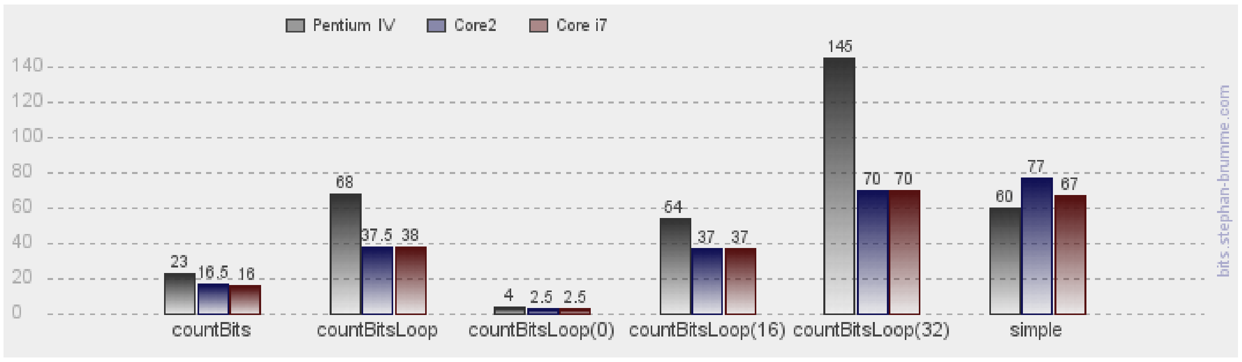


Fig. 8. In this figure one can see CPU cycles for some processors. The first diagram shows the result of the elegant method (countBits). Second diagram shows the average result of the third improvement (countBitsLoop). There are also result for 0, 16 and 32 set bits. The last diagram shows the result for the simple algorithm which is also constant

This algorithm needs as many loops as the number of ones.

E. An elegant method

It's possible to count bits in a constant time. Consider 32 bits as 16 times 2 bits *ab*. Now define *c* as number of ones in *ab*.

a	b	c	ab-0a
0	0	00	00
0	1	01	01
1	0	01	01
1	1	10	10

It's possible to the number of set bits in the pattern *ab* as $ab - 0a$. The corresponding code is $v = v - ((v >> 1)\&0x55555555)$.
Now one sum up two neighboring 2 bits to a 4 bit. There is no carry because the maximum possible number is 4 and there are 4 bits to store them. First one masks the right 2 bits with $v\&0x33333333$. After that one pushes the left one to the right and masks again the right 2 bits. This calculation can be done with $(v\&0x33333333) + ((v >> 2)\&0x33333333)$.
Similarly to the last calculation one sums up 2 neighbor 4 bits to 8 bit with $(v + (v >> 4))\&0x0F0F0F0F$.
Now one has to sum up this 4 times 8 bits which we call A,B,C and D. By calculation of $v \cdot 0x01010101$ one gets $\mathbf{D}000 + \mathbf{C}D00 + \mathbf{B}CD0 + \mathbf{A}BCD$. As you can see the required sum is stored in the first 8 bit and there is still no carry. To get the result one has to calculate $(v \cdot 0x01010101) >> 24$. This result provides the set bits. Code 8 contains the summary of the above introduced codes.

Code 8. An elegant method

```
v = v - ((v >> 1) & 0x55555555);
// count bits in two groups
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
// Add 2 groups-> 4 groups
v = (v + (v >> 4));
// Add 4 groups-> 8 groups
v &= 0x0F0F0F0F;          // delete useless bits
c = (v \cdot 0x01010101) >> 24;
// Add the 4 8 groups
```

In figure 8 one can see CPU cycles for some processors. The first diagram shows the result of the elegant method (countBits). As one can see the result is independent of the input vector. Second diagram shows the average result of the third improvement (countBitsLoop). There are also result for 0, 16 and 32 set bits. As you can see the third improvement is the fast one for 0 set bits. But on average the elegant method is the best algorithm. The last diagram shows the result for the simple algorithm which is also constant.

V. Conclusion

In this paper i introduced methods to get the inverse square root, median and the bit counting faster. Inverse sqaure root can be calculated 4 times faster with an accuracy of $< 1\%$. In addition there is a method to get the median with linear complexity. Finally i show methods to do bit counting faster. There is also an elegant which do this calculation in a constant time independent of the input vector.

Appendix A
Newton's method

Newton's method is used to get a zero of a function. This algorithm needs an initial value. This value is used to create a tangent at this point. In the next iteration one calculates the zero of the tangent and at this point one creates again a tangent and so on. This algorithm will end if the zero of the tangent is close enough to the zero of the function. The convergence of this algorithm depends also from the initial value. Newton step can be calculated with the equation 31.

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}, n \geq 0 \tag{31}$$

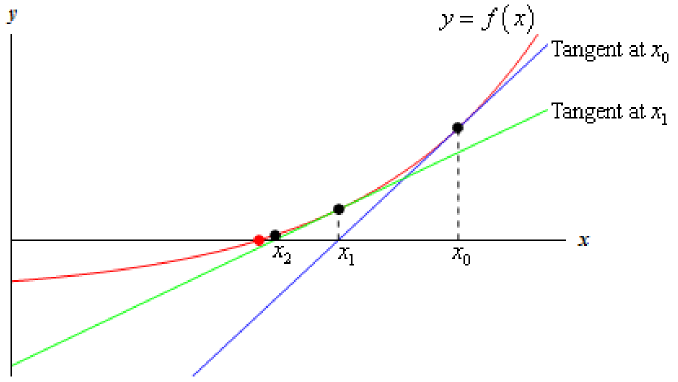
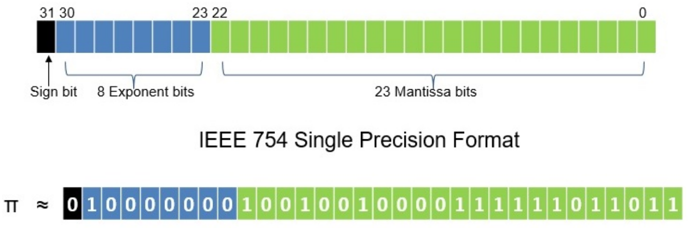


Fig. 9. Newton's Method [4]

Appendix B
IEEE 754 single precision format

Single precision floating numbers are stored as 32 bit numbers



31 30
23 22
0
Sign bit
8 Exponent bits
23 Mantissa bits
IEEE 754 Single Precision Format
π ≈ 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1

Fig. 10. IEEE 754 single precision format

The stored number is

$$x = (-1)^{sign} \cdot (1.Mantissa) \cdot 2^{Exponent-127} \tag{32}$$

References

[1] http://h14s.p5r.org/2012/09/0x5f3759df.html
[2] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/median.htm
[3] http://wiki.kip.uni-heidelberg.de/ti/Informatik-Vorkurs/index.php/TippsAndTricks
[4] http://tutorial.math.lamar.edu/Classes/CalcI/NewtonsMethod_files/image001.gif