A method of calculating faithful rounding of l_2-norm for n-vectors

Stef Graillat

LIP6/PEQUAN, Sorbonne Universités, UPMC Univ Paris 06, CNRS

Joint work with Christoph Lauter, Peter Tang, Naoya Yamanaka and Shin'ichi Oishi

SCAN 2014, 16th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computation

Würzburg, Germany, September 21-26, 2014

Problem

Let $\mathbb{F}$ be a set of floating-point numbers, ε unit roundoff, $\circ$ rounding to nearest

For the sake of simplicity, we can assume that $\mathbb{F}$ is the set of `binary64` floating-point numbers in IEEE 754 standard.

Problem

Let $\mathbb{F}$ be a set of floating-point numbers, ε unit roundoff, $\circ$ rounding to nearest

For the sake of simplicity, we can assume that $\mathbb{F}$ is the set of `binary64` floating-point numbers in IEEE 754 standard.

Aim

We are concerned with the problem of calculating l_2-norm of n-vectors $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t \in \mathbb{F}^n$,

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Motivations

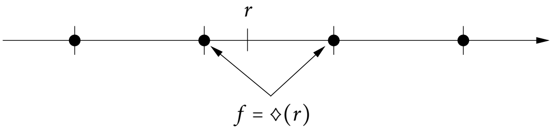
- The computation of l_2-norm is used in
  - the normalization of vectors
  - in Gram-Schmidt process for orthonormalizing vectors
  - in QR decomposition using Householder reflections
  - in some algorithms to compute eigenvalues (power iteration method)

Motivations

- The computation of l_2-norm is used in
  - the normalization of vectors
  - in Gram-Schmidt process for orthonormalizing vectors
  - in QR decomposition using Householder reflections
  - in some algorithms to compute eigenvalues (power iteration method)
- With some guaranteed accuracy,
  - we increase the accuracy
  - we simplify error analysis
  - we make a step toward more accurate algorithms
  - we improve the chance to get reproducible results when computations are done in parallel

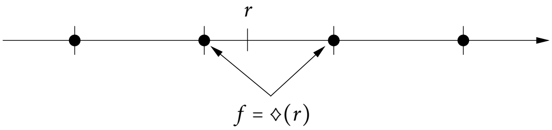
Purpose (1/2)

Our aim is to get a faithful rounding of $\|\mathbf{x}\|_2$ at a reasonable cost.



Purpose (1/2)

Our aim is to get a faithful rounding of $\|\mathbf{x}\|_2$ at a reasonable cost.



- To get a floating-point number faithful to $\|\mathbf{x}\|_2$, calculating $\sum x_i^2$ up to nearest and taking the square root is enough.
- However, calculating $\sum x_i^2$ up to nearest sometimes requires a lot of computations.
- Calculating $\sum x_i^2$ up to faithful is not enough to get a faithful rounding of $\|\mathbf{x}\|_2$.

Purpose (2/2)

Thus, our purpose is to seek an efficient algorithm to calculate a floating-point number faithful to $\|\mathbf{x}\|_2$ for $\mathbf{x} \in \mathbb{F}^n$.

- First calculate $S \approx \sum x_i^2 := \sigma$ with a little bit rough accuracy compared with nearest but more accurate compared with faithful.
- Then, calculate $\sqrt{S}$ using the square root of IEEE754.

Purpose (2/2)

Thus, our purpose is to seek an efficient algorithm to calculate a floating-point number faithful to $\|\mathbf{x}\|_2$ for $\mathbf{x} \in \mathbb{F}^n$.

- First calculate $S \approx \sum x_i^2 := \sigma$ with a little bit rough accuracy compared with nearest but more accurate compared with faithful.
- Then, calculate $\sqrt{S}$ using the square root of IEEE754.

Problem

To which accuracy, we need to calculate $S \approx \sum x_i^2$ so as to a floating-point number $\circ(\sqrt{S})$ becomes faithful to $\|\mathbf{x}\|_2$.

→ We also want to deal with underflow and overflow

Existing solutions

- Common implementations such as the public version of LAPACK released by `netlib` essentially compute the l_2-norm as

$$\widehat{x} \times \|\mathbf{x}/\widehat{x}\|_2$$

  where $\widehat{x} = \max_j |x_j|$.

- That implementation requires n divisions in total, which is significantly more expensive than the naïve formula would suggest.
- In the worst-case scenario, the last $\log_{10}(n)$ digits of the result could be corrupted.
- Avoid overflow but not underflow

Main theorem

Problem

To what accuracy, we need to calculate $S \approx \sum x_i^2$ so as to a floating-point number $\circ(\sqrt{S})$ becomes faithful to $\|\mathbf{x}\|_2$.

Let $\sigma = \sum x_i^2$ then $\|\mathbf{x}\|_2 = \sqrt{\sigma}$

Main theorem

Problem

To what accuracy, we need to calculate $S \approx \sum x_i^2$ so as to a floating-point number $\circ(\sqrt{S})$ becomes faithful to $\|\mathbf{x}\|_2$.

Let $\sigma = \sum x_i^2$ then $\|\mathbf{x}\|_2 = \sqrt{\sigma}$

Theorem

Let σ be a real number and $S, s \in \mathbb{F}$ where $\circ(S + s) = S$. If $|(S + s) - \sigma| < \varepsilon\sigma/8$, then $\circ(\sqrt{S}) \in \Diamond(\sqrt{\sigma})$.

Using double-FP

```
function SumNonNeg(A, B) // [A, a] + [B, b]
// A = [A, a], B = [B, b] nonnegative: A + a, B + b ≥ 0
    H ← TwoSum(A, B)      // H = [H, h], H + h = A + B exactly
    c ← a ⊕ b       // c = a + b + δ_c
    d ← h ⊕ c       // d = h + c + δ_d.
    S ← FastTwoSum(H, d) // S = [S, s], S + s = H + d exactly
    return S
end SumNonNeg
```

Using double-FP

```
function SumNonNeg(A, B) // [A, a] + [B, b]
// A = [A, a], B = [B, b] nonnegative: A + a, B + b ≥ 0
    H ← TwoSum(A, B)      // H = [H, h], H + h = A + B exactly
    c ← a ⊕ b      // c = a + b + δ_c
    d ← h ⊕ c      // d = h + c + δ_d.
    S ← FastTwoSum(H, d) // S = [S, s], S + s = H + d exactly
    return S
end SumNonNeg
```

Theorem

Let $\mathbf{S} = [S, s]$ *be the result from applying* SumNonNeg *on nonnegatives* $\mathbf{A} = [A, a]$ *and* $\mathbf{B} = [B, b]$. *Let* $\alpha = A + a \geq 0$, $\beta = B + b \geq 0$ *denote the exact input values, and* $\sigma = \alpha + \beta$ *denote the exact sum. Then* $|(S + s) - \sigma| \leq 3\varepsilon^2\sigma$.

Computing Sum Of Square with double-FP

```
function SumOfSquares(x) // Accurate accumulation
    S ← [0, 0]
    for j = 1, 2, ..., n do:
        P ← TwoProd(x_j, x_j)      // P = [P, p], P + p = x_j^2 exactly
        S ← SumNonNeg(S, P)
    return S
end SumOfSquares
```

Computing Sum Of Square with double-FP

```
function SumOfSquares(x) // Accurate accumulation
    S ← [0, 0]
    for j = 1, 2, ..., n do:
        P ← TwoProd(x_j, x_j)      // P = [P, p], P + p = x_j^2 exactly
        S ← SumNonNeg(S, P)
    return S
end SumOfSquares
```

Theorem

Let n be the length of a vector $\mathbf{x}$ in safe range and σ denote $\sum_j x_j^2$. Let SumOfSquares($\mathbf{x}$) *return the result $[S, s]$. Then*

$$|(S + s) - \sigma| \le \Delta_{n-1}(3\varepsilon^2)\sigma, \qquad \textit{where} \qquad \Delta_\ell(\delta) = \ell\delta/(1 - \ell\delta).$$

In particular, if the length n satisfies $n < ((24 + \varepsilon)\varepsilon)^{-1}$, then

$$|(S + s) - \sigma| < \varepsilon\sigma/8.$$

Parallel version (1/2)

- partition the input vector $\mathbf{x}$ to τ subvectors of roughly equal length
- Perform the sum of squares on each subvector in parallel
- The partial sums of squares are then accumulated in a serial manner.

function SumOfSquaresP($\mathbf{x}$) // Parallel SumOfSquares
  Partition $\mathbf{x}$ into τ portions, $\mathbf{x}^{(t)}$, $t = 1, 2, \ldots, \tau$
  // length of each $\mathbf{x}^{(t)}$ is no more than $m = \lceil n/\tau \rceil$.
  $\mathbf{S}^{(t)} \leftarrow \texttt{SumOfSquares}(\mathbf{x}^{(t)}), \quad t = 1, 2, \ldots, \tau.$
  // In parallel, each $\mathbf{S}^{(t)} = [S^{(t)}, s^{(t)}]$ is a double-FP.
  $\mathbf{S} \leftarrow [0, 0]; \quad \mathbf{S} \leftarrow \texttt{SumNonNeg}(\mathbf{S}, \mathbf{S}^{(t)}), \quad t = 1, 2, \ldots, \tau.$
  // In serial, summing the τ partial sums of squares
  // $\mathbf{S} = [S, s]$ at this point; $S + s \approx \sum_j^n x_j^2$.
  return S
end SumOfSquaresP

Parallel version (2/2)

Theorem

Let n be the length of $\mathbf{x}$ and $\mathbf{S} = [S, s]$ be the result of `SumOfSquaresP`$(\mathbf{x})$ *with τ portions and $m = \lceil n/\tau \rceil$. Then*

$$|(S + s) - \sigma| \leq \Delta_{m+\tau}(3\varepsilon^2)\sigma.$$

In particular,

$$|(S + s) - \sigma| \leq \Delta_{n-1}(3\varepsilon^2)\sigma$$

whenever $m + \tau \leq n - 1$.

Dealing with underflow and overflow (1/2)

Problems

- Direct computation of $P + p = x_j^2$ not possible
  - square would overflow for large x_j
  - square would underflow for small x_j
  - square stays on normal range only for medium x_j

Dealing with underflow and overflow (1/2)

Problems

- Direct computation of $P + p = x_j^2$ not possible
  - square would overflow for large x_j
  - square would underflow for small x_j
  - square stays on normal range only for medium x_j

Solution

- Use of the "tree bins" strategy [Blue 1978]
  - scale large x_j down with γ a statically chosen power of 2, accumulate in bin $\mathcal{A}$
  - scale small x_j up with γ^{-1}, accumulate in bin $\mathcal{C}$
  - let medium x_j as-is, accumulate in bin $\mathcal{B}$

Dealing with underflow and overflow (2/2)

Given the input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, the three bins are

$$\begin{array}{lcllcl} \mathcal{A} & = & \{ & \gamma x_j & | & |x_j| \geq \beta_{\text{hi}} & \}, \\ \mathcal{B} & = & \{ & x_j & | & \beta_{\text{lo}} \leq |x_j| < \beta_{\text{hi}} & \}, \\ \mathcal{C} & = & \{ & x_j/\gamma & | & |x_j| < \beta_{\text{lo}} & \}. \end{array}$$

By design $\beta_{\text{lo}} \leq |\widehat{x}_j| < \beta_{\text{hi}}$ for $\widehat{x}_j \in \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$. Denote the partial, scaled, sums-of-squares as

$$\widehat{\sigma}_{\mathcal{A}} = \sum_{\widehat{x}_j \in \mathcal{A}} \widehat{x}_j^2, \quad \widehat{\sigma}_{\mathcal{B}} = \sum_{\widehat{x}_j \in \mathcal{B}} \widehat{x}_j^2, \quad \text{and} \quad \widehat{\sigma}_{\mathcal{C}} = \sum_{\widehat{x}_j \in \mathcal{C}} \widehat{x}_j^2.$$

Furthermore,

$$\sigma = \sum_j x_j^2 = \gamma^{-2}\,\widehat{\sigma}_{\mathcal{A}} + \widehat{\sigma}_{\mathcal{B}} + \gamma^2\,\widehat{\sigma}_{\mathcal{C}}. \tag{1}$$

General case (1/3)

function `SumOfSquaresBins`($\mathbf{x}$) // general inputs

Obtain bins $\mathcal{U}$, $\mathcal{V}$, and integer k as discussed

// $\gamma^k(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}})$ approximates $\sum_j x_j^2$ accurately

// $k = -2$ if $\mathcal{U}$ is $\mathcal{A}$, $k = 0$ if $\mathcal{U}$ is $\mathcal{B}$

// Note that $k = -2$ if and only if bin $\mathcal{A}$ is nonempty

$[U, u] \leftarrow$ `SumOfSquaresP`($\mathbf{x}^{(\mathcal{U})}$);

$[V, v] \leftarrow$ `SumOfSquaresP`($\mathbf{x}^{(\mathcal{V})}$);

if $U = 0$ // $\mathcal{A}$ and $\mathcal{B}$ are both empty

$m \leftarrow 2$, $[S, s] \leftarrow [V, v]$,

return m and $\mathbf{S} = [S, s]$.

if $U \geq \beta_{\text{lo}}^2/\varepsilon^3$ **or** $V \leq \beta_{\text{hi}}^2\varepsilon^2$

$m \leftarrow k$, $[S, s] \leftarrow [U, u]$

return m and $\mathbf{S} = [S, s]$

if $|v| \leq \beta_{\text{hi}}^2\varepsilon^2$, $v \leftarrow 0$.

$[U, u] \leftarrow [\gamma^{-1}U, \gamma^{-1}u]$; $[V, v] \leftarrow [\gamma V, \gamma v]$; $m \leftarrow k + 1$;

$[S, s] \leftarrow$ `SumNonNeg`($[U, u], [V, v]$)

return m and $\mathbf{S} = [S, s]$

end `SumOfSquaresBins`

General case (2/3)

Theorem

Let $\texttt{SumOfSquaresBins}(\mathbf{x})$ *return* m *and* $\mathbf{S} = [S, s]$. *Denote by* $\widehat{\sigma}$ *the scaled sums of squares* $\widehat{\sigma} = \gamma^{-m}\sigma = \gamma^{-m}\sum_j x_j^2$. *If the length* n *of* $\mathbf{x}$ *satisfies* $n + 3 < ((24 + \varepsilon)\varepsilon)^{-1}$, *then* $\circ(\sqrt{S}) \in \diamond(\sqrt{\widehat{\sigma}})$.

```
function AccuNrm2(x) // general faithful l_2-norm
    (m, S) ← SumOfSquaresBins(x)
    // m is an integer in the range [−2, 2] and γ^m (S + s) ≈ ∑_j x_j^2
    // By design, γ^m is an even power of 2.
    Z ← sqrt(S)
    return γ^{m/2} ⊗ Z
end AccuNrm2
```

General case (3/3)

Theorem

Let $\mathbf{x}$ *be a vector of length* n*. If* $n < L'$ *with* $L' = ((24 + 3\varepsilon)\varepsilon)^{-1} - 3$*, then* $\texttt{AccuNrm2}(\mathbf{x}) \in \diamond(\|\mathbf{x}\|_2)$ *and reports overflow and underflow faithfully.*

	Vector length bound $n < L'$
binary32	$L' = 699047$
binary64	$L' = 3.75299968947538 \cdot 10^{14}$

Numerical experiments (1/4)

- Tests on a 4-core Intel Core i7 at 2.67 GHz with 4Gb of RAM and on a 8-core Intel Xeon E3-1275 v3 at 3.50 GHz with 32Gb of RAM
- All implementations were written in C and compiled using `gcc` version 4.8 and options `-std=c99 -O3 -march=native`
- Timings are given cycles per vector element

Numerical experiments (2/4)

Maximum error in ulps observed for various domains and vector lengths n, plain SSE implementation

	vectors with normal results		vectors for which results underflow	
	$n = 10^3$	$n = 10^7$	$n = 10^3$	$n = 10^7$
`NaiveNorm`	∞	∞	$8.84 \cdot 10^{12}$	$5.46 \cdot 10^{10}$
`NetlibNorm`	2.01	524	0.496	0.698
`MPFRNorm`	0.494	0.481	0.490	0.498
`FaithfulNorm`	0.620	0.628	0.497	0.499

	vectors with entries around 1.0		vectors with chosen "half-ulp" entries	
	$n = 10^3$	$n = 10^7$	$n = 10^3$	$n = 10^7$
`NaiveNorm`	7.73	861	250	$2.50 \cdot 10^6$
`NetlibNorm`	7.58	609	250	$2.50 \cdot 10^6$
`MPFRNorm`	0.468	0.497	0.0749	0.484
`FaithfulNorm`	0.605	0.701	0.0749	0.484

Numerical experiments (3/4)

Computation time in cycles per vector element, plain SSE version on Intel Core i7

	vectors with normal results	vectors for which results underflow	vectors with entries around 1.0	vectors for which results overflow	vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	47.0	137.	3.48	46.8	128.
`NetlibNorm`	156.	472.	19.1	156.	274.
`MPFRNorm`	1080	2670	818.	1090	1660
`FaithfulNorm`	34.2	289.	25.3	34.2	62.2

Computation time in cycles per vector element, plain SSE version on Intel Xeon E3-1275

	vectors with normal results	vectors for which results underflow	vectors with entries around 1.0	vectors for which results overflow	vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	4.95	4.75	4.72	4.70	4.52
`NetlibNorm`	21.9	158.	12.8	21.1	21.8
`MPFRNorm`	810.	1160	536.	803.	717.
`FaithfulNorm`	21.5	87.3	21.8	21.7	20.3

Numerical experiments (4/4)

Computation time in cycles per vector element, AVX version w/o FMA on Intel Xeon E3-1275

	vectors with normal results	vectors for which results underflow	vectors with entries around 1.0	vectors for which results overflow	vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	4.85	4.61	4.68	4.86	4.52
`NetlibNorm`	21.1	157.	13.3	21.6	21.8
`MPFRNorm`	795.	1250	552.	765.	720.
`FaithfulNorm`	12.0	50.7	12.5	12.6	14.8

Computation time in cycles per vector element, AVX version using FMA on Intel Xeon E3-1275

	vectors with normal results	vectors for which results underflow	vectors with entries around 1.0	vectors for which results overflow	vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	4.52	4.52	4.52	4.52	4.52
`NetlibNorm`	20.5	151.	12.6	20.5	22.0
`MPFRNorm`	722.	1110	481.	723.	770.
`FaithfulNorm`	6.94	42.3	6.94	6.94	10.4

Conclusion and future work

Conclusion:

- an efficient algorithm to compute a faithful rounding of the l_2-norm of a floating-point vector
- this algorithm does not generate overflows nor underflows spuriously
- this algorithm is well suited for parallel implementation and vectorization
- the implementation runs up to 3 times faster than the `netlib` version on current processors.

Future work:

- finding an efficient algorithm for vectors of small size
- finding an efficient algorithm with rounding to nearest result

Thank you for your attention