Comparison between iRRAM, double, double double(DD), and quad double(QD)

November 2, 2019

1 Accuracy Test

As iRRAM supports arbitrary precision, all values given by iRRAM are assumed true. The other data structures will show prominent errors when calculating $x_{n+1} = 3.75x_n(1 - x_n), x_1 = 0.5$ at some point. For double-double(DD) and quad-double(QD), we take advantage of [1]. Take caution when you initialize a variable. If you put a floating number directly, it will be regarded as a double value first, and then converted to the target data structure. Hence, it is advisable to use division for a quotient number.

Algorithm 1 Part of accuracy code

```
(...)

// initial values
REAL    xr=RATIONAL(1,2), cr=RATIONAL(15,4);
double xd=1.0/2, cd=15.0/4;
dd_real xdd=1, cdd=15; xdd/=2; cdd/=4;
qd_real xqd=1, cqd=15; xqd/=2; cqd/=4;

(...)

// calc
REAL errD, errDD, errQD;
for(long i=1;i<=count;i++ ) {
  // calc errors
  errD = xr - REAL(xd);
  errDD = xr - REAL(xdd.to_string());
  errQD = xr - REAL(xqd.to_string());

  // check if double, DD, and QD has showed error more than epsilon, respectively
  if(!isDbroken) {  if(abs(errD) > eps) {
    isDbroken = true; breakD = REAL(xd); breakDidx = i; }}
  if(!isDDbroken) {  if(abs(errDD) > eps) {
    isDDbroken = true; breakDD = REAL(xdd.to_string()); breakDDidx = i; }}
  if(!isQDbroken) {  if(abs(errQD) > eps) {
    isQDbroken = true; breakQD = REAL(xqd.to_string()); breakQDidx = i; }}

  (...)
}
(...)
```

Running with 200 iterations(input), we had

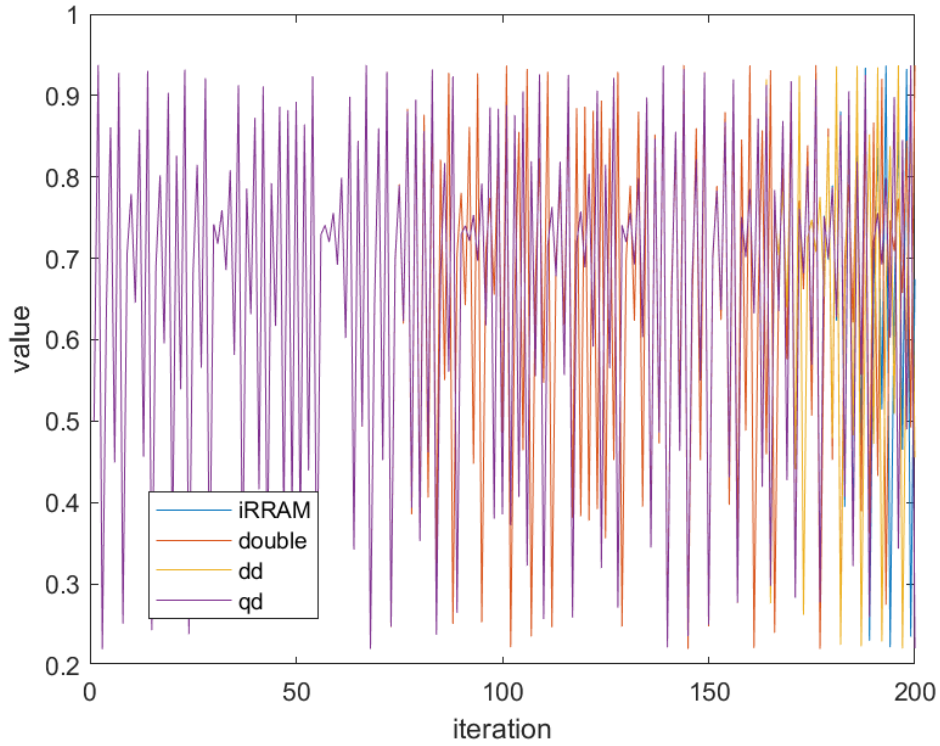

Figure 1: raw values

You probably want to see clear disparity, or error.

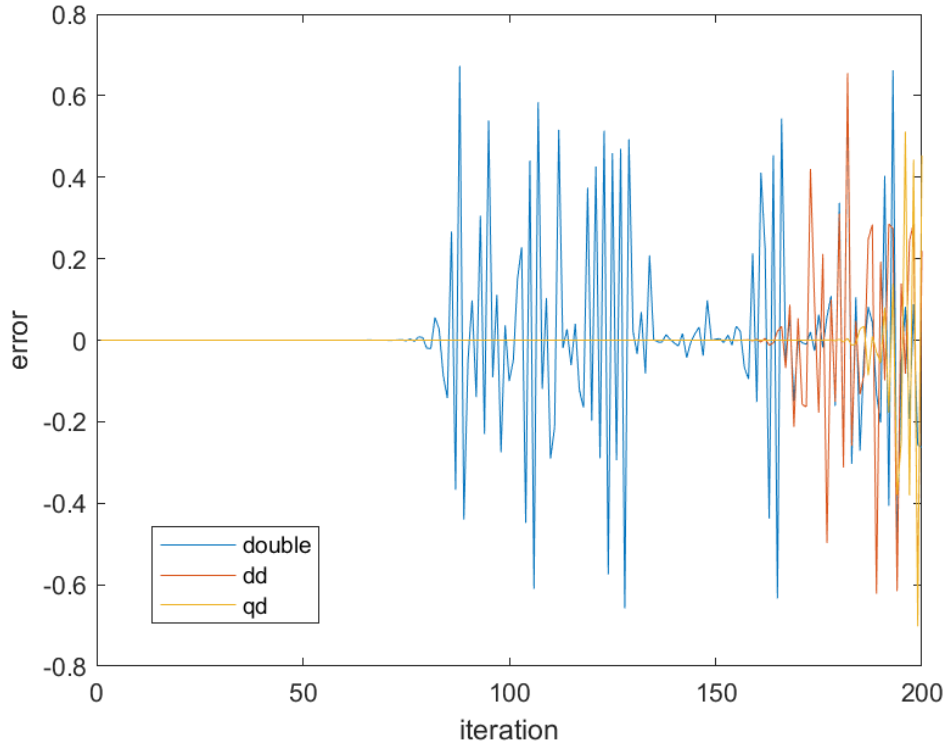

Figure 2: errors

The first iteration # at which err is greater than $\epsilon = 0.000001$ are 55, 138, and 165 for Double, DD, and QD, respectively. Surprisingly, from Double to DD, more than double iterations has been proceeded with relatively

low error. However, from DD to QD, only 27 more steps were extended, even if both transitions make storage size double.

2 Speed

We analyzed for some basic opertions. Each operation is applied repeatedly and we measure the elapsed time. This must not break correctness, thereby, we first check if the value is correct, and then do the test. Also, we repeated each experiment at least 2^{10}times and took the average because outliers came up out of blue. Lastly, when the data seem to make a line, we made a linear regression on them.

Algorithm 2 Part of code for addition performance test

```
1 // iRRAM
2 for(int i=0;i<m;i++) {
3   beginTime = high_resolution_clock::now();      // start timer
4   r = rInit;
5   for(long i=0;i<nPlus;i++) r = sqrt(r);         // apply the operation
6   endTime = high_resolution_clock::now();        // stop timer
7   rElapsed += duration_cast<nanoseconds>(endTime-beginTime).count(); // elapsed time
8 }
9 rElapsed /= m;           // take the average
```

2.1 Addition

Test setup: $x_{n+1} = x_n + c,\ x_0 = 1.0, c = 3.14159,\ |e_n| < 2^{-20}$

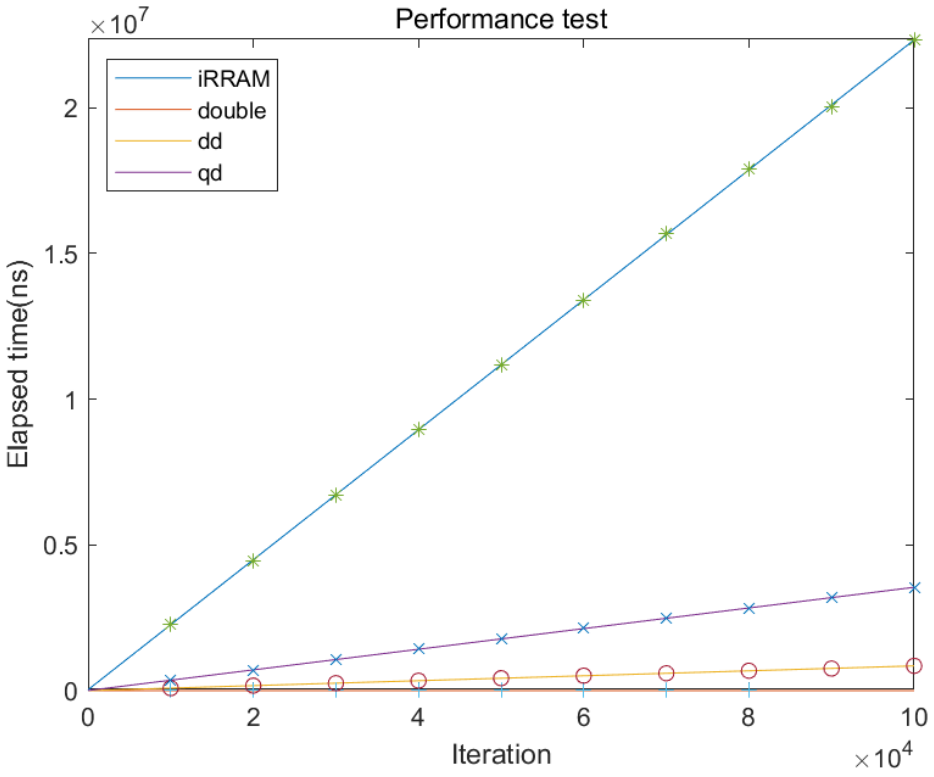

Figure 3: Addition Test

itr	iRRAM	double	DD	QD
10000	2277466	40	85559	352622
20000	4464846	61	171552	706488
30000	6706339	59	255780	1070855
40000	8975040	40	341247	1426889
50000	11187951	51	426594	1774732
60000	13375699	47	513372	2134957
70000	15690342	39	595202	2500221
80000	17920073	39	682645	2807334
90000	20019130	-	758295	3203934
100000	22350155	-	849898	3541002

	iRRAM	double	DD	QD
slope	223	-0.0002	8.5	35.4

Table 1: Elapsed time(ns) and slope(ns/itr)

Double is the fastest followed by DD, QD, and iRRAM. Double showed a constant time complexity, but failed at iteration # 89893 because of lack of precision.

2.2 Multiplication

Test setup: $x_{n+1} = x_n c,\ x_0 = 1.0, c = 3.14159,\ |e_n| < 2^{-20}$

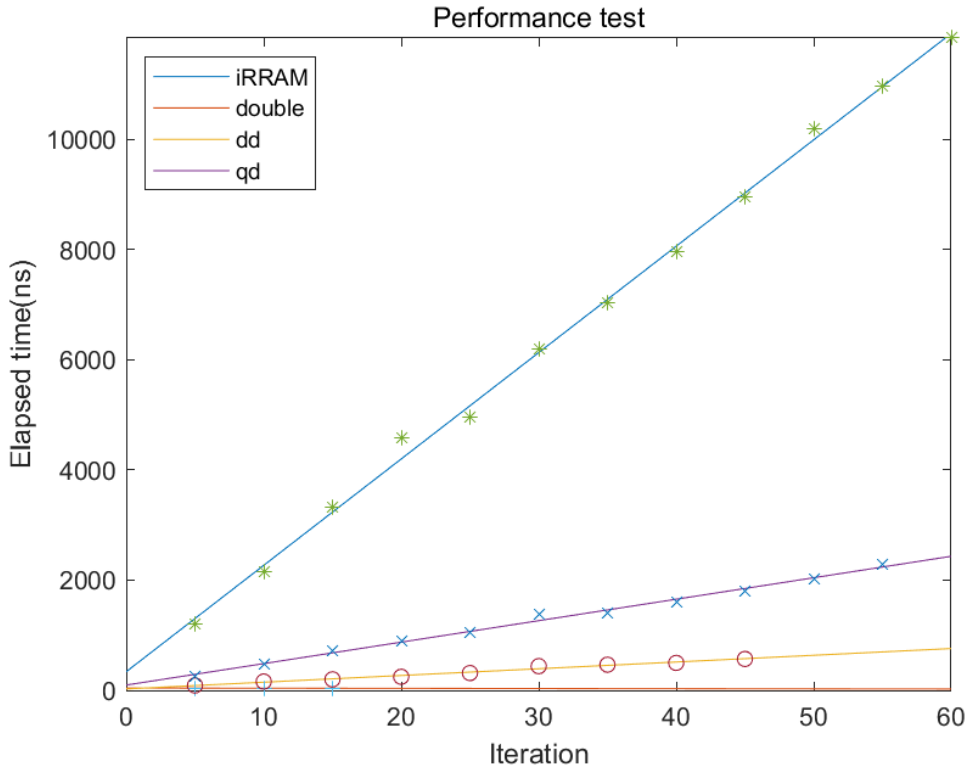

Figure 4: Multiplication Test

itr	iRRAM	double	DD	QD
5	1207	43	90	253
10	2164	40	162	466
15	3314	40	201	728
20	4575	-	248	898
25	4954	-	316	1058
30	6190	-	441	1388
35	7033	-	467	1411
40	7952	-	500	1606
45	8960	-	571	1803
50	10191	-	-	2016
55	10952	-	-	2296
60	11834	-	-	-

	iRRAM	double	DD	QD
slope	192.93	-0.3000	12.210	38.926

Table 2: Elapsed time(ns) and slope(ns/itr)

With dramatically reduced iterations, the same performance order was shown up.

2.3 Sqrt

Test setup: $x_{n+1} = \sqrt{x_n},\ x_0 = 3.14159,\ |e_n| < 2^{-70}$

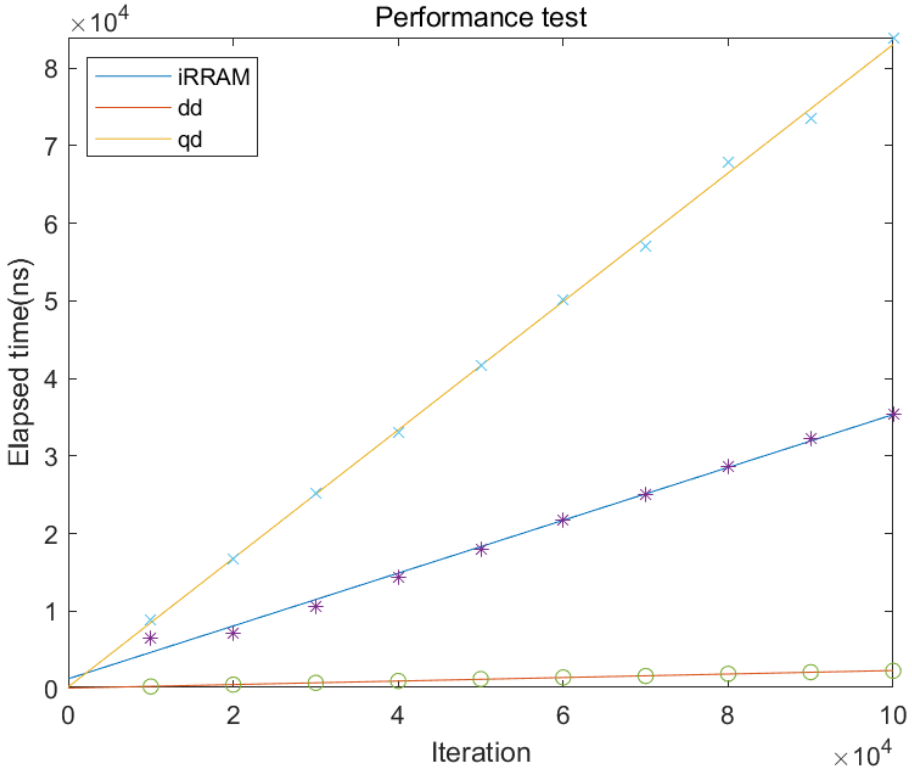

Figure 5: Sqrt Test

itr	iRRAM	double	DD	QD
10	6576	-	280	8824
20	7204	-	518	16734
30	10624	-	752	25149
40	14309	-	986	33089
50	18002	-	1245	41635
60	21792	-	1447	50126
70	25075	-	1635	56960
80	28702	-	1912	67796
90	32293	-	2125	73445
100	35448	-	2300	83925

	iRRAM	double	DD	QD
slope	340.6	-	22.653	828.02

Table 3: Elapsed time(ns) and slope(ns/itr)

double couldn't produce any correct answer. iRRAM outperformed QD. Maybe sqrt for QD is poorly implemented.

2.4 Sin

Test setup: $x_{n+1} = \sin x_n,\ x_0 = 3.14159,\ |e_n| < 2^{-20}$

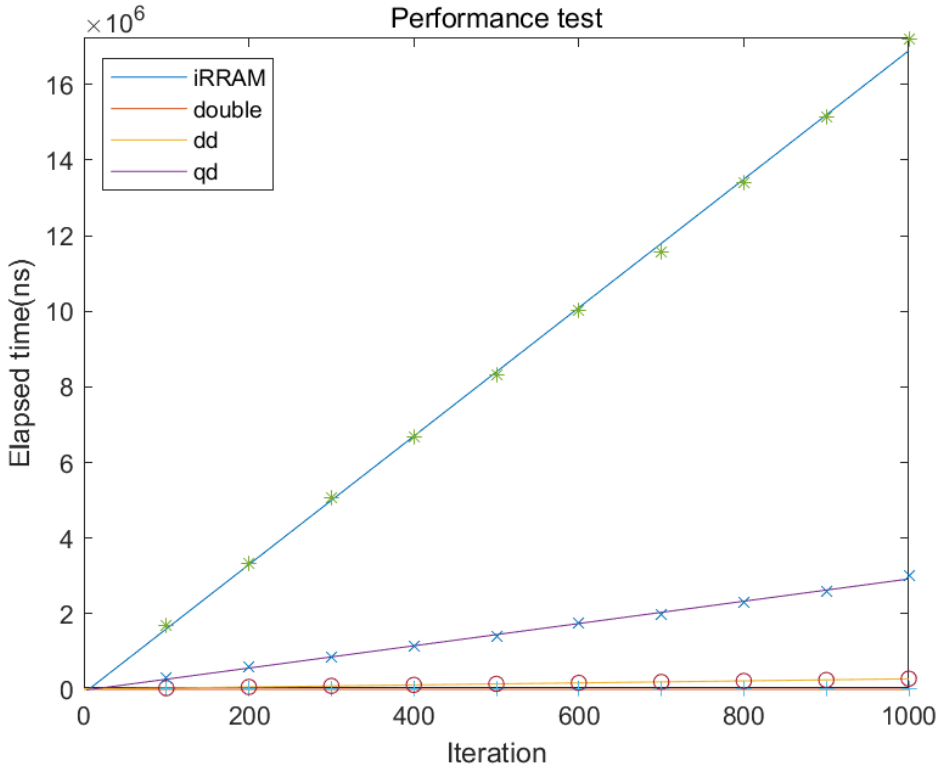

Figure 6: sin Test

itr	iRRAM	double	DD	QD
100	1681688	53	31403	291835
200	3346861	53	63817	584602
300	5080634	56	92358	867127
400	6676655	40	118339	1138493
500	8311261	40	144700	1411165
600	10034068	40	171408	1748068
700	11581949	54	194743	1984476
800	13409914	53	222295	2316362
900	15135516	39	247799	2602703
1000	17218219	40	282751	3000588

	iRRAM	double	DD	QD
slope	17000	-0.0114	270	2947

Table 4: Elapsed time(ns) and slope(ns/itr)

In the last part of iRRAM curve, there is a subtle performance improvement. This is because of the error on measurement. The cpu must have done something else during iteration # 900 case, which had effect on the elapsed time.

2.5 Euler Number (e=2.71...)

In previous cases, each data structure produces true value until the roundoff error takes place. On the other hand, this case produces only approximation of e at every iteration. So, let us change the strategy. We set precision p and aims $|x_n - e| \leq 2^p$. Because double, DD, and QD cannot have different limited precisions, we proceed the experiments with different precisions for each data structure, namely, $p_m \leq p < 0$ where $p_m = -15$ for double, $p_m = -97$ for DD, and $p_m = -196$ for QD.

Now that we are given a precision p, how do we approximate e? We know

$$e = \sum_{k=0}^{n} \frac{1}{k!} + E_n \tag{1}$$

where error E_n is

$$E_n = \sum_{k=n+1}^{\infty} \frac{1}{k!} \tag{2}$$

But

$$\begin{aligned}
E_n =& \frac{1}{(n+1)!} + \frac{1}{(n+2)!} + \frac{1}{(n+3)!} + \cdots \\
=& \frac{1}{(n+1)!}\left[1 + \frac{1}{(n+2)} + \frac{1}{(n+2)(n+3)} + \cdots\right] \\
\leq& \frac{1}{(n+1)!}\left[1 + \frac{1}{1} + \frac{1}{1\times 2} + \frac{1}{1\times 2\times 3}\cdots\right] \\
=& \frac{e}{(n+1)!}
\end{aligned}$$

Meanwhile, $\frac{e}{n+1} \leq 2$ for $n \geq 1$. By multiplying $1/n!$ on both hands,

$$|E_n| = \frac{e}{(n+1)!} \leq \frac{2}{n!} \tag{3}$$

Therefore, we keep adding up $1/k!$ from (1) until $2/n! \leq 2^p$ is satisfied.

2.5.1 iRRAM vs double

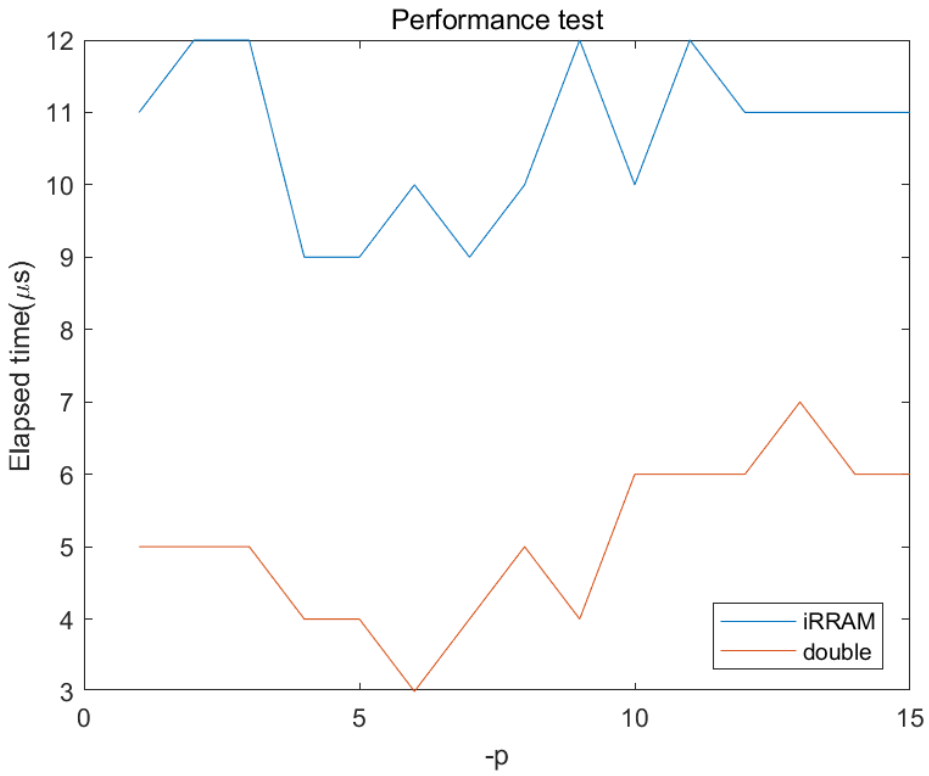

Figure 7: Runtime: iRRAM vs double

-p	iRRAM	double
1	11	5
2	12	5
3	12	5
4	9	4
5	9	4
6	10	3
7	9	4
8	10	5
9	12	4
10	10	6
11	12	6
12	11	6
13	11	7
14	11	6
15	11	6

Table 5: Runtime: iRRAM vs double

Note that x-axis indicates $-p$ instead of p. Double beats iRRAM.

2.5.2 iRRAM vs double double

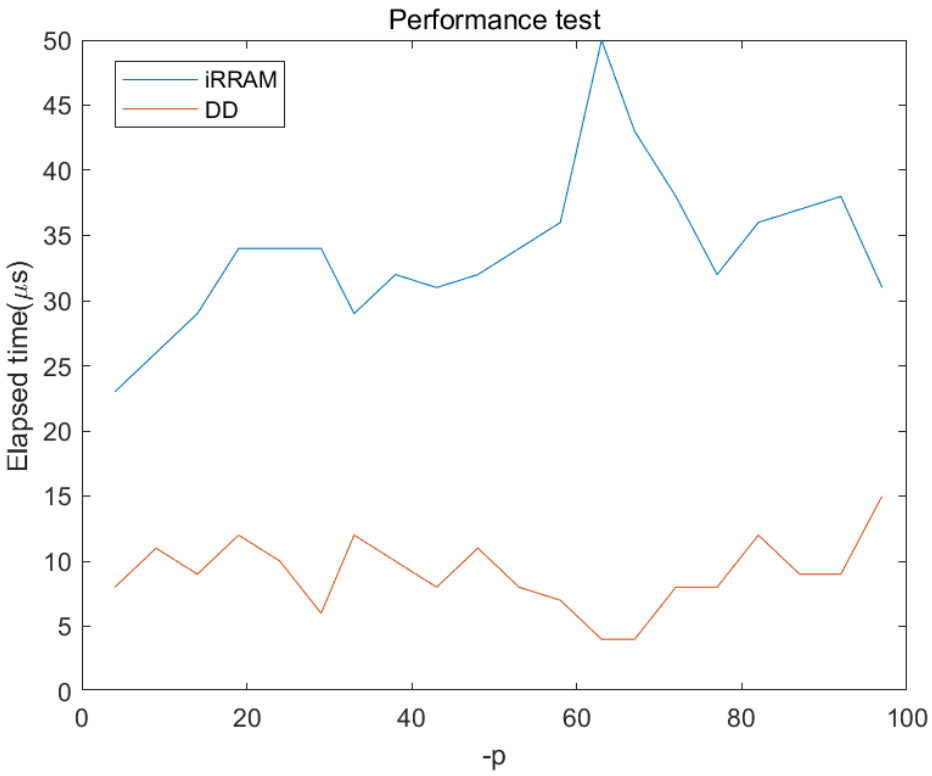

Figure 8: Runtime: iRRAM vs DD

-p	iRRAM	DD
4	23	8
9	26	11
14	29	9
19	34	12
24	34	10
29	34	6
33	29	12
38	32	10
43	31	8
48	32	11
53	34	8
58	36	7
63	50	4
67	43	4
72	38	8
77	32	8
82	36	12
87	37	9
92	38	9
97	31	15

Table 6: Runtime: iRRAM vs DD

Once again, DD beats iRRAM.

2.5.3 iRRAM vs quad double

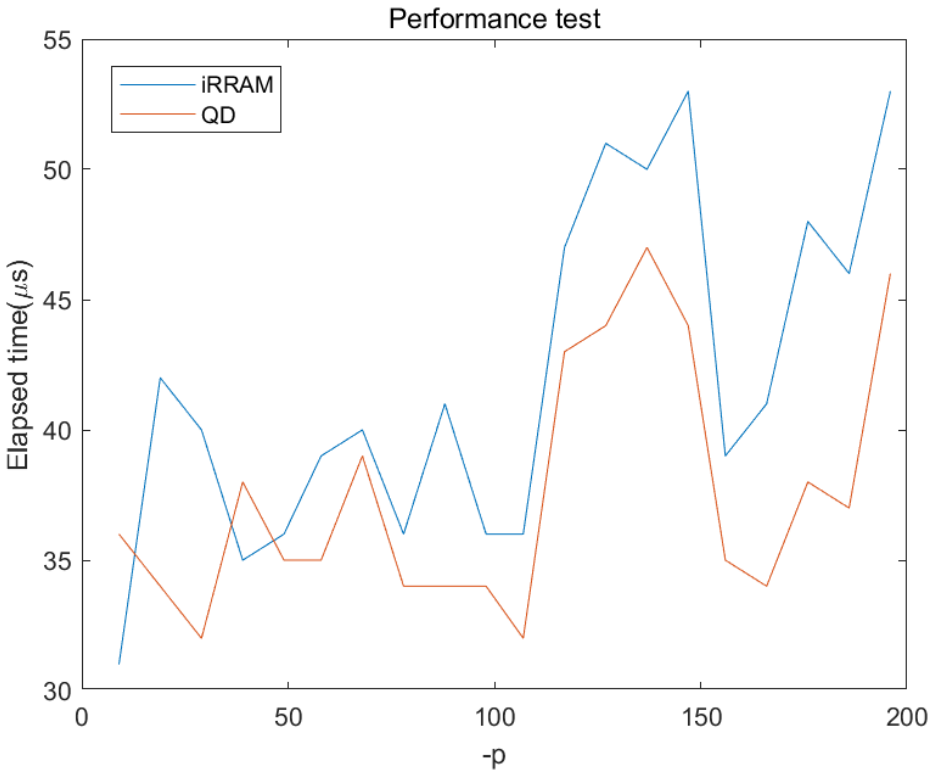

Figure 9: Runtime: iRRAM vs QD

-p	iRRAM	QD
9	31	36
19	42	34
29	40	32
39	35	38
49	36	35
58	39	35
68	40	39
78	36	34
88	41	34
98	36	34
107	36	32
117	47	43
127	51	44
137	50	47
147	53	44
156	39	35
166	41	34
176	48	38
186	46	37
196	53	46

Table 7: Runtime: iRRAM vs QD

This looks pretty different. First, the difference isn't as big as those of the previous cases. Second, there are some sections on which iRRAM runs faster than QD especially on $-p \in (0, 50)$. Whenever we do this experiment, the graph changes. However, these two remarks never change. It is another clue to doubt the implementaion of QD along with Sec 2.3.

3 Conclusion

We couldn't see the break-even point. Double and DD outperformed iRRAM for all experiments. In addition, QD showed slower performance than iRRAM does at Sec 2.3, or similar performance at Sec 2.5.3. Therefore it is efficient to use double, DD, and QD if the desired precision is small enough for those data structure except sqrt.

References

[1] Yozo Hida, Xiaoye S Li, and David H Bailey. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory*, 2007.