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

October 19, 2019

1 Accuracy Test

As iRRAM supports arbitrary precision, all values given by iRRAM is 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 [**?**].

Algorithm 1 Part of accuracy code

```
(...)

// initial values
REAL   xr= 0.5, cr=3.75;
double xd= 0.5, cd=3.75;
dd_real xdd= 0.5, cdd=3.75;
qd_real xqd= 0.5, cqd=3.75;

(...)

// 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

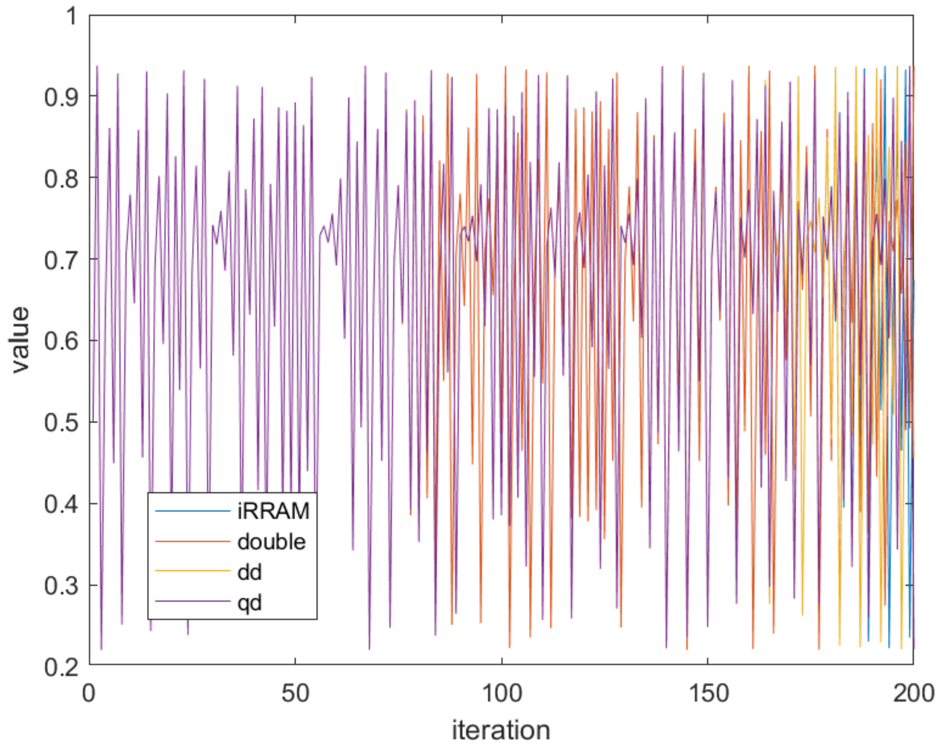

Figure 1: raw values

You probably want to see clear disparity, or error.

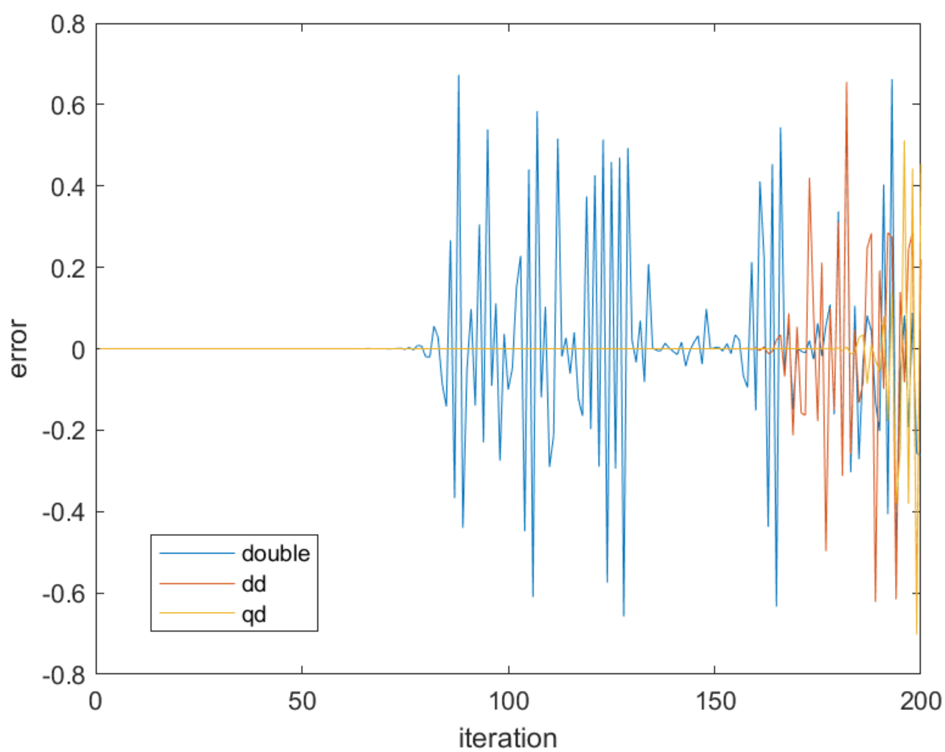

Figure 2: errors

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.

Algorithm 2 Part of code for addition performance test

```
// iRRAM
r = rInit;
beginTime = high_resolution_clock::now();   // timer begin
for(long i=0;i<nPlus;i++) r += rPi;       // apply operation
endTime = high_resolution_clock::now();   // timer end
elapsed = duration_cast<nanoseconds>(endTime-beginTime).count();
// get elapsed time
cout << "iRRAM: " << elapsed << " ns \n";   // print
```

2.1 Addition

Test setup: $x_{n+1} = x_n + c,\ x_0 = 1.0, c = 3.14159$

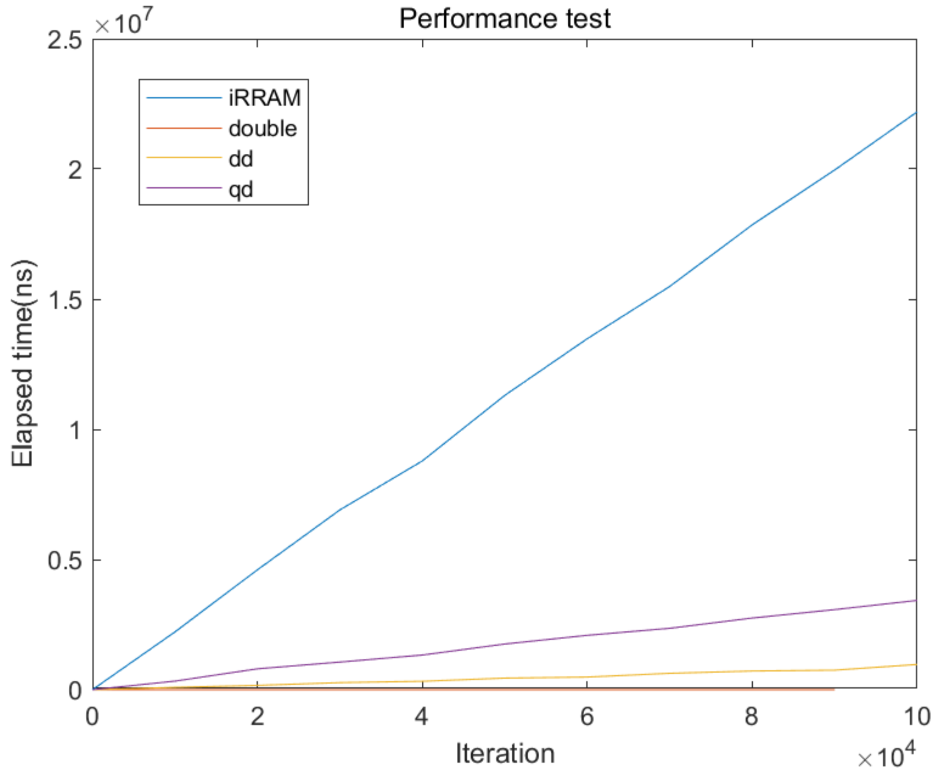

Figure 3: Addition Test

itr	iRRAM	double	DD	QD
10000	2218024	37	89737	329317
20000	4602518	43	165619	801740
30000	6904207	37	270899	1061150
40000	8786108	37	323277	1330845
50000	11305168	41	445272	1754624
60000	13481659	38	484875	2088568
70000	15487987	37	630103	2358365
80000	17854700	36	716757	2752079
90000	19961321	38	750007	3075830
100000	22194439	-	968803	3428676

Table 1: Elapsed time(ns)

Double is the fastest followed by DD, QD, and iRRAM. Double showed a constant time complexity, but failed at iteration # 90936 because of lack of precision.

2.2 Multiplication

Test setup: $x_{n+1} = x_n c,\ x_0 = 1.0, c = 3.14159$

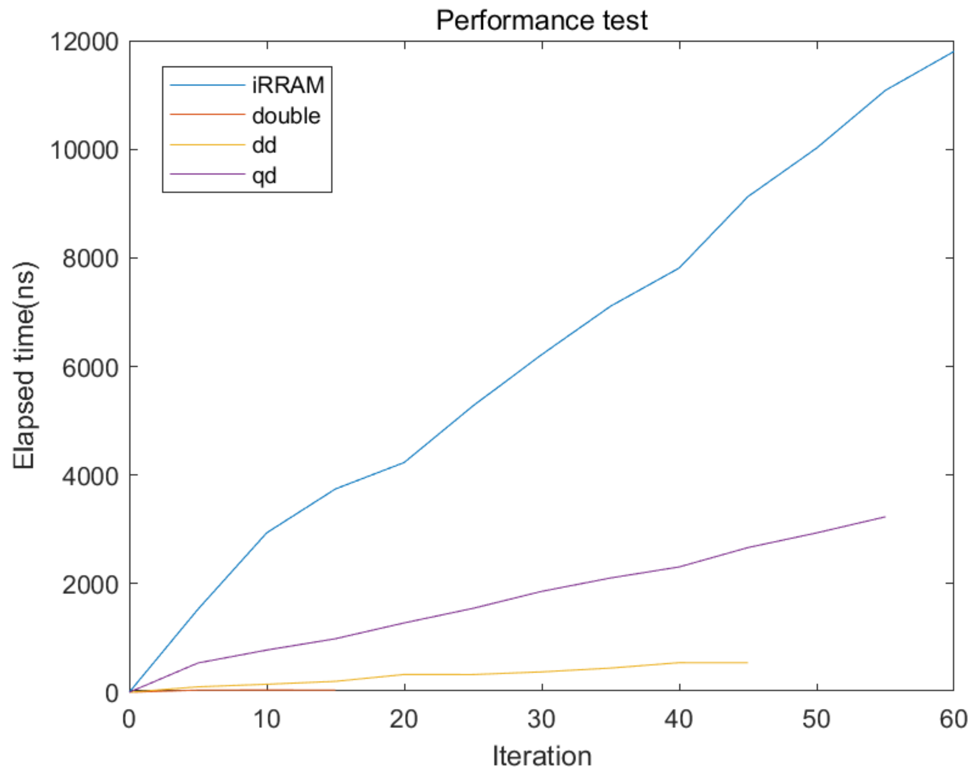

Figure 4: Multiplication Test

itr	iRRAM	double	DD	QD
5	1529	39	100	539
10	2936	41	147	777
15	3744	39	201	986
20	4231	-	327	1277
25	5273	-	326	1545
30	6218	-	375	1858
35	7108	-	445	2104
40	7812	-	545	2308
45	9127	-	543	2664
50	10022	-	-	2935
55	11084	-	-	3231
60	11802	-	-	-

Table 2: Elapsed time(ns)

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

2.3 Sqrt

Test setup: $x_{n+1} = \sqrt{x_n},\ x_0 = 3.14159$

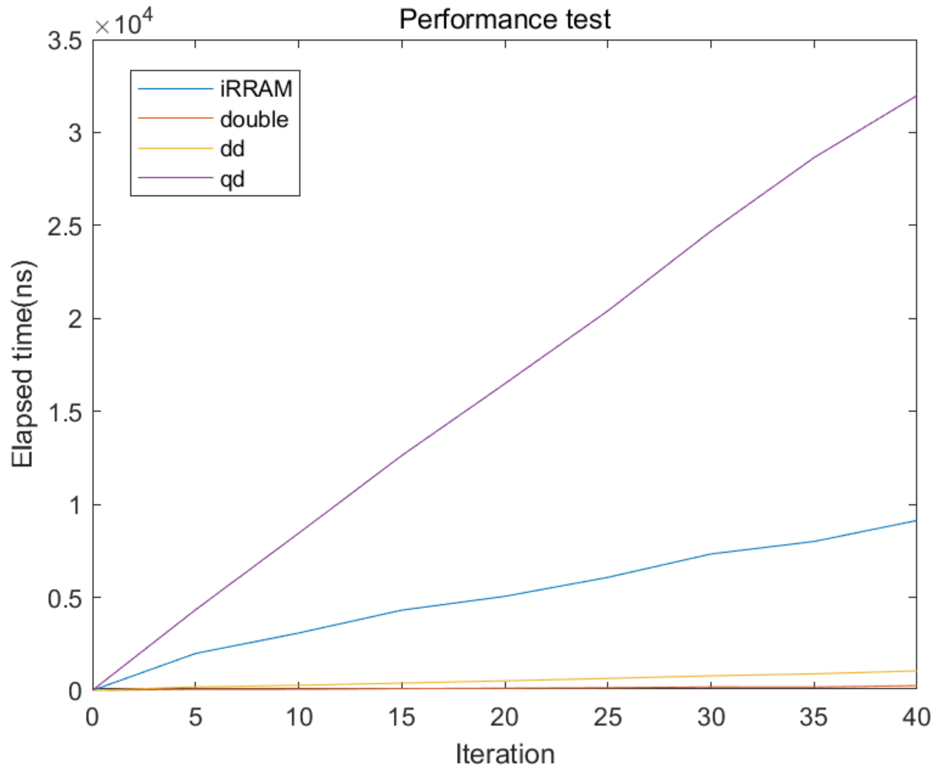

Figure 5: Sqrt Test

itr	iRRAM	double	DD	QD
5	1982	61	186	4333
10	3082	63	272	8447
15	4311	91	390	12624
20	5056	115	512	16489
25	6077	139	643	20411
30	7333	184	781	24697
35	8007	179	887	28649
40	9137	254	1048	31991

Table 3: Elapsed time(ns)

iRRAM outperformed QD. Maybe sqrt for QD is poorly implemented.

2.4 Sin

Test setup: $x_{n+1} = \sin x_n, \; x_0 = 3.14159$

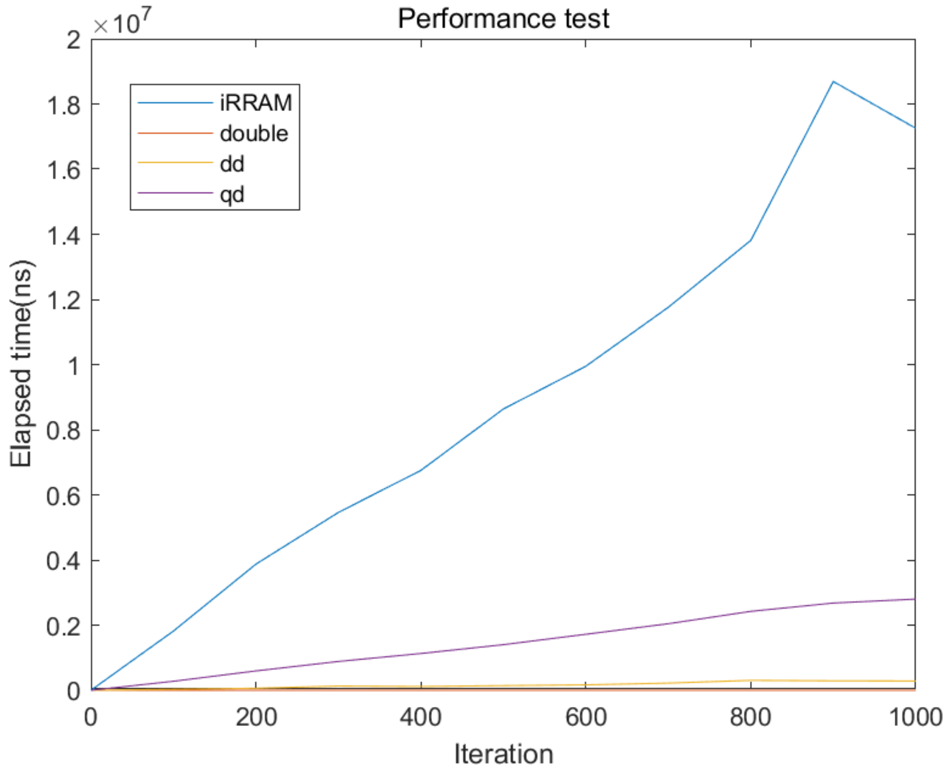

Figure 6: sin Test

itr	iRRAM	double	DD	QD
100	1818310	38	30133	276612
200	3871717	51	61892	594097
300	5459542	39	130214	884736
400	6750785	42	115878	1129187
500	8635247	40	141628	1400049
600	9947132	38	166278	1719012
700	11758728	38	219942	2043182
800	13812230	47	303960	2423032
900	18689463	48	290232	2677895
1000	17253055	50	285337	2798857

Table 4: Elapsed time(ns)

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.

As one last remark, we repeated each experiment 2^{14}times and took the average because outliers came up out of blue.

2.5.1 iRRAM vs double

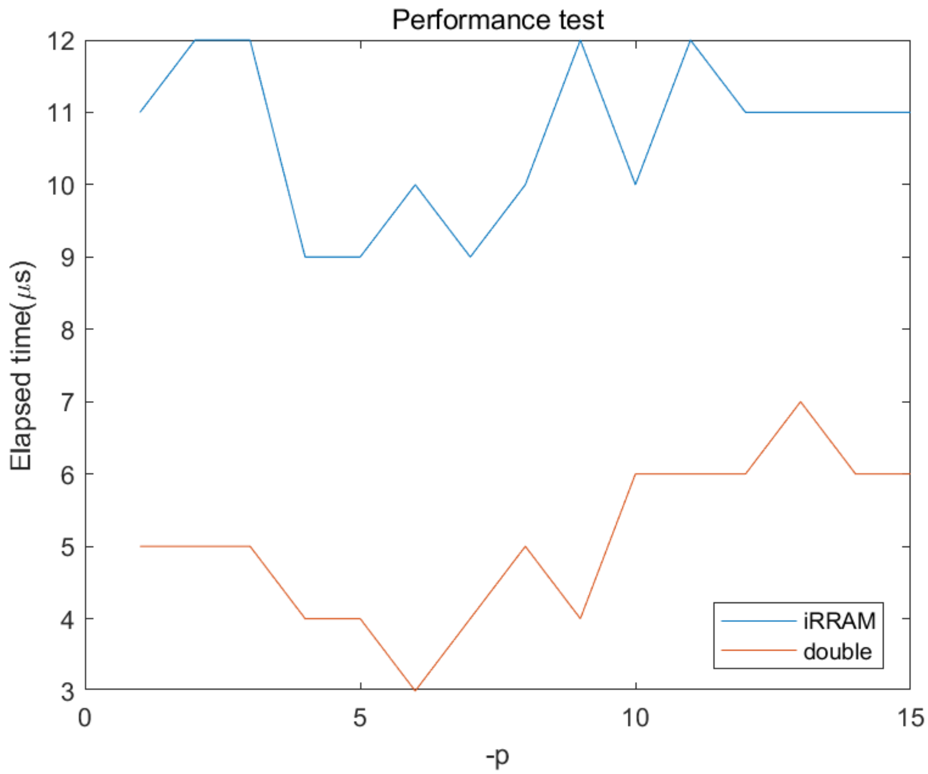

Figure 7: Runtime: iRRAM vs double

-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

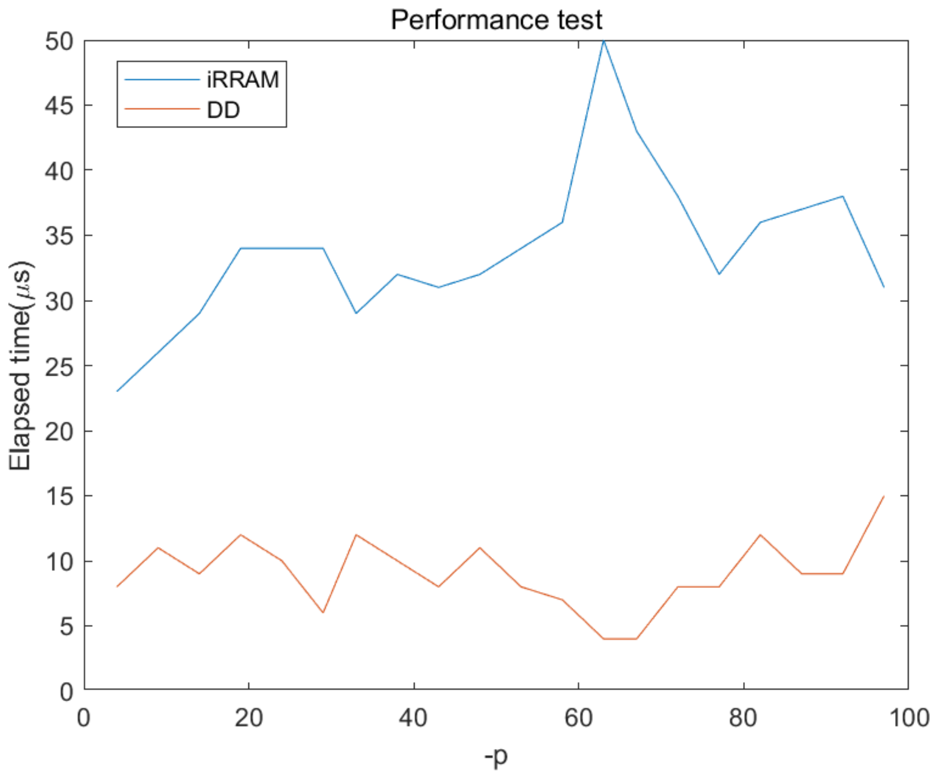

Figure 8: Runtime: iRRAM vs DD

-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

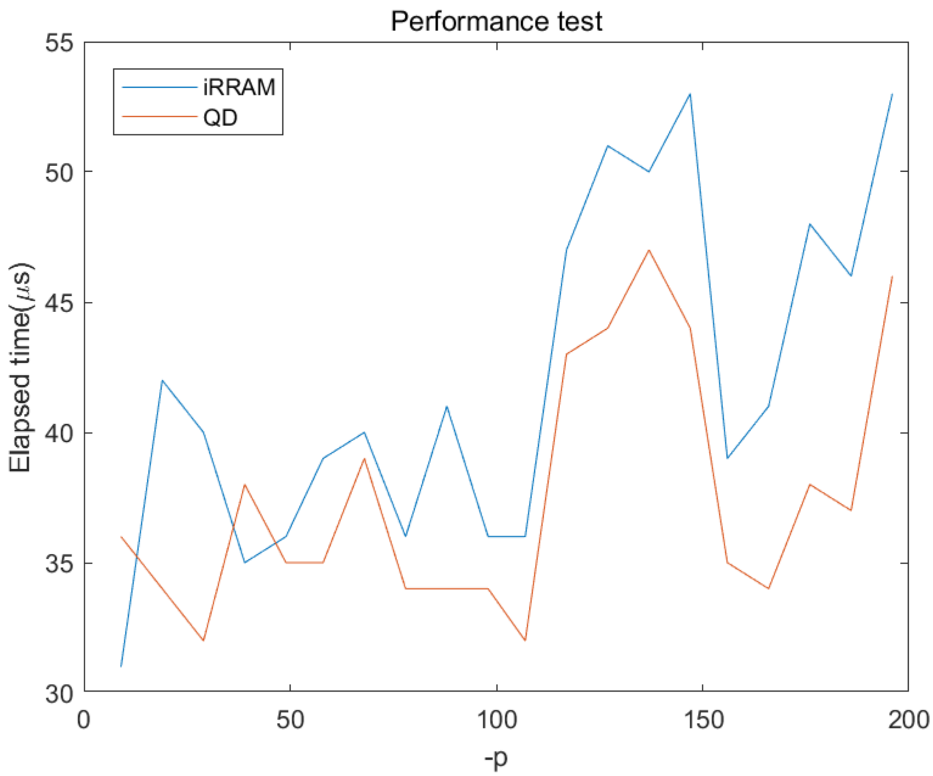

Figure 9: Runtime: iRRAM vs QD

-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.