Balancing Act: Achieving Time and Memory Efficiency in SVP

Michal

Word Count: 742

Approach

My methodology was heavily influenced by my initial research into three of the most known methods of solving SVP:

Algorithm type	**Time complexity**	**Space complexity**
Enumeration	$n^{O(n)}$	$O(n^2)$
Sieving	$O(2^n)$	$2^{O(n)}$
Voronoi	$O(2^{2n})$	$O(2^n)$

Table 1: Time & Space complexities of varying types of Lattice-based algorithms [1]

While Sieving was conceptually the most intuitive, I found that Enumeration was ideal for this task.

Asymptotically, the time complexity of Enumeration is much worse than Sieving or Voronoi, however, empirical evidence suggests that for low-dimensions Enumeration outperforms them. Additionally, Enumeration has polynomial space complexity which is much better than Sieving and Voronoi.

Accuracy

A big worry of this assignment were floating point inaccuracies, hence I initially chose C's built-in `double`.

Arbitrarily, I set an accuracy threshold, T where

$$T = 5 \cdot 10^{-5}$$

This meant that if

$$|\text{Expected result} - \text{Actual result}| \leq T$$

then I would consider my result as correct.

I found this to be a better metric than percentage difference as it ensured correct results were closely aligned with the actual answer, and unaffected by the result's magnitude.

I generated test lattices using `latticegen` from the `fplll` library [2]. It is often referred to as the best lattice-based solver available, hence I trusted its answers. I made multiple bash & python scripts to automate test generation, and focused on uniform & knapsack-like lattices.

Lacking prior experience in C, I decided to begin implementing a prototype in Python.

I found many basis-reduction algorithms, and while BKZ is most commonly used, I struggled to implement it and instead implemented LLL [3]–[5].

Once I implemented LLL and Schnorr Euchner enumeration [6] according to pseudocode, I began testing different configurations, while varying δ.

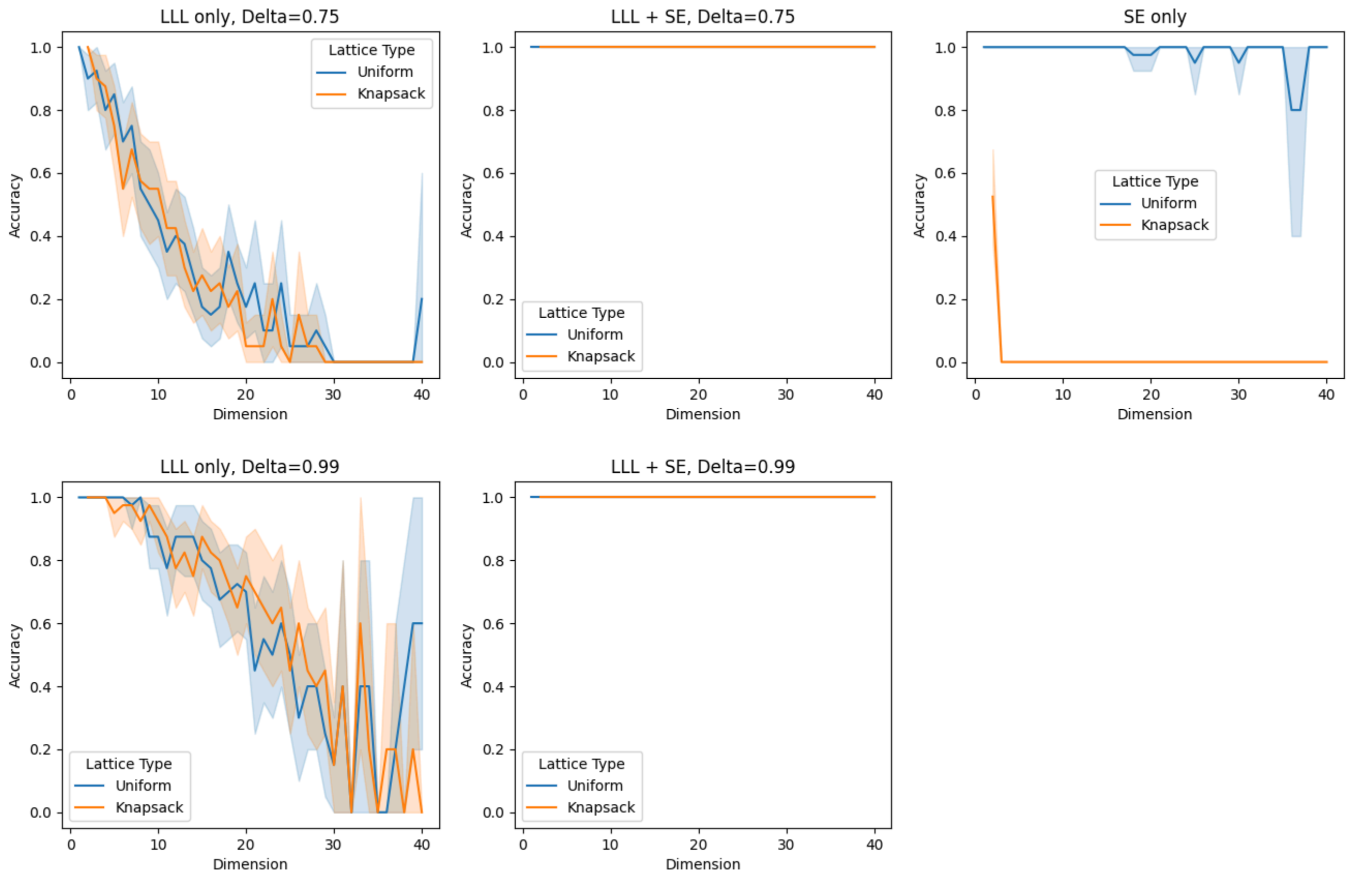


Figure 1: Accuracy vs. Dimension for various algorithm configurations

This highlighted that:

1. LLL and SE on their own had unsatisfactory accuracy
   - LLL gave approximations from its reduced basis which weren't always correct.
   - SE struggled with knapsack-like lattices (due to accumulating floating-point inaccuracies)
2. Increasing δ led to a higher accuracy. This is because a higher δ yields a better basis reduction [7], therefore, a better approximation.
3. LLL and SE combined exhibit superior performance.

To determine whether `long double` was necessary, I tested my implementation using both.

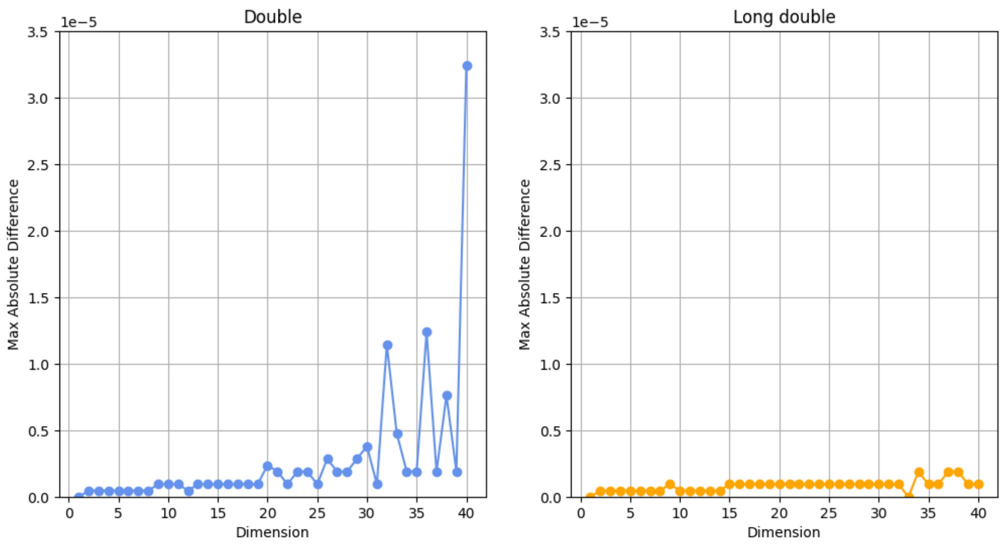
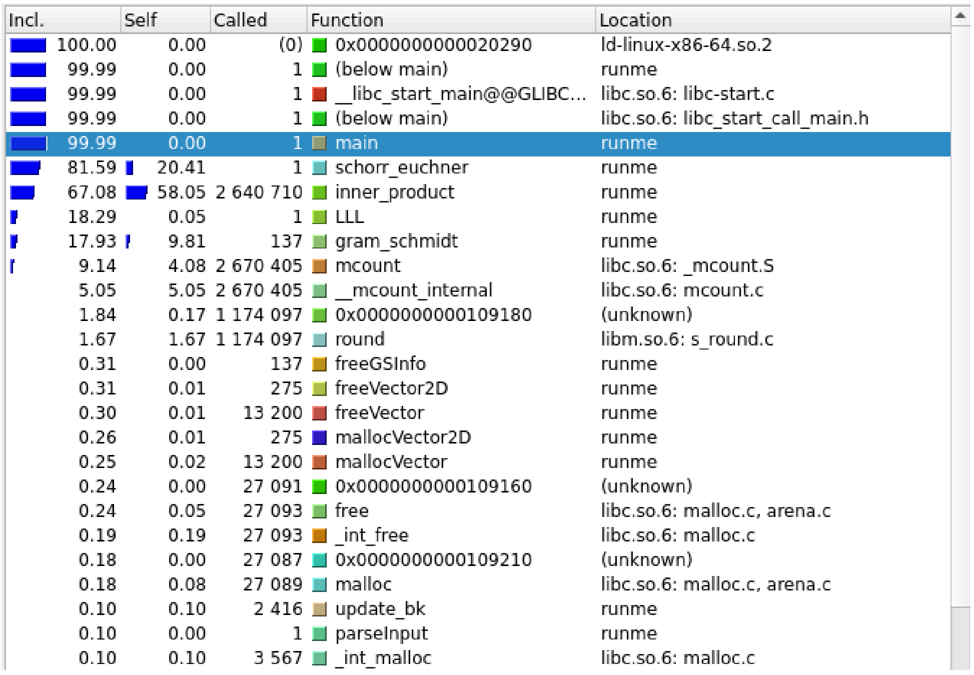


Figure 2: Maximum Absolute Difference vs. Dimension when using double and long double in LLL+SE, δ=0.99

Based on this, and considering the coursework's requirements, I concluded it was unnecessary as the accuracy stayed within the tolerance T. Furthermore, I could not justify the extra memory and computation time needed for `long double`.

Time

Upon running `valgrind`'s `callgrind` and feeding the result into `kcachegrind`, it became evident where optimisations would be most beneficial.



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.99	0.00	1	(below main)	runme
99.99	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.99	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.99	0.00	1	main	runme
81.59	20.41	1	schorr_euchner	runme
67.08	58.05	2 640 710	inner_product	runme
18.29	0.05	1	LLL	runme
17.93	9.81	137	gram_schmidt	runme
9.14	4.08	2 670 405	mcount	libc.so.6: _mcount.S
5.05	5.05	2 670 405	__mcount_internal	libc.so.6: mcount.c
1.84	0.17	1 174 097	0x0000000000109180	(unknown)
1.67	1.67	1 174 097	round	libm.so.6: s_round.c
0.31	0.00	137	freeGSInfo	runme
0.31	0.01	275	freeVector2D	runme
0.30	0.01	13 200	freeVector	runme
0.26	0.01	275	mallocVector2D	runme
0.25	0.02	13 200	mallocVector	runme
0.24	0.00	27 091	0x0000000000109160	(unknown)
0.24	0.05	27 093	free	libc.so.6: malloc.c, arena.c
0.19	0.19	27 093	_int_free	libc.so.6: malloc.c
0.18	0.00	27 087	0x0000000000109210	(unknown)
0.18	0.08	27 089	malloc	libc.so.6: malloc.c, arena.c
0.10	0.10	2 416	update_bk	runme
0.10	0.00	1	parseInput	runme
0.10	0.10	3 567	_int_malloc	libc.so.6: malloc.c

Figure 3: Snippet of function call summary provided by kcachegrind

My `schnorr_euchner`, `lll`, and `gram_schmidt` all relied on calculating millions of `inner_products`.

```c
double inner_product(const Vector v1, const Vector v2, const int dim) {
    double total = 0;
    for (int i = 0; i < dim; i++) {
        total += v1[i] * v2[i];
    }
    return total;
}
```

Code Sample 1: My implentation of the Euclidean Inner Product

The only optimisation here was potentially parallelising using multiple threads, which would only be effective on higher dimensions due to thread overhead.

I realised that by memoising/precalculating inner products I could drastically reduce the number of calls to `inner_product`, thereby decreasing the number of operations.

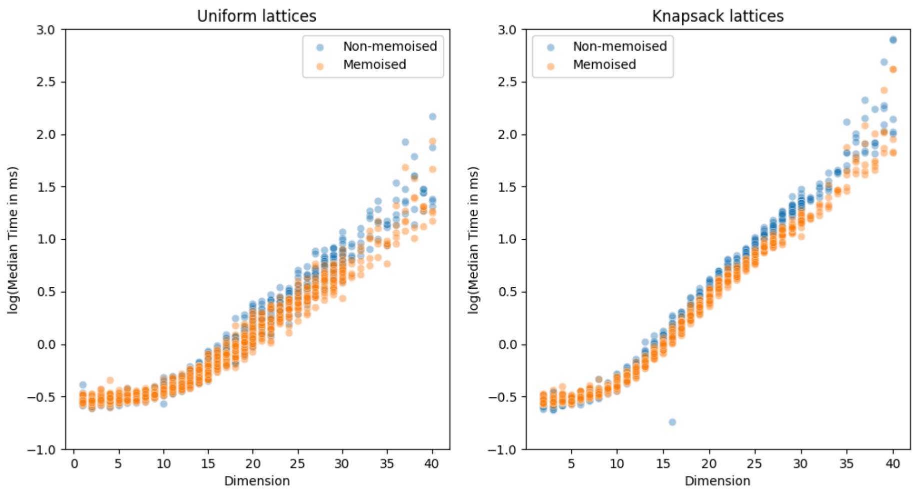


Figure 4: Effects of memoisation on median run-time of LLL + SE, δ=0.99

While the graph isn't perfect, it shows that memoising has a positive impact on performance - this is amplified by the logarithmic y-axis.

I decided to investigate the δ parameter more:

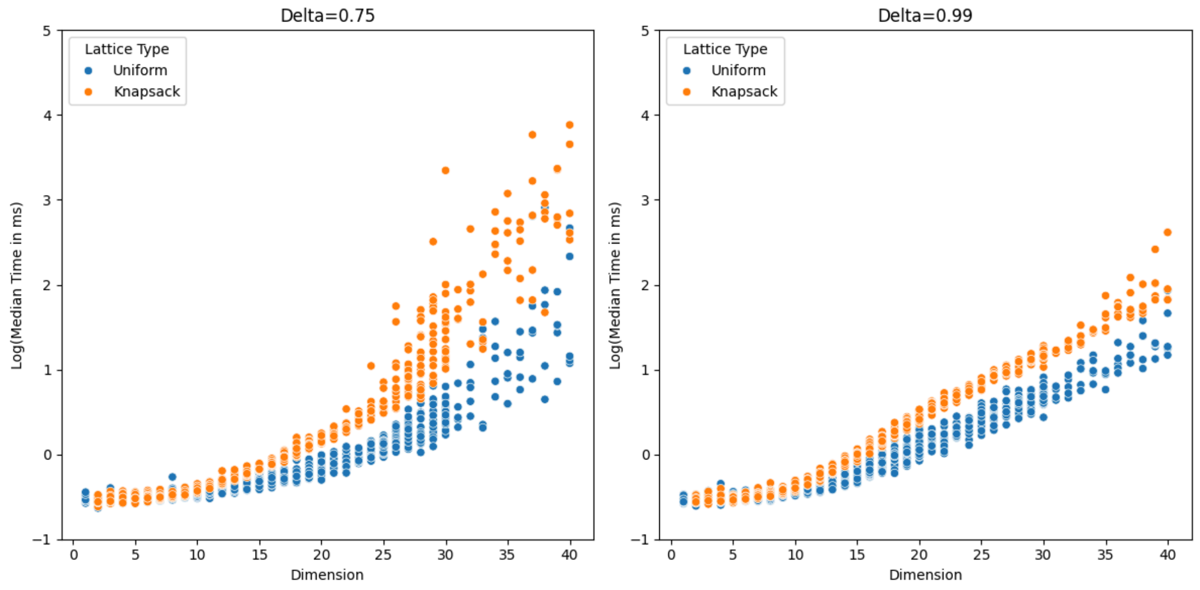


Figure 5: Effects of changing LLL's δ value on median run-time of LLL + SE

I found that for both uniform and knapsack lattices, a higher delta resulted in less variance of run-time, evident by times being less scattered (Figure 5).

Delta	**Dimensions**							
	1-5	**6-10**	**11-15**	**16-20**	**21-25**	**26-30**	**31-35**	**36-40**
0.75	0.30	0.33	0.42	0.59	0.98	2.37	9.89	108.38
0.99	0.30	0.33	0.47	0.93	1.98	3.96	7.92	22.39

Figure 6: Median speed (in ms) of LLL+SE on each dimension range on **Uniform** Lattices
Faster delta value underlined

Delta	**Dimensions**							
	1-5	**6-10**	**11-15**	**16-20**	**21-25**	**26-30**	**31-35**	**36-40**
0.75	0.30	0.34	0.55	1.13	2.78	43.54	227.18	1330.40
0.99	0.30	0.36	0.70	1.92	5.00	11.42	26.48	102.61

Figure 7: Median speed (in ms) of LLL+SE on each dimension range on **Knapsack** Lattices
Faster delta value underlined

For dimensions 10-25/30 (Figure 6 and Figure 7), the run-time using $\delta = 0.99$ increases due to more iterations inside LLL. However, asymptotically, $\delta = 0.99$ was better.

Memory

For memory, I used `valgrind` (tools: `massif` and `dhat`) which helped me address potential memory leaks and segmentation faults.

The `memusage` tool was also useful as it gave me a distribution of memory block sizes, and from this, I could pinpoint inefficiencies in my data structures.

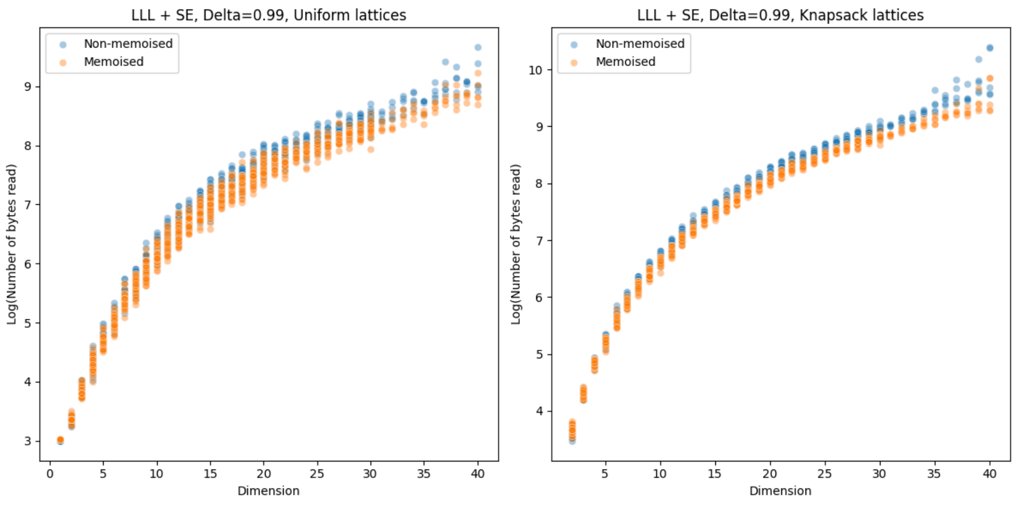


Figure 8: Effects of memoisation on the number of bytes read from memory

From Figure 8 it is clear memoisation leads to less memory reads. This is because, by memoising the inner products, the only value being read from memory is the inner product itself, and not the component vectors needed to compute the inner product.

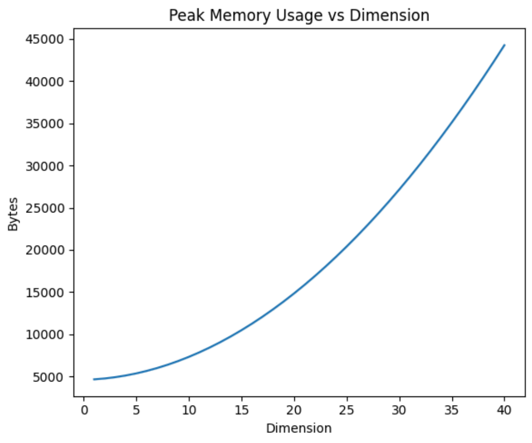


Figure 9: Peak memory usage of LLL + SE, $\delta = 0.99$

Figure 9 shows that

$$\text{Peak memory} \propto (\text{Dimension})^2$$

which aligns with Table 1 and [1]

From Figure 3, it was also clear that `malloc` and `free` were called too often. This was resolved by initialising `GS_Info` at the beginning and then reusing it throughout the program's execution.

Readability

My initial implementation was centered around `structs`, however these were unnecessary and made my code unreadable.

Old implementation

```c
typedef struct {
    double *e;
} Vector;

typedef struct {
    Vector **v;
    int dimension;
} Vector2D;
```

New implementation

```c
typedef double* Vector;

typedef Vector* Matrix;
```

Sample 2: Changes in implementation of `Vector` and `Vector2D`

Old implementation

```c
for (int k = 0; k < i; k++) {
    double ip = inner_product(B->v[i], Bs->v[k], dim);
    mu->v[i]->e[k] = ip / inner_products[k];

    for (int j = 0; j < dim; j++) {
        Bs->v[i]->e[j] -= mu->v[i]->e[k] * Bs->v[k]->e[j];
    }
}
```

New implementation

```c
for (int k = 0; k < i; k++) {
    double ip = inner_product(B[i], Bs[k], dim);
    mu[i][k] = ip / inner_products[k];

    for (int j = 0; j < dim; j++) {
        Bs[i][j] -= mu[i][k] * Bs[k][j];
    }
}
```

Sample 3: Changes in implementation of projection calculation in Gram Schmidt

This led to much better readability, which Sample 3 is a great example of.

Conclusion

Overall, I believe my implementation is quite fast, and accurate to a high number of dimensions. I would like to try this challenge again with a Domain-Specific Language and with more advanced methods to improve upon the performance.

Bibliography

[1] D. Micciancio, “Lattice cryptography”. [Online]. Available: https://cseweb.ucsd.edu/~daniele/LatticeLinks/SVP.html

[2] T. F. development team, “fplll, a lattice reduction library, Version: 5.4.5”. [Online]. Available: https://github.com/fplll/fplll

[3] S. Bhattacherjee, J. Hernandez-Castro, and J. Moyler, “A Greedy Global Framework for LLL”. [Online]. Available: https://eprint.iacr.org/2023/261

[4] S. D. Galbraith, [Online]. Available: https://www.math.auckland.ac.nz/~sgal018/crypto-book/main.pdf

[5] H. Cohen, “A course in computational algebraic number theory”. 1993. doi: 10.1007/978-3-662-02945-9.

[6] M. Yasuda, “A Survey of Solving SVP Algorithms and Recent Strategies for Solving the SVP Challenge”. pp. 189–207, 2021. doi: 10.1007/978-981-15-5191-8_15.

[7] N. (. Landsman, G. Lord, and G. Heckman, “Reduction of Matrices and Lattices”. [Online]. Available: https://www.math.ru.nl/magma/text536.html

[8] Z. Zheng, X. Wang, G. Xu, and Y. Yu, “Orthogonalized lattice enumeration for solving SVP”. [Online]. Available: https://doi.org/10.1007/s11432-017-9307-0

[9] K. Matsuda, A. Takayasu, and T. Takagi, “Explicit Relation between Low-Dimensional LLL-Reduced Bases and Shortest Vectors”. pp. 1091–1100, 2019. doi: 10.1587/transfun.E102.A.1091.

[10] F. Correia, A. Mariano, A. Proença, C. Bischof, and E. Agrell, “Parallel Improved Schnorr-Euchner Enumeration SE++ for the CVP and SVP”, vol. 0, no. pp. 596–603, 2016. doi: 10.1109/PDP.2016.95.