Written by Marcel K. Goh. Last updated July 28, 2020 at 11:02

1. Introduction. This literate program performs lattice reduction using the celebrated LLL algorithm of A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász [*Math. Annalen* **261** (1982), 515–534]. It is a C implementation of the algorithm as described and analysed by H. Cohen in Section 2.6.1 of his book *A Course in Computational Algebraic Number Theory* (New York: Springer, 1996).

Vectors will be represented as C arrays, but since arrays are 0-indexed in C, we will always allocate one extra entry of memory and then keep the zeroth cell empty. This is for consistency with the usual numbering $\mathbf{b}_1, \ldots, \mathbf{b}_n$ of vectors in a basis.

The input to the program is a set of n vectors $(\mathbf{b}_i)$ that form a $\mathbf{Z}$-basis for the lattice L that we wish to reduce. We also need to specify the quadratic form q, which is done with a matrix Q. If x is a vector, then the function $b(x, y) = Qx \cdot y$ is bilinear (where $\cdot$ is the ordinary Euclidean dot-product), and we have the associated quadratic form $q(x) = b(x, x) = Qx \cdot x$.

This program does not take input from the console. To change its arguments, modify the three macros `DIM`, `INPUT_BASIS`, and `INPUT_QUAD`. The LLL-reduced basis will be printed as well as a change-of-basis matrix H.

2. This is the main outline of the program.

```
#define DIM 3
#define INPUT_BASIS {{15.0, 23.0, 11.0}, {46.0, 15.0, 3.0}, {32.0, 1.0, 1.0}}
#define INPUT_QUAD {{1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}}
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
  int n;    /* global variables, for convenience */
  double bb[DIM + 1][DIM + 1], Q[DIM + 1][DIM + 1];
  ⟨Linear algebra subroutines 3⟩;
  ⟨Lattice reduction algorithm lll 4⟩;
  int main()
  {
    n = DIM;
    ⟨Format input into global variables 11⟩;
    int **H;
    H = lll(bb);    /* set H to the output of the LLL algorithm, modify bb in place */
    if (H ≠ Λ) {
      ⟨Output basis bb 12⟩;
      ⟨Output matrix H 14⟩;
      return 0;
    }
    else {
      return 1;
    }
  }
```

3. Linear algebra subroutines. We begin with some linear algebra subroutines that will help us treat arrays as vectors. Calling *set*(z, x) sets the entries of z to the entries of x, while *sub*(z, x, y) stores the vector difference of x and y to z. We can scale a vector with *scale*, and the function *dot* is the ordinary Euclidean dot-product. The functions b and q both rely on the matrix Q; we have $q(x) = Qx \cdot x$ and $b(x, y) = Qx \cdot y$.

⟨ Linear algebra subroutines 3 ⟩ ≡

```
void set(double z[n], double x[n])
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = x[i];
  }
}
void add(double z[n], double x[n], double y[n])
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = x[i] + y[i];
  }
}
void sub(double z[n], double x[n], double y[n])
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = x[i] − y[i];
  }
}
void scale(double z[n], double lambda, double x[n])
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = lambda ∗ x[i];
  }
}
void set_i(int z[n], int x[n])      /∗ integer versions of set, add, sub, and scale ∗/
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = x[i];
  }
}
void add_i(int z[n], int x[n], int y[n])
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = x[i] + y[i];
  }
}
void sub_i(int z[n], int x[n], int y[n])
{
  for (int i = 1; i ≤ n; ++i) {
    z[i] = x[i] − y[i];
  }
}
void scale_i(int z[n], int lambda, int x[n])
{
  for (int i = 1; i ≤ n; ++i) {
```

```
      z[i] = lambda * x[i];
    }
  }
  double dot(double x[n], double y[n])
  {
    double sum = 0;
    for (int i = 1; i ≤ n; ++i) {
      sum += x[i] * y[i];
    }
    return sum;
  }
  double b(double x[n], double y[n])
  {
    double sum = 0;
    for (int i = 1; i ≤ n; ++i) {
      sum += dot(Q[i], x) * y[i];
    }
    return sum;
  }
  double q(double x[n])
  {
    return b(x, x);
  }
```

This code is used in section 2.

4. The LLL lattice reduction algorithm. This is the interesting part of the program. The variable *bb* denotes the basis $(\mathbf{b}_i)$. We will use the Gram-Schmidt orthogonalisation procedure to find an orthogonal basis $(\mathbf{b}_i^*)$, but we do this incrementally, as the algorithm progresses. We keep track of the dot products $\mathbf{b}_i^* \cdot \mathbf{b}_i^*$ in the array B.

The variable k is the main loop variable, but it doesn't always increase from iteration to iteration; sometimes it decreases and sometimes it maintains its value. We will therefore need to store *k_max*, the largest value that k has attained. For $1 \le k, j \le n$, $\mu_{k,j} = b(\mathbf{b}_k, \mathbf{b}_j^*)/q(\mathbf{b}_j^*)$. We will not want to compute this every time it is needed, so we store the μ values in a table called *mu*.

The basis $(\mathbf{b}_i)$ is modified in place so that it is LLL-reduced once the algorithm terminates. The output is an integer matrix H that represents the new, reduced basis in terms of the original basis, i.e., if M is the matrix whose columns are the vectors $\mathbf{b}_i$, then $M \cdot H$ has the LLL-reduced basis as its columns. Note that H_i is the ith *column* of H.

```
⟨ Lattice reduction algorithm lll 4 ⟩ ≡
  int **lll(double bb[n + 1][n + 1])
  {
    int k, k_max, l;
    int **H = malloc((n + 1) * sizeof(int *));
    double mu[n + 1][n + 1];
    double bb_star[n + 1][n + 1];
    double B[n + 1];
    double temp[n + 1], tempb[n + 1];     /* temporary arrays for calculations */
    int temp_i[n + 1];
    ⟨ Initialisation 5 ⟩;
    int num_loops = 0;
    do {
      if (k > k_max) {
        ⟨ Add one Gram-Schmidt vector 6 ⟩;
      }
      l = k − 1;
      ⟨ Reduce bb[k] by subtracting multiples of bb[l] 7 ⟩;
      if (⟨ Lovász condition 8 ⟩) {
        for (l = k − 2; l > 0; −−l) {
          ⟨ Reduce bb[k] by subtracting multiples of bb[l] 7 ⟩;
        }
        ++k;
      }
      else {
        ⟨ Swap bb[k] with bb[k − 1] 9 ⟩;
        k = (2 > k − 1) ? 2 : k − 1;
        continue;
      }
    } while (k ≤ n);
    return H;
  }
```

This code is used in section 2.

5. A **for**-loop initialises the *mu* and *bb_star* arrays to 0 and sets the H matrix to the identity. The Gram-Schmidt procedure is kickstarted by setting $\mathbf{b}_1^* \leftarrow \mathbf{b}_1$, and the main loop variable k is set to 2.

⟨ Initialisation 5 ⟩ ≡

```
for (int i = 1; i ≤ n; ++i) {
   H[i] = malloc((n + 1) ∗ sizeof(int));
   B[i] = 0;
   for (int j = 1; j ≤ n; ++j) {
      H[i][j] = (i ≡ j) ? 1 : 0;
      mu[i][j] = bb_star[i][j] = 0.0;
   }
}
k = 2;
k_max = 1;
set(bb_star[1], bb[1]);
B[1] = q(bb_star[1]);
```

This code is used in section 4.

6. If k is bigger than it has ever been, we do exactly one step of the Gram-Schmidt orthogonalisation procedure. To add a new vector $\mathbf{b}_k$ to the orthogonal basis, we trim away all components of $\mathbf{b}_k$ that are not orthogonal to some $\mathbf{b}_j^*$ for $j < k$. At the end of this process, the $\mathbf{b}_k^*$ can safely be added to the orthogonal basis $(\mathbf{b}_i^*)$ if it is nonzero; otherwise, there is some linear dependence in the original basis $(\mathbf{b}_i)$ so we signal an error and return Λ.

⟨ Add one Gram-Schmidt vector 6 ⟩ ≡

```
k_max = k;
set(bb_star[k], bb[k]);
for (int j = 1; j < k; ++j) {
   mu[k][j] = b(bb[k], bb_star[j])/B[j];
   scale(temp, mu[k][j], bb_star[j]);
   sub(bb_star[k], bb_star[k], temp);
}
B[k] = b(bb_star[k], bb_star[k]);
if (B[k] < DBL_EPSILON) {
   printf("ERROR:␣The␣input␣vectors␣do␣not␣form␣a␣basis.\n");
   return Λ;
}
```

This code is used in section 4.

7. When we want to determine if a candidate vector $\mathbf{b}_k$ is to be added into the lattice, we can subtract integer multiples of a vector $\mathbf{b}_l$ already in the basis. The result is sort of a "remainder vector" (taking a vector "modulo" another should remind you of Euclidean division), that becomes the new working vector. We will also have to update the H matrix and the mu table.

⟨ Reduce $bb[k]$ by subtracting multiples of $bb[l]$ 7 ⟩ ≡

```
if (fabs(mu[k][l]) > 0.5) {
   int rounded = (int) floor((0.5 + mu[k][l]));
   scale(temp, rounded, bb[l]);
   sub(bb[k], bb[k], temp);      /* subtract some integer multiple of b_l */
   scale_i(temp_i, rounded, H[l]);
   sub_i(H[k], H[k], temp_i);
   mu[k][l] = mu[k][l] − rounded;
   for (int i = 1; i < l; ++i) {
      mu[k][i] = mu[k][i] − rounded * mu[l][i];
   }
}
```

This code is used in section 4.

8. At this stage of the algorithm, we are trying to determine if a candidate vector $\mathbf{b}_k$ should be added to the LLL-reduced basis. This is done by checking the so-called Lovász condition, namely,

$$B_k \geq (3/4 - {\mu_{k,k-1}}^2) B_{k-1}.$$

If it is satisfied, we can add $\mathbf{b}_k$ to the LLL-basis; but if not (this happens when $\mathbf{b}_{k-1}$ is "too long", in some sense), we must swap $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$ and update the auxiliary tables accordingly. After this step, $\mathbf{b}_{k-1}$ is discarded because $\mathbf{b}_k$ is still the candidate vector, but now the LLL-reduced basis is only $(\mathbf{b}_1, \ldots, \mathbf{b}_{k-2})$.

⟨ Lovász condition 8 ⟩ ≡

```
B[k] ≥ (0.75 − mu[k][k − 1] * mu[k][k − 1]) * B[k − 1]
```

This code is used in section 4.

9. Here we perform the gnarly task of swapping $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$. This means all of the auxiliary arrays and tables must be updated.

⟨ Swap $bb[k]$ with $bb[k-1]$ 9 ⟩ ≡

```
set(temp, bb[k]);    /* swap b_k with b_{k-1} */
set(bb[k], bb[k - 1]);
set(bb[k - 1], temp);
set_i(temp_i, H[k]);    /* swap H_k with H_{k-1} */
set_i(H[k], H[k - 1]);
set_i(H[k - 1], temp_i);
double t, m;    /* temporary scalars */
if (k > 2) {
  for (int j = 1; j <= k - 2; ++j) {
    t = mu[k][j];    /* swap mu_{k,j} with mu_{k-1,j} */
    mu[k][j] = mu[k - 1][j];
    mu[k - 1][j] = t;
  }
}
m = mu[k][k - 1];
t = B[k] + m * m * B[k - 1];
mu[k][k - 1] = m * B[k - 1]/t;
set(tempb, bb_star[k - 1]);
scale(temp, m, tempb);
add(bb_star[k - 1], bb_star[k], temp);
scale(tempb, B[k]/t, tempb);
scale(temp, -1.0 * mu[k][k - 1], bb_star[k]);
add(bb_star[k], temp, tempb);
B[k] = B[k - 1] * B[k]/t;
B[k - 1] = t;
for (int i = k + 1; i <= k_max; ++i) {
  t = mu[i][k];
  mu[i][k] = mu[i][k - 1] - m * t;
  mu[i][k - 1] = t + mu[k][k - 1] * mu[i][k];
}    /* phew! */
```

This code is used in section 4.

10. Input-output functionality. These components of the *main* function format the input and print the output to the console.

11. ⟨ Format input into global variables 11 ⟩ ≡

```
double input_lattice[DIM][DIM] = INPUT_BASIS;
double input_quad[DIM][DIM] = INPUT_QUAD;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    bb[i + 1][j + 1] = input_lattice[i][j];
    Q[i + 1][j + 1] = input_quad[i][j];
  }
}
printf("Input␣lattice␣basis:\n");
⟨ Print bb 13 ⟩;
printf("Input␣Q␣matrix:\n");
for (int j = 1; j ≤ n; ++j) {
  for (int i = 1; i ≤ n; ++i) {
    printf("%f␣", (Q[i][j]));
  }
  printf("\n");
}
```

This code is used in section 2.

12. ⟨ Output basis *bb* 12 ⟩ ≡

```
printf("Reduced␣basis:\n");
⟨ Print bb 13 ⟩;
```

This code is used in section 2.

13. ⟨ Print *bb* 13 ⟩ ≡

```
for (int i = 1; i ≤ n; ++i) {
  printf("(");
  for (int j = 1; j ≤ n; ++j) {
    printf("%f", (bb[i][j]));
    if (j ≠ n) printf(",␣");
  }
  printf(")\n");
}
```

This code is used in sections 11 and 12.

14. Note that we interchanged i and j in the loops, because $H[i]$ is the ith column of H.

⟨ Output matrix H 14 ⟩ ≡

```
printf("H␣matrix:\n");
for (int j = 1; j ≤ n; ++j) {
  for (int i = 1; i ≤ n; ++i) {
    printf("%d␣", (H[i][j]));
  }
  printf("\n");
}
```

This code is used in section 2.

15. Index.

LLL