

A Tree-code Algorithm Program

The program `treecode.cpp` implements a simple tree-code algorithm for a 2-D system of charges. The code evaluates the potential energy of the system. With very simple modifications, it can be used to compute the Coulomb forces and hence the accelerations of the particles.

We first include standard headers. The new header `list` defines template list objects which will be used to store indexes of particles in the nodes of the quadtree. This could also be done with C++ arrays, but this is a nice application of the C++ Standard Template Library (STL) `list` containers.

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <complex>
#include <ctime>
#include <list>

using namespace std;

int N = 1000;           // number of charges
double L = 10;          // side of square region containing charges
complex<double> *z;      // positions of particles
double *q;              // charges of particles
complex<double> Epot;    // total potential energy of system
```

Variables to implement the quadtree structure

```
int levels;             // number of levels of subdivision
```

```
list<int> **boxList;      // 2-D array of lists of particle indexes
int M = 1;               // maximum order of multipole expansion
complex<double> *a;      // complex multipole moments in a box

// function declarations
int boxesEachDirection(int level); // number of boxes at a given level
void sortBodies(int level);        // sorts and stores indexes by box
void computeMoments(int level);    // finds multipole moments for a box
```

Memory is allocated for the global arrays at the start of the program and the number of `levels` i.e., generations of children in the quadtree is computed.

```
void initialize() {

    // allocate arrays
    z = new complex<double> [N];
    q = new double [N];

    // place particles with random charges at random locations
    for (int i = 0; i < N; i++) {
        q[i] = 2 * rand() / double(RAND_MAX) - 1;
        double x = L * rand() / double(RAND_MAX);
        double y = L * rand() / double(RAND_MAX);
        z[i] = complex<double>(x, y);
    }
}
```

The number of levels is computed as the smallest integer ℓ for which $4^\ell \geq N$ the number of particles. Thus the smallest boxes will contain fewer than one particle on average.

```
// compute number of levels
levels = 0;
int boxes = 1;
while (boxes < N) {
    boxes *= 4;
    ++levels;
}
```

At the finest level of subdivision the number of boxes $4^\ell = B^2$ where B is the number of boxes in each of the x and y directions. A separate function is called to compute B , and then a $B \times B$ array of lists of integers is allocated. Thus there are B^2 lists. Each list will contain the particle indices of the box associated with that array element.

```
int B = boxesEachDirection(levels);

// allocate memory for lists
boxList = new list<int>* [B];
for (int i = 0; i < B; i++)
    boxList[i] = new list<int> [B];
```

Memory is allocated to hold the multipole moments

$$a_0 = \sum_i^{N_c} q_i, \quad a_k = \sum_i^{N_c} \frac{q_i z_i^k}{k}, \quad k = 1, 2, \dots, M$$

due to the N_c charges in a particular box. Here z_i is the position of the charge relative to the center of the box.

```
// allocate memory for multipole moments
a = new complex<double> [M + 1];
```

```
}
```

Here is the definition of the function which computes the number of boxes in each direction at a given `level`. This calculation is needed at several places in the program.

```
int boxesEachDirection(int level) {  
  
    // find number of boxes in each direction  
    int B = 1;  
    for (int i = 0; i < level; i++)  
        B *= 2;  
  
    return B;  
}
```

In the calculation of the potential energy, or the accelerations of the charges, the positions of the charges remain fixed. At any given `level` in the quadtree algorithm, the system volume L^2 is subdivided into B^2 boxes. The charges are sorted into these boxes. Each element `boxList[i][j]` will contain a list of indexes, i.e., the integers $0, 1, \dots, N-1$ of particles whose position is inside this box.

Since the same data structure `boxList` is reused at every level, previous lists are deleted using the `clear()` member function of the `list` class. Each complex particle coordinate `z[n]` is converted to box indexes `i,j` and the index `n` of the particle is added to the `boxList[i][j]` using the `push_front` member function of the `list` class.

```
void sortBodies(int level) {  
  
    int B = boxesEachDirection(level);
```

```
// clear any lists previously constructed
for (int i = 0; i < B; i++)
    for (int j = 0; j < B; j++)
        boxList[i][j].clear();

// add particle indexes to list
for (int n = 0; n < N; n++) {
    int i = int(z[n].real() / L * B);
    int j = int(z[n].imag() / L * B);
    boxList[i][j].push_front(n);
}

}
```

The following function computes the multipole moments in a box with indexes i, j at a particular level with B boxes in each direction. It is assumed that the positions of the particles in the box have already been stored in `boxList[i][j]` by the `sortBodies` function.

This function uses an “iterator” object to traverse the list of particle indices. An iterator is basically a cursor or pointer to a member of the list. When the iterator is incremented using the `++` operator, it moves to the next member in the list. The value stored in a list member is accessed using the dereference `*` operator on the iterator. If you want to use STL container classes you need to become familiar with the use of iterators. The member function `begin()` returns a pointer to the first member of the list, and the member function `end()` returns a pointer to an imaginary list element *beyond* the last member of the list.

```
void computeMoments(int B, int i, int j) {

    // find center of box
```

```

complex<double> zCenter(i + 0.5, j + 0.5);
zCenter *= L / B;

// zero moments
for (int m = 0; m <= M; m++)
    a[m] = 0;

// loop over particles in box
list<int>::const_iterator pos;
for (pos = boxList[i][j].begin(); pos != boxList[i][j].end(); pos++) {
    int n = *pos;
    a[0] += q[n];
    complex<double> zn = z[n] - zCenter;
    complex<double> zk = zn;
    for (int k = 1; k <= M; k++) {
        a[k] += q[n] * zk / k;
        zk *= zn;
    }
}
}

```

The code given above to compute the moments implements the definitions

$$a_0 = \sum_i^{N_c} q_i, \quad a_k = \sum_i^{N_c} \frac{q_i z_i^k}{k}, \quad k = 1, 2, \dots, M$$

Recall that z_i is the position of the charge relative to the center of the box.

Processing the interaction lists

The following function computes the contributions to the potential energy due to a particular box indexed by i, j at a particular level with B boxes in each direction. The charge distribution in the box is approximated by its multipole expansion about the center of the box. At this level, the multipoles interact with all of the charges in the interaction list of the box.

```
void processInteractionLists(int B, int i, int j) {  
  
    // find center of box  
    complex<double> zCenter(i + 0.5, j + 0.5);  
    zCenter *= L / B;
```

Recall that the interaction list consists of all children of the near neighbors of the parent box which are not near neighbors. There are 36 children of the near neighbors of the parent. These are enumerated by finding the base indexes i_0, j_0 of the 6×6 square of children. Boxes which fall outside the system volume are excluded, as are the near neighbors.

```
    // find interaction list  
    int i0 = (i / 2) * 2 - 2;  
    int j0 = (j / 2) * 2 - 2;  
    for (int i1 = i0; i1 < i0 + 6; i1++) {  
        if (i1 < 0 || i1 >= B)                // outside system volume  
            continue;  
        for (int j1 = j0; j1 < j0 + 6; j1++) {  
            if (j1 < 0 || j1 >= B)            // outside system volume  
                continue;  
            // exclude near neighbors  
            if ( (i - i1) * (i - i1) < 2 && (j - j1) * (j - j1) < 2 )  
                continue;
```

For each box in the interaction list we loop over all the charges in the box. The contributions to the potential energy of each charge is computed using the multipole expansion of the potential

$$U(z) = a_0 \ln z - \sum_{k=1}^M \frac{a_k}{z^k},$$

where z is the position of a charge relative to the center of the multipole expansion. Note that the potential energy is got by multiplying the potential due to the multipoles by the charge at position z .

```

// interact moments in box ij with charges in box i1,j1
list<int>::const_iterator pos;
for (pos = boxList[i1][j1].begin();
     pos != boxList[i1][j1].end(); pos++) {
    int n = *pos;
    complex<double> dz = z[n] - zCenter;
    Epot += q[n] * a[0] * log(dz);
    complex<double> dzk = dz;
    for (int k = 1; k <= M; k++) {
        Epot -= q[n] * a[k] / dzk;
        dzk *= dz;
    }
}
}
}
}
}

```

Adding all contributions from the quadtree

The following function adds all contributions to the potential energy. The potential energy variable is initialized to zero. We then loop over levels starting with level 2. At each level, the particles are sorted into box lists. Then all B^2 boxes at this level are visited, the multipole moments due to charges in the box are computed by calling `computeMoments`, and the moments are then interacted with all charges in the interaction list of the box by calling `processInteractionLists`.

```
void computeEpot() {  
  
    // initialize to zero  
    Epot = complex<double>(0, 0);  
  
    // iterate over levels  
    for (int level = 2; level <= levels; level++) {  
  
        sortBodies(level);  
  
        // loop over boxes  
        int B = boxesEachDirection(level);  
        for (int i = 0; i < B; i++)  
            for (int j = 0; j < B; j++) {  
                computeMoments(B, i, j);  
                processInteractionLists(B, i, j);  
            }  
    }  
}
```

Finally, we need to take into account the near neighbor boxes at the finest level of subdivision. Note that the

charges have already been sorted at this level when the loop above completes. The computation of the interactions is straightforward. All boxes i, j at this level are visited. For each box, the 9 near-neighbor boxes k, l are visited. Note that box i, j is a near neighbor of itself, so interactions of particles in each box are also taken into account. For each pair of boxes, the potential energy of each pair of charges is computed using the exact formula

$$q_{n_1} q_{n_2} \ln(z_{n_1} - z_{n_2}) ,$$

where n_1 is in box i, j and n_2 is in box k, l .

```
// near neighbor interactions at deepest level
int B = boxesEachDirection(levels);
for (int i = 0; i < B; i++)
for (int j = 0; j < B; j++) {
    for (int k = i - 1; k <= i + 1; k++)
    for (int l = j - 1; l <= j + 1; l++) {
        if (k < 0 || k >= B || l < 0 || l >= B) // outside system volume
            continue;
        list<int>::const_iterator p1, p2;
        for (p1 = boxList[i][j].begin(); p1 != boxList[i][j].end(); p1++)
        for (p2 = boxList[k][l].begin(); p2 != boxList[k][l].end(); p2++) {
            int n1 = *p1;
            int n2 = *p2;
            if (n1 != n2)
                Epot += q[n1] * q[n2] * log(z[n1] - z[n2]);
        }
    }
}
```

You can convince yourself that the procedures given above count each pair of particles twice. To get the potential energy of the system it is therefore necessary to divide by 2.

```
Epot /= 2;  
}
```

Finall, here is the `main` function:

```
int main(int argc, char *argv[]) {  
  
    if (argc > 1)  
        N = atoi(argv[1]);  
    if (argc > 2)  
        L = atof(argv[2]);  
    cout << "Number of charges = " << N << endl;  
    cout << "Side of square box = " << L << endl;  
  
    initialize();  
    clock_t t0 = clock();  
    computeEpot();  
    clock_t t1 = clock();  
    cout.precision(16);  
    cout << "    Potential energy = " << Epot << endl;  
    cout << "            CPU time = " << double(t1 - t0) / CLOCKS_PER_SEC  
        << " sec" << endl;  
  
}
```

