ECN
MSc Computational Mechanics
Sem 3

Domain Decomposition and Iterative Solver

Lab 2: LU and Cholesky Factorisation, dense storage and symmetric band

Prasad ADHAV

Introduction

In this lab we are meant to apply what was seen during the second lecture: dense and band matrix factorization.

1 Part 1

In this part of the lab, 3 LU factorization must be implemented, without pivoting as seen in the class. To help with this implementation, some codes are provided. From those codes the main code to be executed for part 1 is `lab2_part1.cc`. The functions `factorBasic, factorL2` and `factorL3` are declared in `dLU_denseCM.h`. They were incomplete and are completed accordingly. The matrix are stored in column major format, the first version `dLU1b` uses hand written loop, the second version `dLU2b` uses Level 1 BLASxSCAL and LEVEL2BLAS ger. And the third version `dLU3b` implements the block LU decomposition, using `dLU2b` t factorize block of size r, and Level 3 BLAS dTRSM and dGEMM. The implementations for all three methods are given as follows:

```
int factorBasic(int n,  double *a, int LDA ){
  int k=0;
  double piv=a[0];
  const double minpiv=1e-6;
  while((fabs(piv)>minpiv) && (k<n-1))
  {
      for(int i=k+1; i<n; i++)
      {
          a[i+k*LDA]=a[i+k*LDA]/piv;
      }
      for (int i=k+1; i<n; i++)
      {
          for (int j=k+1; j<n; j++)
          {
              a[i+j*LDA] =a[i+j*LDA]-a[i+k*LDA]*a[k+j*LDA];
          }
      }
      k +=1;
      piv=a[k+k*LDA];
  }
  if(fabs(piv)<=minpiv)
  {
      std::cout<<"Null point in dLU1: "<<piv<<__FILE__<<":"<<__LINE__<<std::endl;
      return 0;
  }
  return 1;
```

```
int factorL2(int n,  double *a, int LDA ){
    int k=0;
    const double minpiv=1e-6;
    double piv =a[0];
    while((fabs(piv)>minpiv) && (k< n-1))
    {
        dscal_(n-(k+1),1./piv, a+(k+1+k*LDA),1);
        dger_(n-(k+1), n-(k+1), -1., a+(k+1+k*LDA), 1, a+(k+(k+1)*LDA), LDA, a+(k+1+(k+1)*LDA), LDA);
        k +=1;
        piv=a[k+k*LDA];
    }
    if (fabs(piv)<=minpiv)
    {
        std::cout<<"Null point in dLU2: "<<piv<<__FILE__<<":"<<__LINE__<<std::endl;
        return 0;
    }
    return 1;
};
```

```
int factorL3(int r, int n,  double *a, int LDA ){
  int l=0;
  while(l<n)
  {
     int m = std::min(n, l+r);
      int bsize = m-l;
      int success=factorL2(bsize, a+(l+l*LDA), LDA);
      if (!success)
      {
          std::cout<<"Can't fatorize one block" <<std::endl;
          return 0;
      }
      dtrsm_('L', 'L', 'N', 'U', bsize, n-m, 1.0, a+(l+l*LDA), LDA, a+l+m*LDA, LDA);
      dtrsm_('R', 'U', 'N', 'N', n-m, bsize, 1.0, a+(l+l*LDA), LDA, a+m+l*LDA, LDA);
      dgemm_('N','N', n-m, n-m, bsize, -1.0, a+m+l*LDA, LDA, a+l+m*LDA, LDA, 1.0, a+m+m*LDA,
          LDA);
      l=m;
  }
  return 1;
}
```

Multiple random matrices are generated of size 100 to 2000 (size increment 190), and time evolution for the three methods implemented is shown in the fig 1. It is clear from the implementation that the second and third implementations are faster than the first implementation, which is expected.

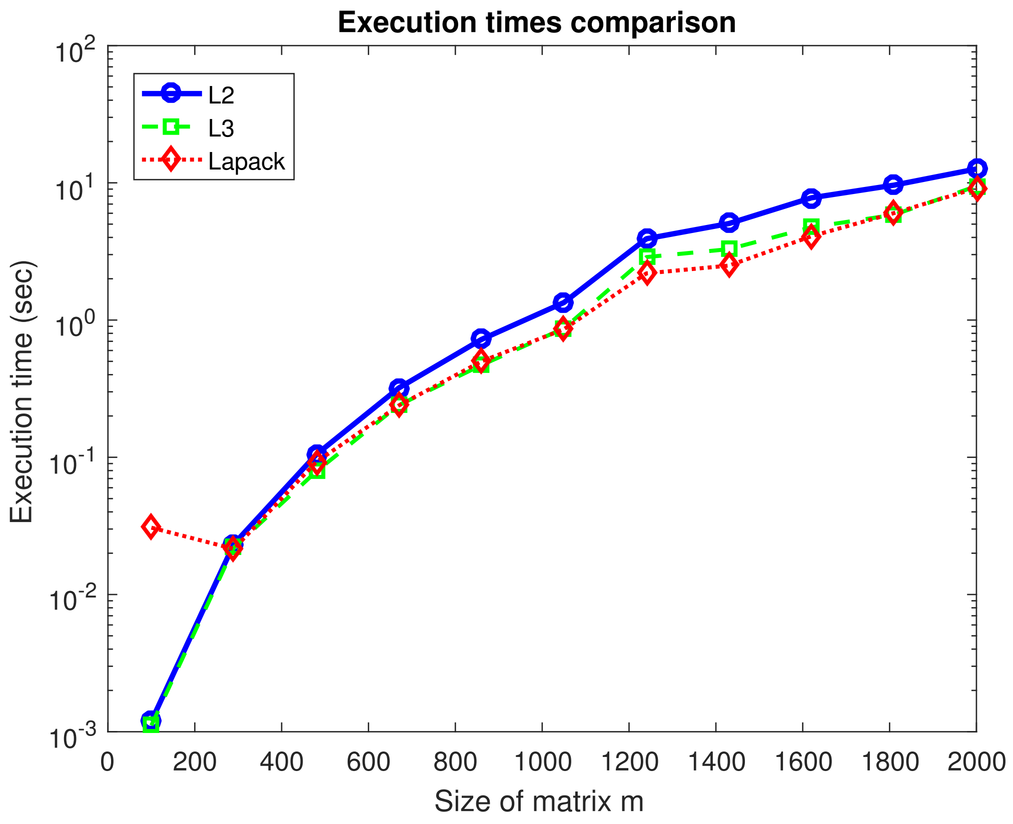

Figure 1: Time evolution for the three implementations

The comparison for all the implementations is given in table 2. And there we can observe the efficiency of the implementation done in the lab.

2 Part 2

In this part we have to test the factorization of positive definite band matrix. Two classes are given: dsquarematrix_symband and dCholesky_band. The first class is meant to store and manipulate a band matrix, the second one compute and store the Cholesky factorization of a matrix in symmetric band storage, using functions of the lapack library. The function computeBandWidthUp and computeBandWidthdown is to be implemented in the dmatrix_denseCM.cc file. These functions are meant to return the number of non-zero upper and lower diagonal respectivelly, taking as input a matrix in denseCM format. These values are used to construct a band storage of the matrix. The implementations for these two are done as follows:
`computeBandwidthUp`

```
dmatrix_denseCM operator*(const dmatrix_denseCM &A, const dmatrix_denseCM &B){
  return muldgemm(A,B);
}

int computeBandwidthUp(const dmatrix_denseCM &A ){
    int start_i=0, max_i=0, min_j=0, m, UpperBandwidth=0;
    double sum=0.0, sumBuffer=0.0;
    m=A.getNbLines();
    int ceilrow=int(std::ceil(m));
    for (int j=0; j<ceilrow; j++)
    {
        sum=0.0;
        sumBuffer=0.0;
        int Test;
        for (int i=start_i; i<m; i++)
        {
            sum=sum+A(i,j);
            Test=(sum>sumBuffer);
            max_i=Test*i+!Test*max_i;
            min_j=Test*j+!Test*min_j;
            sumBuffer=sum;
            start_i=max_i;
        }
        start_i++;
        UpperBandwidth=std::max(UpperBandwidth, max_i-min_j);
    }
    std::cout<<"Calculated Upper Bandwith is: "<<UpperBandwidth<<"\n";
    return UpperBandwidth;
};
```

`computeBandwidthDown`

```
int  computeBandwidthDown(const dmatrix_denseCM &A ){
    int start_j=0, max_j=0, min_i=0, n, LowerBandwidth=0;
    double sum=0.0, sumBuffer=0.0;
    n=A.getNbColumns();
    int ceilcolumn=int(std::ceil(n));
    for (int i=0; i<ceilcolumn; i++)
    {
        sum=0.0;
        sumBuffer=0.0;
        int Test;
        for (int j=start_j; j<n; j++)
        {
            sum=sum+A(i,j);
            Test=(sum>sumBuffer);
            max_j=Test*j+!Test*max_j;
            min_i=Test*i+!Test*min_i;
            sumBuffer=sum;
            start_j=max_j;
        }
        start_j++;
        LowerBandwidth=std::max(LowerBandwidth, max_j-min_i);
    }
    //UpperBandwidth=max_i-min_j;
    std::cout<<"Calculated Lower Bandwith is: "<<LowerBandwidth<<"\n";
    return LowerBandwidth;
};
```

In the class dsquarematrix_symband, we need to implement the operator()(int i, int j) that return the i,j term of a matrix, stored using the symmetric band storage. Note that this function is used when constructing a band matrix. Once this 3 functions are implemented correctly, the program in lab2_part2.cc should work properly, and you should see what can be gained using band storage. The implementation for the same is given as

```
double& dsquarematrix_symband::operator()(int i, int j) {
    int formula=(lb-j+i)+j*(lb+1);
    return *(a+formula);
}
```

The results for different matrices for the second implementation can be seen as follows:

	Size	Factorisation time	Time	lbu	lbd	Error
data_band.mat	5	0.000232438	$2.937e-05$	2	2	0
bcsstk14.mtx	1806	0.531188	0.0121683	161	161	$1.25193e-11$
bcsstk15.mtx	3948	6.2573	0.130958	437	437	$7.14698e-10$

Table 1: The results for different matrices for Part 2

Conclusion

The difference of performance for the different implementations can be observed in the following table. The time and error for `bcsstk14.mtx` is give in 2

Factorisation Policy	Time for factorization	Time	Error
Basic (Part 1)	$7.48e-07$	150.731	$2.23514e-11$
L2 (Part 1)	$5.37e-07$	7.70804	$2.15511e-11$
L3-L2 (Part 1)	$4.96e-07$	5.56005	$1.41449e-11$
Lapack (Part 1)	$4.6e-07$	3.82515	$1.14181e-15$
Lapack Part 2	0.531188	0.0121683	$6.07675e-17$

Table 2: Time and error comparison for different implementations

From the above table it is apparent that the implementation in part 2 is much more efficient that the implementations done in part 1. Although we see Lapack in Part 1, the factorization is done in form of lower and upper triangular matrix, which is slower than the band wise factorization implemented in part 2. Hence we can say that the symmetric band factorization is much more efficient and thus faster than all the other implementation seen in this lab.