Discussion on Direct Solvers

TMA4280—Introduction to Supercomputing

Based on 2016v slides by Eivind Fonn

NTNU, IMF

February 27. 2017

The problem

— We want to solve

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \qquad \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^N, \qquad \boldsymbol{A} \in \mathbb{R}^{N \times N}$$

where $\boldsymbol{A}$ is the system resulting from discretiziing a Poisson problem using finite differences (see the previous slides).

— We use standard notation for matrices and vectors, i.e.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

Computer algorithms

— We know that the solution is given by

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}.$$

— Explicitly forming the inverse $\boldsymbol{A}^{-1}$ is expensive, prone to round-off errors, and something we seldom do on computers.

— Exception: Very small and frequently re-used sub-problems that are known to be well conditioned.

— Matrix inversion has the same time complexity as matrix multiplication (typically $\mathcal{O}(n^3)$).

— Instead, we implement algorithms that solve a linear system given a specific right-hand-side vector.

— The structure and properties of the matrix $\boldsymbol{A}$ determine which algorithms we can use.

Computer algorithms

— For example: if $\boldsymbol{A}$ is known to be orthogonal, then

$$\boldsymbol{x} = \boldsymbol{A}^{\mathsf{T}}\boldsymbol{b}.$$

— Orthogonal matrices are the exception, and not the rule.

— We are more likely to find and exploit properties such as

- Symmetry
- Definiteness
- Sparsity
- Bandedness

— We will now consider a number of different algorithms, their implementation and usability in a parallel context.

All the methods rely on intensive use of basic linear algebra operations which should be optmized for performance.

Complexity of operations

Dot Product:

$$\alpha = \sum_i \boldsymbol{x}_i \boldsymbol{y}_i$$

The reduction incurs a cost of $log2(N)$ for $N/2$ processors.

Scalar-Vector Product:

$$\boldsymbol{y}_i = \alpha \boldsymbol{x}_i$$

The operation parallelizes completely and has a constant complexity using N processors

Complexity of operations

Vector Addition:

$$\boldsymbol{z}_i = \boldsymbol{x}_i + \boldsymbol{y}_i$$

The operation parallelizes completely and has a constant complexity using N processors. Also note the provided operation *AXPY*: $\boldsymbol{y}_i = \alpha \boldsymbol{x}_i + \boldsymbol{y}_i$.

Sum of N Vectors of size M:

$$\boldsymbol{y}_i = \sum_k \boldsymbol{x}_{k,i}$$

The operation is a reduction for eah component and each component can be done in parallel. complexity is $log2(N)$ on $MN/2$ processors

Sum of N Matrices of $\mathbb{R}^{M \times M}$: Simple extension of the vector case.

Complexity of operations

Matrix-Vector Multiplication in $\mathbb{R}^{M\times N}$:

$$\boldsymbol{y}_i = \boldsymbol{A}_{ij}\ \boldsymbol{x}_j$$

The operation can be seem as M inner-products of size N. Two options are available depending on the loop ordering:

1. i, j: innner-product model
2. j, i: linear combination of column vectors

The use of one or the other depends on the data layout of 2d arrays:

1. row-major: storage row by row (C/C++)
2. column-major: storage column by column (FORTRAN)

Complexity is bound to the reduction $log2(N)$ for $MN/2$ processors.

For M processors it is $O(N)$, ror 1 processor it is $O(MN)$

Complexity of operations

Matrix-matrix Multiplication in $\mathbb{R}^{M\times N}$, $\mathbb{R}^{N\times L}$:

$$\boldsymbol{A}_{ij} = \boldsymbol{B}_{ik}\ \boldsymbol{C}_{kj}$$

Six options are available depending on the loop ordering i, j, k.

Complexity is bound to the reduction $log2(N)$ for $MNL/2$ processors.

For square matrices: N^3 processors it is $O(log2(N))$, for N^2 processor it is $O(N)$, and for N processor it is $O(N^2)$

Linear algebra packages

BLAS	Linear algebra routines, reference (FORTRAN)
OpenBLAS	Optimized multithreaded BLAS (C, Assembly)
LAPACK	Direct and eigenvalue solvers (FORTRAN)
Armadillo	Linear algebra package (C++)
SuiteSparse	Factorizations with GPU support
MUMPS	Parallel sparse matrix direct solvers (FORTRAN)

Software packages like PETSc, Trilinos provde general interfaces to these packages.

Linear algebra performance

Performance is measured with the number of floating-point operations per second (FLOPS).

Performance of linear algebra operations is heavily influenced by:

1. ratio between computations and data movement,
2. memory access patterns.

Performance can be further improved by parallelism: some algorithms offer intrisically more opportunities for parallelism than other (think in terms of data dependencies)

Linear algebra performance

Vector-vector operations (AXPY):

1. $2n$ ops
2. $2n$ data

$\rightarrow r_{ops/data} = O(1)$ performance guided by ILP and caching limits.

Matrix-matrix operations (MM): $A \in \mathbb{R}^{mxk}, B \in \mathbb{R}^{kxn}$

1. $mn(2k-1)$ ops
2. $(m+n)k$ data

$\rightarrow r_{ops/data} = O(N)$ data reuse by several floating-point operations.

Such considerations were explained when introducing computational efficiency of dense linear algebra benchmarks (LINPACK) vs. sparse matrix benchmarks (HPCG)

BLAS

— A very helpful library here is *BLAS*: Basic Linear Algebra Subprograms.

— BLAS is an old specification from the late seventies and early eighties.

— It consist of a collection of functions with strangely cryptic and short names.

— Can be installed on Ubuntu with `sudo apt-get install libblas-dev`.

— On Vilje, Intel's implementation of BLAS is available under *MKL*: the Math Kernel Library, and on Lille OpenBLAS is available.

BLAS levels

— BLAS functions are organized by *level*.

— Level 1: vector-vector operations.

$$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$$

```
daxpy(n, alpha, y, 1, x, 1)
```

— Level 2: matrix-vector operations.

$$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$$

```
dgemv('N', m, n, alpha, A, m, x, 1, beta, y, 1)
```

— Level 3: matrix-matrix operations

$$\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$$

```
dgemm('N', 'N', m, n, k, alpha, A, m, B, k, C, m)
```

BLAS conventions

- All functions in BLAS starts with one of the letters
  - s for *single*.
  - d for *double*.
  - c for *single complex*.
  - z for *double complex*.
- If the operation involves a matrix, two letters describing the matrix format follow. The most important of these are
  - ge for *general* matrices.
  - po for *symmetric* matrices.
  - gb for *general banded* matrices.
  - pb for *symmetric banded* matrices.

BLAS

- The BLAS home page can be found at http://netlib.org/blas/
- BLAS is written in Fortran and therefore expects Fortran memory layout (column-major ordering).
- For C, the *CBLAS* implementation is popular. CBLAS supports both row- and column-major orders.

Serial BLAS

```
void MxV(double *u, double *A, double *v, int N)
{
  dgemv('N', N, N, 1.0, A, N, v, 1, 0.0, u, 1);
}

double innerproduct(double *u, double *v, int N)
{
  // Necessary adjustments must be made for MPI
  return ddot(N, u, 1, v, 1);
}
```

Simple example

Computing

$$\alpha = \sum_{i=1}^{K} \boldsymbol{v}_i^{\mathsf{T}} \boldsymbol{A} \boldsymbol{v}_i$$

where $\boldsymbol{v} \in \mathbb{R}^{N \times K}$ and $\boldsymbol{A} \in \mathbb{R}^{N \times N}$. There are K terms, each of which require us to compute a matrix-vector product and an inner product.

Serial

```
void MxV(double *u, double **A, double *v, int N)
{
  for (size_t i = 0; i < N; i++) {
    u[i] = 0.0;
    for (size_t j = 0; j < N; j++)
      u[i] += A[i][j] * v[j];
  }
}

double innerproduct(double *u, double *v, int N)
{
  double result = 0.0;
  for (size_t i = 0; i < N; i++)
    result += u[i] * v[i];
  return result;
}
```

Serial

```
double dosum(double **A, double **v, int K, int N)
{
  double alpha = 0.0, temp[N];
  for (size_t i = 0; i < K; i++) {
    MxV(temp, A, v[i], N);
    alpha += innerproduct(temp, v[i], N);
  }

  return alpha;
}
```

OpenMP micro-version

— It is tempting to exploit all parallelism in sight. However, don't do that.
— Let us use OpenMP for micro-parallelism. That is, we exploit parallelism within the inner product and the matrix-vector operation.
— That means two fork/join operations per term, so $2K$ in total.

OpenMP micro-version

```
void MxV(double *u, double **A, double *v, int N)
{
  #pragma omp parallel for schedule(static)
  for (size_t i = 0; i < N; i++) {
    u[i] = 0.0;
    for (size_t j = 0; j < N; j++)
      u[i] += A[i][j] * v[j];
  }
}
```

OpenMP micro-version

```
double innerproduct(double *u, double *v, int N)
{
  double result = 0.0;
  #pragma omp parallel for schedule(static) \
          reduction(+:result)
  for (size_t i = 0; i < N; i++)
    result += u[i] * v[i];
  return result;
}
```

OpenMP macro-version

— The alternative is to exploit the coarsest parallelism: each iteration in the `dosum` method.

— In this case we perform exactly one fork and one join.

— Problem: the `dosum` method uses a temporary buffer for the matrix-vector multiplication, which cannot be shared between threads. We have to use a separate buffer for each thread.

OpenMP macro-version

```
double dosum(double **A, double **v, int K, int N)
{
  double alpha = 0.0;
  double **temp = createMatrix(K, N);
  #pragma omp parallel for schedule(static) \
          reduction(+:alpha)
  for (size_t i = 0; i < K; i++) {
    MxV(temp[i], A, v[i], N);
    alpha += innerproduct(temp[i], v[i], N);
  }

  return alpha;
}
```

MPI macro-version

```
double dosumMPI(double **A, double **v,
                int myK, int N)
{
  double myalpha = dosum(A, V, myK, N);
  double alpha;
  MPI_Allreduce(&myalpha, &alpha, 1, MPI_DOUBLE,
                MPI_SUM, MPI_COMM_WORLD);
  return alpha;
}
```

In addition to the usual MPI code, you have to decide on a particular division of the work among the nodes. For brevity's sake, it is left out in this example.

Speedup results

Table: $N = 2048$, $K = 1024$

Threads	Micro	Macro	MPI
1	1.00	1.00	1.00
2	1.84	1.83	1.56
4	2.79	2.76	3.46

Table: $N = 16$, $K = 32768$

Threads	Micro	Macro	MPI
1	1.00	1.00	1.00
2	0.50	2.00	2.00
4	0.33	3.49	4.00

Timing results

Table: $N = 2048$, $K = 1024$

Threads	Macro	w/ BLAS	MPI	w/ BLAS
1	35.20	2.06	35.27	2.05
2	17.68	1.06	18.73	1.17
4	9.08	0.66	9.15	0.61
8	4.54	0.36	4.82	0.32

Timing results

Table: $N = 16$, $K = 32768$ (milliseconds)

Threads	Macro	w/ BLAS	MPI	w/ BLAS
1	9.44	9.10	10.71	9.36
2	20.08	24.31	7.62	4.48
4	15.20	28.78	6.20	6.23
8	7.36	23.89	5.58	4.68

General matrices

— In introductory linear algebra we learned two ways to invert general matrices.

— First is Cramer's rule. The solution to a linear system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

can be found by repeated application of

$$x_i = \frac{\det \boldsymbol{A}_i}{\det \boldsymbol{A}}$$

where $\boldsymbol{A}_i$ is $\boldsymbol{A}$ with the ith column replaced with $\boldsymbol{b}$.

— Naive implementations scale as $N!$, which makes Cramer's rule useless.

General matrices

— Instead we tend to resort to Gaussian elimination.

— A systematic procedure that allows us to transform the matrix $\boldsymbol{A}$ into triangular form, which allows for easy inversion using backward substitution.

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{pmatrix}$$

— The last equation is trivial. Solve that, plug the value in the next-to-last equation, which makes *that* trivial, etc.

General matrices

— The equations are on the form

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 \quad &+ \ldots + a_{1,N}x_N \quad = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 \quad &+ \ldots + a_{2,N}x_N \quad = b_2 \\ \vdots \qquad\qquad\qquad\qquad\qquad\qquad &= \quad \vdots \end{aligned}$$

— We want to get rid of the term $a_{2,1}x_1$. We do this by applying the row-operation (row 2) $- a_{2,1}/a_{1,1}$ (row 1).

— This yields in the second row

$$0 + \left(a_{2,2} - \frac{a_{2,1}}{a_{1,1}}a_{1,2}\right)x_2 + \ldots + \left(a_{2,N} - \frac{a_{2,1}}{a_{1,1}}a_{1,N}\right)x_N = b_2 - \frac{a_{2,1}}{a_{1,1}}b_1$$

General matrices

— We repeat this for all rows and all columns beneath the diagonal.

— Note: we need $a_{k,k} \neq 0$ when eliminating column k. If that's not the case, we have to exchange two rows. This is called *pivoting*.

— To limit cancellation errors due to limited precision, we also want $a_{k,k} \gg 0$. Therefore, choose the row with the largest element. This is called *partial pivoting*. (Full pivoting also interchanges columns.)

— This is a procedure with relatively simple rules, which makes it suitable for implementation.

General matrices

There are two problems with Gaussian elimination.

- It modifies the right-hand-side vector $\boldsymbol{b}$. If we want to solve the same linear system with a different $\boldsymbol{b}$ we have to redo the whole procedure.
- It is still prone to round-off errors, even with partial pivoting.
- Therefore, the typical implementation of Gaussian eliminiation is in terms of the LU decomposition: we seek two matrices $\boldsymbol{L}$ and $\boldsymbol{U}$, lower and upper triangular, such that $\boldsymbol{A} = \boldsymbol{LU}$.
- This decomposition is independent of $\boldsymbol{b}$.

General matrices

— We can then find the solution to a system

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{L}\boldsymbol{U}\,\boldsymbol{x} = \boldsymbol{b}$$

in this way.

— First, solve $\boldsymbol{L}\,\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$. (Forward substitution.)

— Then, solve $\boldsymbol{U}\,\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$. (Backward substitution.)

— Forward substitution works the same way as backward substitution. Just backward. . . in other words, forward.

General matrices

— The LU decomposition has redundancy: there are two sets of diagonal elements. There are two popular implementations: Doolittle's method (unit diagonal on $\boldsymbol{L}$) and Crout's method (unit diagonal on $\boldsymbol{U}$).

— Doolittle's algorithm can be stated briefly as

$$u_{1,k} = a_{1,k} \qquad k = 1, \ldots, N$$

$$\ell_{j,1} = \frac{a_{j,1}}{u_{1,1}} \qquad j = 2, \ldots, N$$

$$u_{j,k} = a_{j,k} - \sum_{s=1}^{j-1} \ell_{j,s} u_{s,k} \qquad k = j, \ldots, N$$

$$\ell_{j,k} = \frac{1}{u_{k,k}} \left(a_{j,k} - \sum_{s=1}^{k-1} \ell_{j,s} u_{s,k} \right) \qquad j = k+1, \ldots, N$$

LAPACK

- LU decomposition is somewhat tedious to implement, particularly in an efficient way.
- Smart people have done it for you.
- *LAPACK*: The Linear Algebra Pack, which builds on BLAS.
- Same naming conventions and matrix formats.
- Just like with BLAS and CBLAS there is a LAPACK and a CLAPACK (and LAPACKE).
- Here we will stick to LAPACK (Fortran numbering).

LAPACK

Function prototype:

```
void dgesv(const int *n, const int *rhs,
           double *A, const int *lda, int *ipiv,
           double *B, const int *ldb, int *info);
```

Usage:

```
void lusolve(Matrix A, Vector x)
{
  int *ipiv = malloc(x->len * sizeof(int));
  int one = 1, info;
  dgesv(&x->len, &one, A->data[0], &x->len,
        ipiv, x->data, &x->len, &info);
  free(ipiv);
}
```

LAPACK

— `dgesv` overwrites the matrix $\boldsymbol{A}$ with the factorization $\boldsymbol{L}$, $\boldsymbol{U}$.

— `dgesv` actually calls two functions:

- `dgetrf` to compute the decomposition,
- `dgetrs` to solve the system.

— Therefore we can solve for different right-hand-sides after the first call, by calling `dgetrs` ourselves.

— It would be a mistake to call `dgesv` more than once!

Improving performance

— While LAPACK is efficient, it cannot do wonders.

— LU decomposition has the same complexity as matrix multiplication: $\mathcal{O}(N^3)$. It is *asymptotically* as bad as matrix inversion (although in realistic terms much better).

— Even if $\boldsymbol{A}$ is sparse, in general $\boldsymbol{L}$ and $\boldsymbol{U}$ aren't. (Although for banded matrices they actually are.)

LU

The constructive proof of existence give an algorithm to compute the factorization:

$$\boldsymbol{L}^{(p)} = \begin{pmatrix} \boldsymbol{L}^{(p-1)} & \boldsymbol{0} \\ \boldsymbol{G}^t & 1 \end{pmatrix}$$

$$\boldsymbol{U}^{(p)} = \begin{pmatrix} \boldsymbol{U}^{(p-1)} & \boldsymbol{H} \\ \boldsymbol{G}^t & 1 \end{pmatrix}$$

$$\boldsymbol{A}^{(p)} = \begin{pmatrix} \boldsymbol{A}^{(p-1)} & \boldsymbol{S} \\ \boldsymbol{S}^t & a_{pp} \end{pmatrix}$$

$$\boldsymbol{A}^{(p)} = \boldsymbol{L}^{(p)}\ \boldsymbol{U}^{(p)}$$

LU

The operations involved are:

$$\boldsymbol{A}^{(p-1)} = \boldsymbol{L}^{(p-1)}\ \boldsymbol{U}^{(p-1)}$$

which is known.

$$\boldsymbol{L}^{(p-1)}\ \boldsymbol{H} = \boldsymbol{R}$$

$$\boldsymbol{U}^{(p-1)T}\ \boldsymbol{G} = \boldsymbol{R}$$

$$u_{pp} = a_{pp} - \boldsymbol{G}^{T}\ \boldsymbol{H}$$

This constructive approach has complexity of:

$$O\left(\frac{2}{3}\ N^3\right) + O(N^2)$$

which is the same order as a matrix-matrix multiplication.

LU

Let us write the decomposition to compute the determinant of $\boldsymbol{M}$

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{pmatrix}$$

$$\det(\boldsymbol{M}) = \det(\boldsymbol{A}) \det(\underbrace{\boldsymbol{D} - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{B}}_{\text{Complement of } \boldsymbol{A} \text{ in } \boldsymbol{M}})$$

With the decomposition written as:

$$\boldsymbol{L} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{C}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix} \qquad \boldsymbol{U} = \begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{0} & \boldsymbol{D} - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{B} \end{pmatrix}$$

In this case the inverson can be performed using matrix-matrix operations.

Symmetry

- If $\boldsymbol{A}$ is symmetric and positive definite, we find that $\boldsymbol{U} = \boldsymbol{L}^{\mathsf{T}}$.
- Thus we can save a factor of two in memory and in floating point operations.
- No pivoting is required for such systems.
- This is termed *Cholesky factorization*.
- Note that all conditions are satisfied by the Poisson matrices.

LAPACK

Function prototype:

```
void dposv(char *uplo, const int *n,
           const int *nrhs, double *A,
           const int *lda, double *B,
           const int *ldb, int *info);
```

Usage:

```
void llsolve(Matrix A, Vector x)
{
  int one = 1, info;
  char uplo = 'L';
  dposv(&uplo, &x->len, &one, A->data[0],
        &x->len, x->data, &x->len, &info);
}
```

Improving performance

— LU decomposition is unfit for parallel implementation
  - It is sequential in nature (you must eliminate for row 2 before row 3).
  - Pivoting requires synchronization for every row.
  - The substitution phase is completely sequential (but this is not so important, since it is $\mathcal{O}(N^2)$).
  - Smart people have tried to make it work by identifying independent blocks in the matrix.

— For most systems the results are mediocre and with limited scalability.

— $\Rightarrow$ LU decomposition is unusable in a parallel context.

— However, it is useful as a component in other algorithms.

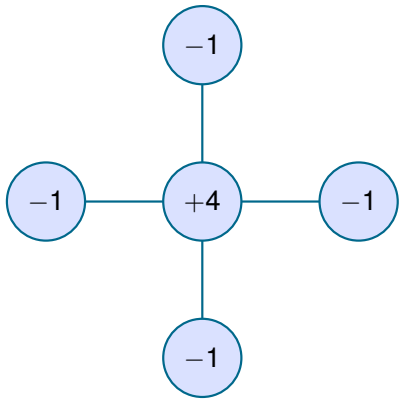
Better approaches

- Solution methods can be categorized in two classes.
- *Direct methods* yield the solution in a predictable number of operations. Cramer's rule, Gaussian elimination and LU decomposition are good examples of direct methods.
- *Iterative methods* converge to the solution through some iterative procedure with an unpredictable number of operations.
- Let us now consider another example of a direct method.

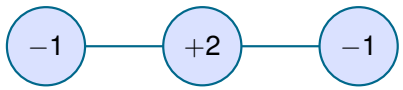
Diagonalization

— This is not a hammer. We must exploit known properties about the matrix $\boldsymbol{A}$.

— We recall the five-point stencil:



— This results from a double application of the three-point stencil in two directions.



Diagonalization

— In the following, the matrix $\boldsymbol{A}$ results from the five-point stencil (2D problem), while the matrix $\boldsymbol{T}$ results from the three-point stencil ($\boldsymbol{T}$).

— For a symmetric positive definite matrix $\boldsymbol{C}$, we know that we can perform an eigendecomposition

$$\boldsymbol{C}\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{\Lambda}$$

where $\boldsymbol{Q}$ is the matrix of eigenvectors (as columns) and $\boldsymbol{\Lambda}$ is the diagonal matrix of eigenvalues.

— Since $\boldsymbol{C}$ is SPD, $\boldsymbol{Q}$ is orthogonal, so

$$\boldsymbol{C} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}}.$$

Diagonalization

— Using this in the linear system, we get

$$\boldsymbol{C}\boldsymbol{x} = \boldsymbol{b} \quad \Longrightarrow \quad \boldsymbol{Q}\Lambda\boldsymbol{Q}^{\mathsf{T}} = \boldsymbol{b}.$$

— Multiply from the left by $\boldsymbol{Q}^{\mathsf{T}}$, recalling that $\boldsymbol{Q}^{\mathsf{T}} = \boldsymbol{Q}^{-1}$.

$$\Lambda \underbrace{\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{x}}_{\tilde{\boldsymbol{x}}} = \underbrace{\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{b}}_{\tilde{\boldsymbol{b}}}.$$

— This reduces the problem to three relatively easy steps.

Diagonalization

1. Calculate $\tilde{\boldsymbol{b}}$ with a matrix-vector product

$$\tilde{\boldsymbol{b}} = \boldsymbol{Q}^{\mathsf{T}} \boldsymbol{b}$$

   in $\mathcal{O}(N^2)$ operations.
2. Solve the system

$$\boldsymbol{\Lambda} \tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$$

   in $\mathcal{O}(N)$ operations ($\boldsymbol{\Lambda}$ is diagonal).
3. Calculate $\boldsymbol{x}$ with another matrix-vector product

$$\boldsymbol{x} = \boldsymbol{Q} \tilde{\boldsymbol{x}}$$

   in $\mathcal{O}(N^2)$ operations.

Diagonalization

— This looks promising: it seems we have found a way to solve the system in $\mathcal{O}(N^2)$ operations instead of $\mathcal{O}(N^3)$!

— Unfortunately this is not true, as the computation of the eigendecomposition itself ($\boldsymbol{Q}$ and $\boldsymbol{\Lambda}$) requires $\mathcal{O}(N^3)$ operations.

— However, in certain cases we can still exploit this method.

Diagonalization

— We constructed the matrix $\boldsymbol{A}$ by applying the three-point stencil in two directions.

— In the language of linear algebra, this translates to a tensor product.

$$\boldsymbol{A} = \boldsymbol{I} \otimes \boldsymbol{T} + \boldsymbol{T} \otimes \boldsymbol{I}$$

— The linear system

$$(\boldsymbol{I} \otimes \boldsymbol{T} + \boldsymbol{T} \otimes \boldsymbol{I})\, \boldsymbol{x} = \boldsymbol{b}$$

in a global numbering scheme, can equivalently be stated

$$\boldsymbol{T}\boldsymbol{X} + \boldsymbol{X}\boldsymbol{T}^{\mathsf{T}} = \boldsymbol{B}$$

in a local numbering scheme. Here, the unknown $\boldsymbol{X}$ and the right-hand-side $\boldsymbol{B}$ are *matrices*.

Diagonalization

- A global numbering scheme is a scheme where we number the unknowns using a single index. This naturally maps to a vector.
- A local numbering scheme is a scheme where we number the unknowns using one index for each spatial direction. This naturally maps to a matrix (in 2D) or more generally a tensor.
- Thus, we consider in this case a system of matrix equations.
- Alternative way of thinking about it: we operate with $\boldsymbol{T}$ along the columns of $\boldsymbol{X}$, and then along the rows.

$$\underbrace{\boldsymbol{TX}}_{\text{columns}} + \underbrace{(\boldsymbol{TX}^\intercal)^\intercal}_{\text{rows}} = \boldsymbol{TX} + \boldsymbol{XT}^\intercal$$

Diagonalization

— Now let us diagonalize $\boldsymbol{T}$, recalling that it is SPD.

$$\boldsymbol{T}\boldsymbol{X} + \boldsymbol{X}\boldsymbol{T}^{\mathsf{T}} = \boldsymbol{B} \Rightarrow$$
$$\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{X} + \boldsymbol{X}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}} = \boldsymbol{B}.$$

— Multiply with $\boldsymbol{Q}$ from the right,

$$\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{X}\boldsymbol{Q} + \boldsymbol{X}\boldsymbol{Q}\boldsymbol{\Lambda} = \boldsymbol{B}\boldsymbol{Q}$$

— and by $\boldsymbol{Q}^{\mathsf{T}}$ from the left,

$$\boldsymbol{\Lambda}\underbrace{\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{X}\boldsymbol{Q}}_{\tilde{\boldsymbol{X}}} + \underbrace{\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{X}\boldsymbol{Q}}_{\tilde{\boldsymbol{X}}}\boldsymbol{\Lambda} = \underbrace{\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{B}\boldsymbol{Q}}_{\tilde{\boldsymbol{B}}},$$

or

$$\boldsymbol{\Lambda}\tilde{\boldsymbol{X}} + \tilde{\boldsymbol{X}}\boldsymbol{\Lambda} = \tilde{\boldsymbol{B}}.$$

Diagonalization

We find the solution in three steps

1. Calculate $\tilde{\boldsymbol{B}}$ with two matrix-matrix products

$$\tilde{\boldsymbol{B}} = \boldsymbol{Q}^{\mathsf{T}} \boldsymbol{B} \boldsymbol{Q}$$

   in $\mathcal{O}(n^3)$ operations.

2. Solve the system

$$\tilde{x}_{i,j} = \frac{\tilde{b}_{i,j}}{\lambda_i + \lambda_j}$$

   in $\mathcal{O}(n^2)$ operations.

3. Recover the solution with two matrix-matrix products

$$\boldsymbol{X} = \boldsymbol{Q} \tilde{\boldsymbol{X}} \boldsymbol{Q}^{\mathsf{T}}$$

   in $\mathcal{O}(n^3)$ operations.

Diagonalization

- Note that $n = \sqrt{N}$ in two dimensions.
- That gives $\mathcal{O}(N^{3/2})$ operations in total, and a space complexity of $\mathcal{O}(N)$. A very fast algorithm!
- Computing the eigendecomposition is also $\mathcal{O}(N^{3/2})$. Since the matrix is smaller, this no longer dominates the running time.
- It is parallelizable: a few large, global data exchanges are needed (transposing matrices). Multiplying matrices can be done with relatively little communication. (More on this in the big project.)
- LU decomposition would require $\mathcal{O}(N^3)$ operations and $\mathcal{O}(N^2)$ storage. Bandewd LU decomposition would require $\mathcal{O}(N^2)$ operations and $\mathcal{O}(N^{3/2})$ storage.

Diagonalization

— The diagonalization method is quite general, applicable to any SPD system with a tensor-product operator.

— We used Poisson to make it more easily understandable (I hope).

— It turns out that we can do even better by exploiting even more structure in the Poisson problem.

Diagonalization

— The continuous eigenvalue problem

$$-u_{xx} = \lambda u, \qquad \text{in } \Omega = (0,1),$$
$$u(0) = u(1) = 0$$

has solutions for $j = 1, 2, \ldots$

$$\overline{u}_j(x) = \sin(j\pi x),$$
$$\overline{\lambda}_j = j^2\pi^2$$

— We now consider operating with $\boldsymbol{T}$ on vectors consisting of u_j sampled at the gridpoints, i.e.

$$\overline{\boldsymbol{q}}_i^{(j)} = \overline{u}_j(x_i) = \sin\left(\frac{ij\pi}{n}\right)$$

Diagonalization

— This yields

$$(\boldsymbol{T}\overline{\boldsymbol{q}}^{(j)})_i = \underbrace{2\left(1-\cos\left(\frac{j\pi}{n}\right)\right)}_{\lambda_j}\underbrace{\sin\left(\frac{ij\pi}{n}\right)}_{\overline{\boldsymbol{q}}_i^{(j)}}$$

— Thus, the eigenvectors of $\boldsymbol{T}$ are the eigenfunctions of $-\partial^2/\partial x^2$ sampled at the gridpoints. (Lucky!)

— Let us normalize to obtain the matrix $\boldsymbol{Q}$.

$$\boldsymbol{q}_j^{\mathsf{T}}\boldsymbol{q}_j = 1 \Rightarrow \boldsymbol{Q}_{i,j} = \sqrt{\frac{2}{n}}\sin\left(\frac{ij\pi}{n}\right)$$

$$\boldsymbol{\Lambda}_{i,i} = 2\left(1-\cos\left(\frac{j\pi}{n}\right)\right)$$

Discrete Fourier Transform

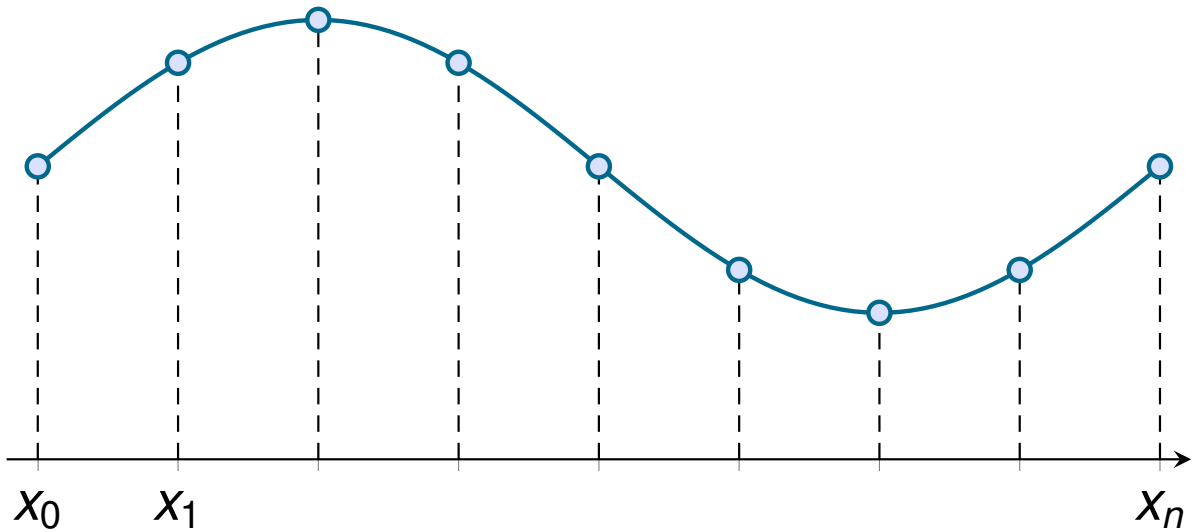
— Now for something completely different. (But not really. It's a magic trick.)

— Consider a periodic function $v(x)$ with period 2π.

— Sample this function at equidistant points x_j, $j = 0, 1, \ldots, n$.

— As with the finite difference grid, $x_j = jh$ where $h = {}^{2\pi}/_{N}$.

— We name the samples $v_j = v(x_j)$.



Discrete Fourier Transform

— Consider the vectors φ_k, where

$$(\varphi_k)_j = \mathrm{e}^{\mathrm{i}kx_j},$$

where

$$\mathrm{e}^{\mathrm{i}kx_j} = \cos(kx_j) + \mathrm{i}\sin(kx_j).$$

— These vectors form a basis for the complex n-dimensional space $\mathbb{C}^n$. In particular, they are orthogonal:

$$\varphi_k^{\mathsf{H}}\varphi_\ell = \begin{cases} n, & k = \ell, \\ 0, & k \neq \ell, \end{cases} \qquad k, \ell = 0, 1, \ldots, n-1.$$

Discrete Fourier Transform

— Since they form a basis, any vector in the space can be expressed as a linear combination of these vectors.

— The vector

$$\boldsymbol{y} = \begin{pmatrix} v_0 & \cdots & v_{n-1} \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^n$$

can be expressed as

$$\boldsymbol{y} = \sum_{k=0}^{n-1} \hat{v}_k \varphi_k,$$

or element by element,

$$v_j = \sum_{k=0}^{n-1} \hat{v}_k (\varphi_k)_j = \sum_{k=0}^{n-1} \hat{v}_k \mathrm{e}^{\mathrm{i}kx_j}.$$

Discrete Fourier Transform

— Here $\hat{v}_k$ are the discrete Fourier coefficients, which are given by the inverse relation. (Since φ_k) are orthogonal, this is easy to compute.)

$$\hat{v}_k = \frac{1}{n} \sum_{j=0}^{n} v_j e^{ikx_j}.$$

— The Fourier transform is extremely useful and has been extensively studied.

— It is useful in, among other things, signal analysis, audio and video compression.

— Important property: the DFT coefficients of an odd, real signal are purely imaginary.

Discrete Sine Transform

— A transform closely related to DFT is the discrete sine transform, the DST.

— It applies to a function $v(x)$, periodic with period 2π, and *odd*. That is, $v(x) = -v(-x)$

— We discretize this function on an equidistant mesh between 0 and π. (The values between π and 2π aren't needed because it is odd.)

— Since v is odd, we also know that $v(x_0) = v(x_n) = 0$. (These are our Poisson boundary conditions.)

— Thus the discrete function is represented by the vector

$$\boldsymbol{y} = \begin{pmatrix} v_1 & \cdots & v_{n-1} \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^{n-1}.$$

Discrete Sine Transform

— As a basis for this space, let us use the sines

$$(\psi_k)_j = \sin\left(\frac{kj\pi}{n}\right), \qquad j = 1, \ldots, n-1,$$

and note that

$$\psi_k^{\mathsf{T}}\psi_\ell = \begin{cases} n/2, & k = \ell, \\ 0, & k \neq \ell. \end{cases}$$

— Thus

$$v_j = \sum_{k=1}^{n-1} \tilde{v}_k \sin\left(\frac{kj\pi}{n}\right) = \left(\boldsymbol{S}^{-1}\tilde{\boldsymbol{y}}\right)_j,$$

and

$$\tilde{v}_k = \frac{2}{n}\sum_{j=1}^{n-1} v_j \sin\left(\frac{jk\pi}{n}\right) = (\boldsymbol{S}\boldsymbol{y})_k$$

Tying it all toghether

— In particular, we have

$$\boldsymbol{Q} = \sqrt{\frac{n}{2}}\boldsymbol{S}, \qquad \boldsymbol{Q}^{\mathsf{T}} = \sqrt{\frac{2}{n}}\boldsymbol{S}^{-1}.$$

Therefore we can compute a matrix-vector product involving $\boldsymbol{Q}$ or $\boldsymbol{Q}^{\mathsf{T}}$ by using the DST.

— How to compute the DST quickly? Consider a vector

$$\boldsymbol{y} = \begin{pmatrix} v_1 & \cdots & v_{n-1} \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^{n-1}.$$

Construct the odd extended vector

$$\boldsymbol{w} = \begin{pmatrix} 0 & v_1 & \cdots & v_{n-1} & 0 & -v_{n-1} & \cdots & -v_1 \end{pmatrix}^{\mathsf{T}} \in \mathbb{R}^{2n}.$$

Tying it all toghether

— It turns out that the DST coefficients of $\boldsymbol{w}$ and the DFT coefficients of $\boldsymbol{w}$ are related by

$$\tilde{v}_k = 2\mathrm{i}\hat{w}_k, \qquad k = 1, \ldots, n-1$$

— Thus, we can find the DST by computing the DFT of the extended vector, multiplying the first $n-1$ coefficients (after the constant mode) by 2i, and throwing away the rest of them.

— This is good because there are very fast algorithms for computing the DFT: the infamous Fast Fourier Transform (*FFT*).

— One FFT is $\mathcal{O}(n \log n)$, so a matrix-matrix product using the FFT is $\mathcal{O}(n^2 \log n)$.

Tying it all together

We find the solution in three steps

1. Calculate $\tilde{\boldsymbol{B}}$ with two matrix-matrix products

$$\tilde{\boldsymbol{B}}^{\intercal} = \boldsymbol{S}^{-1}(\boldsymbol{S}\boldsymbol{B})^{\intercal}$$

   in $\mathcal{O}(n^2 \log n)$ operations.

2. Solve the system

$$\tilde{x}_{i,j} = \frac{\tilde{b}_{i,j}}{\lambda_i + \lambda_j}$$

   in $\mathcal{O}(n^2)$ operations.

3. Recover the solution with two matrix-matrix products

$$\boldsymbol{X} = \boldsymbol{S}^{-1}(\boldsymbol{S}\tilde{\boldsymbol{X}}^{\intercal})^{\intercal}$$

   in $\mathcal{O}(n^2 \log n)$ operations.