Which linear systems can be solved optimally?

Ed Bueler

UAF Math 692 Scalable Seminar

Spring 2023

Outline

1. how fast is the basic matrix-vector product $z = Ax$?
2. complexity of Gaussian elimination for linear systems $Ax = b$
3. banded matrices
4. sparse storage?
5. circulant matrices

matrix-vector products

- the life-goal of a matrix $A \in \mathbb{R}^{m \times n}$ is to act on (multiply) vectors $x \in \mathbb{R}^n$:

$$z = Ax \qquad \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

- this is a simple and familiar operation:

$$z_i = \sum_{j=0}^{n-1} a_{ij} x_j \qquad \text{for } i = 0, \ldots, m-1$$

  - note: I will index rows and columns starting from 0 in this talk
  - note: by default, vectors in $\mathbb{R}^n$ are column vectors

- how fast is matrix-vector multiplication for generic $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$?

- that is, what is its (*algorithmic*) *complexity*?

- this talk is based on genuine codes in Python :

```
def matvec(A,x):
    from numpy import zeros, shape
    [m,n] = shape(A)
    z = zeros((m,1))
    for i in range(m):
        s = 0.0
        for j in range(n):
            s += A[i][j] * x[j]
        z[i] = s
    return z
```

- actually, this talk is based on genuine pseudocodes in Python-ish:

```
def matvec(A,x):
    for i = 0 to m-1:
        s = 0.0
        for j = 0 to n-1:
            s += A[i][j] * x[j]
        z[i] = s
    return z
```

matvec flops

```
def matvec(A,x):
    for i = 0 to m-1:
        s = 0.0
        for j = 0 to n-1:
            s += A[i][j] * x[j]
        z[i] = s
    return z
```

- matvec does exactly $2mn$ floating point operations (*flops*)
  - n additions and n multiplications for each of m entries of result vector Ax
- complexity in big-O notation:
  - $O(mn)$ flops
  - $O(mn)$ storage, including inputs A and x
  - $O(mn)$ time (maybe?)

my definition of *optimal*

- I will throw around the word “optimal” in this talk

Definition

an algorithm for computing a function on a class of problems, which acts on floating-point data of size N, is *optimal* if it requires

$$O(N) \qquad \text{or} \qquad O(N \log N) \qquad \text{flops}$$

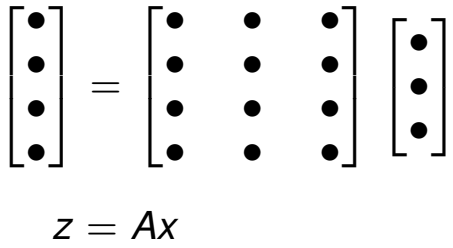
as $N \to \infty$

- this means there exists C so that for all problems, of any size N,

$$(\text{flops}) \leq C\,N \qquad \text{or} \qquad (\text{flops}) \leq C\,N \log N$$

  - quantifier order matters: $\exists C \; \forall$ problems $\forall N \; \ldots$
- is `matvec` optimal?

is `matvec` optimal?

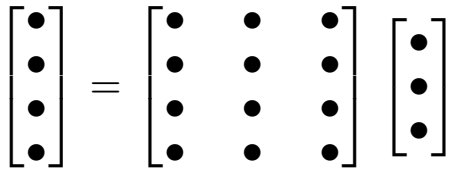


$z = Ax$

- is `matvec` optimal?
- it depends on what are the data and the problems!

1. if the data $=$ vector x ($N = n$), and we consider any matrix $A \in \mathbb{R}^{m \times n}$ for which multiplication is valid, then it ***is not* optimal**
   - $2mn = 2mN \not\leq CN$ for all problems $\leftarrow$ *m depends on the problem*
2. if the data $=$ vector x ($N = n$), and we consider a all matrices A with (fixed) m rows, then it ***is* optimal**
   - $2mn = 2mN \leq CN$ for $C = m$
3. if the data $=$ matrix A ($N = mn$), and we consider any x, then it ***is* optimal**
   - $2mn = 2N \leq CN$ for $C = 2$
4. if the data $=$ vector x ($N = n$), and we consider any square tridiagonal matrix A, then it ***is* optimal** $\leftarrow$ *not counting $a_{ij}x_j$ if $a_{ij} = 0$*
   - $2 \cdot 3 \cdot n = 6N \leq CN$ for $C = 6$

is `matvec` optimal?

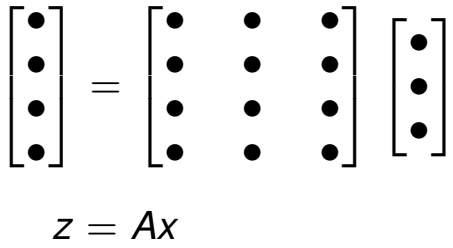


$z = Ax$

- is `matvec` optimal?
- it depends on what are the data and the problems!

1. if the data = vector x ($N = n$), and we consider any matrix $A \in \mathbb{R}^{m \times n}$ for which multiplication is valid, then it ***is not* optimal**
   - $2mn = 2mN \not\leq CN$ for all problems $\leftarrow$ *m depends on the problem*
2. if the data = vector x ($N = n$), and we consider a all matrices A with (fixed) m rows, then it ***is* optimal**
   - $2mn = 2mN \leq CN$ for $C = m$
3. if the data = matrix A ($N = mn$), and we consider any x, then it ***is* optimal**
   - $2mn = 2N \leq CN$ for $C = 2$
4. if the data = vector x ($N = n$), and we consider any square tridiagonal matrix A, then it ***is* optimal** $\leftarrow$ *not counting $a_{ij}x_j$ if $a_{ij} = 0$*
   - $2 \cdot 3 \cdot n = 6N \leq CN$ for $C = 6$

is `matvec` optimal?

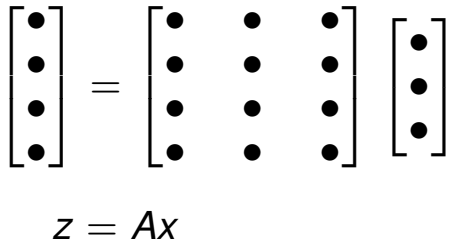


$z = Ax$

- is `matvec` optimal?
- it depends on what are the data and the problems!

1. if the data $=$ vector x ($N = n$), and we consider any matrix $A \in \mathbb{R}^{m \times n}$ for which multiplication is valid, then it ***is not* optimal**
   - $2mn = 2mN \not\leq CN$ for all problems $\leftarrow$ *m depends on the problem*
2. if the data $=$ vector x ($N = n$), and we consider a all matrices A with (fixed) m rows, then it ***is* optimal**
   - $2mn = 2mN \leq CN$ for $C = m$
3. if the data $=$ matrix A ($N = mn$), and we consider any x, then it ***is* optimal**
   - $2mn = 2N \leq CN$ for $C = 2$
4. if the data $=$ vector x ($N = n$), and we consider any square tridiagonal matrix A, then it ***is* optimal** $\leftarrow$ *not counting $a_{ij}x_j$ if $a_{ij} = 0$*
   - $2 \cdot 3 \cdot n = 6N \leq CN$ for $C = 6$

is `matvec` optimal?

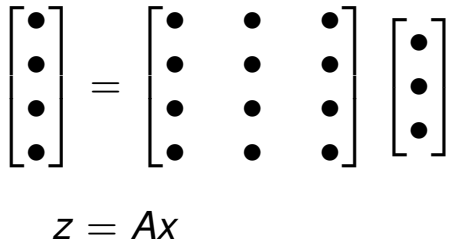


$z = Ax$

- is `matvec` optimal?
- it depends on what are the data and the problems!

1. if the data $=$ vector x ($N = n$), and we consider any matrix $A \in \mathbb{R}^{m \times n}$ for which multiplication is valid, then it ***is not* optimal**
   - $2mn = 2mN \not\leq CN$ for all problems $\leftarrow$ *m depends on the problem*
2. if the data $=$ vector x ($N = n$), and we consider a all matrices A with (fixed) m rows, then it ***is* optimal**
   - $2mn = 2mN \leq CN$ for $C = m$
3. if the data $=$ matrix A ($N = mn$), and we consider any x, then it ***is* optimal**
   - $2mn = 2N \leq CN$ for $C = 2$
4. if the data $=$ vector x ($N = n$), and we consider any square tridiagonal matrix A, then it ***is* optimal** $\leftarrow$ *not counting $a_{ij}x_j$ if $a_{ij} = 0$*
   - $2 \cdot 3 \cdot n = 6N \leq CN$ for $C = 6$

is `matvec` optimal?



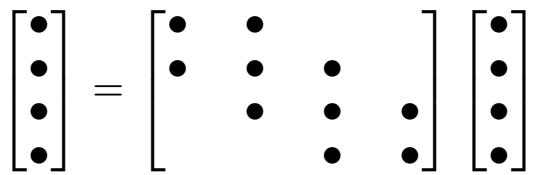
$z = Ax$

- is `matvec` optimal?
- it depends on what are the data and the problems!

1. if the data $=$ vector x ($N = n$), and we consider any matrix $A \in \mathbb{R}^{m \times n}$ for which multiplication is valid, then it ***is not* optimal**
   - $2mn = 2mN \not\leq CN$ for all problems $\leftarrow$ *m depends on the problem*
2. if the data $=$ vector x ($N = n$), and we consider a all matrices A with (fixed) m rows, then it ***is* optimal**
   - $2mn = 2mN \leq CN$ for $C = m$
3. if the data $=$ matrix A ($N = mn$), and we consider any x, then it ***is* optimal**
   - $2mn = 2N \leq CN$ for $C = 2$
4. if the data $=$ vector x ($N = n$), and we consider any square tridiagonal matrix A, then it ***is* optimal** $\leftarrow$ *not counting $a_{ij}x_j$ if $a_{ij} = 0$*
   - $2 \cdot 3 \cdot n = 6N \leq CN$ for $C = 6$

tridiagonal matrix-vector product

- to be honest, the square tridiagonal case is a different algorithm:



```
def matvec_tri(A,x):
    z[0] = A[0][0] * x[0] + A[0][1] * x[1]
    for i = 1 to m-2:
        s = 0.0
        for j = i-1 to i+1:
            s += A[i][j] * x[j]
        z[i] = s
    z[m-1] = A[m-1][m-2] * x[m-2] + A[m-1][m-1] * x[m-1]
    return z
```

- this algorithm does $O(m)$ flops, so it is optimal in any interpretation

regarding optimal algorithms

- I will stick to my definition of "optimal" below!
- however, it is always fair to stop me and ask "what is the data?" or "what is the class of problems?" at any time
  - because whether an algorithm is optimal depends on an agreement regarding which is the data and which is the problem class
- it is fair to wonder if flops are a good metric for modern computers
  - but every other metric is messier?
  - perhaps give a talk using another metric?

Definition

an algorithm for computing a function on a class of problems, which acts on floating-point data of size N, is *optimal* if it requires

$$O(N) \qquad \text{or} \qquad O(N \log N) \qquad \text{flops}$$

as $N \to \infty$

regarding optimal algorithms

- I will stick to my definition of "optimal" below!
- however, it is always fair to stop me and ask "what is the data?" or "what is the class of problems?" at any time
  - because whether an algorithm is optimal depends on an agreement regarding which is the data and which is the problem class
- it is fair to wonder if flops are a good metric for modern computers
  - but every other metric is messier?
  - perhaps give a talk using another metric?

Definition

an algorithm for computing a function on a class of problems, which acts on floating-point data of size N, is *optimal* if it requires

$$O(N) \qquad \text{or} \qquad O(N \log N) \qquad \text{flops}$$

as $N \to \infty$

regarding optimal algorithms

- I will stick to my definition of "optimal" below!
- however, it is always fair to stop me and ask "what is the data?" or "what is the class of problems?" at any time
  - because whether an algorithm is optimal depends on an agreement regarding which is the data and which is the problem class
- it is fair to wonder if flops are a good metric for modern computers
  - but every other metric is messier?
  - *put up or shut up!*

Definition

an algorithm for computing a function on a class of problems, which acts on floating-point data of size N, is *optimal* if it requires

$$O(N) \qquad \text{or} \qquad O(N \log N) \qquad \text{flops}$$

as $N \to \infty$

regarding optimal algorithms

- I will stick to my definition of "optimal" below!
- however, it is always fair to stop me and ask "what is the data?" or "what is the class of problems?" at any time
  - because whether an algorithm is optimal depends on an agreement regarding which is the data and which is the problem class
- it is fair to wonder if flops are a good metric for modern computers
  - but every other metric is messier?
  - perhaps give a talk using another metric?

Definition

an algorithm for computing a function on a class of problems, which acts on floating-point data of size N, is *optimal* if it requires

$$O(N) \qquad \text{or} \qquad O(N \log N) \qquad \text{flops}$$

as $N \to \infty$

Outline

1. how fast is the basic matrix-vector product $z = Ax$?
2. complexity of Gaussian elimination for linear systems $Ax = b$
3. banded matrices
4. sparse storage?
5. circulant matrices

Gaussian elimination to solve linear systems $Ax = b$

- Gaussian elimination (GE) with back-substitution (BS) is the familiar way to solve linear systems $Ax = b$
- for example:

$$\begin{aligned} x_0 + 2x_1 + 3x_2 + 4x_3 &= 6 \\ -x_0 \quad - 2x_2 - 4x_3 &= -7 \\ 2x_0 + 4x_1 + 9x_2 + 9x_3 &= 16 \\ x_0 + 4x_1 + 7x_2 + 7x_3 &= 11 \end{aligned} \xrightarrow{\text{T}} \begin{aligned} x_0 + 2x_1 + 3x_2 + 4x_3 &= 6 \\ 2x_1 + x_2 \quad &= -1 \\ 3x_2 + x_3 &= 4 \\ 2x_3 &= 2 \end{aligned} \xrightarrow{\text{BS}} \begin{aligned} x_0 &= 1 \\ x_1 &= -1 \\ x_2 &= 1 \\ x_3 &= 1 \end{aligned}$$

- triangularization $\xrightarrow{\text{T}}$ occurs in stages, for example:

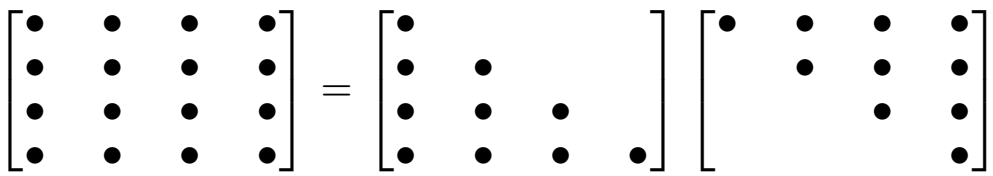
$$\begin{aligned} R_1 &\leftarrow R_1 + R_0 \\ R_2 &\leftarrow R_2 - 2R_0 \\ R_3 &\leftarrow R_3 - R_0 \end{aligned} \qquad \begin{aligned} R_2 &\leftarrow R_2 \\ R_3 &\leftarrow R_3 - R_2 \end{aligned} \qquad R_3 \leftarrow R_3 - R_2$$

- each stage zeros-out a column, and modifies everything to the right:

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & \bullet \end{bmatrix}$$

Gaussian elimination is LU decomposition

- GEBS is usually implemented as LU factorization (*decomposition*) of A,



$$A = L\,U,$$

  followed by triangular-system solves to solve $Ax = b$:

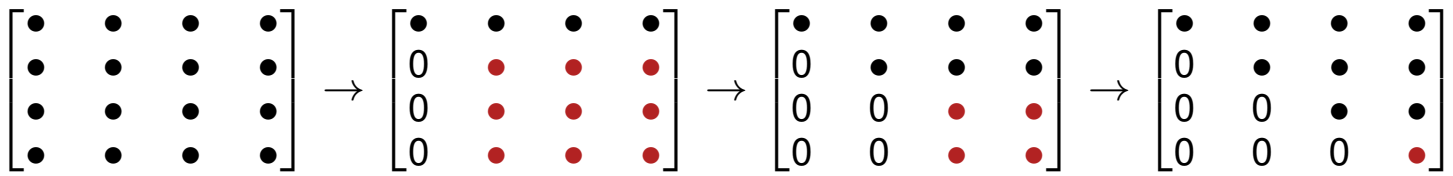
$$Ax = b \quad \Longleftrightarrow \quad L(Ux) = b \quad \Longleftrightarrow \quad \begin{array}{l} Ly = b \\ Ux = y \end{array}$$

  - this way of organizing does not affect the algorithm's complexity
- if $A \in \mathbb{R}^{m \times m}$ then the triangular solves $Ly = b$ and $Ux = y$ (forward- & back-sub.) each require exactly m^2 floating-point operations
  - showing this is a good exercise
- partial pivoting should be done for numerical stability
  - result: $PA = LU$ where P is a permutation-of-rows operation
  - partial pivoting is $O(m^2)$; it does not affect complexity at leading order

LU decomposition pseudocode

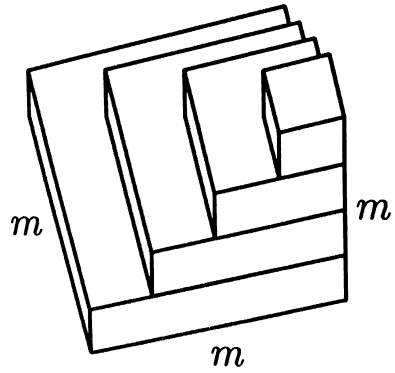
- the $A = LU$ triangularization step dominates the time for GEBS
- one converts A to U by zeroing-out columns below the diagonal
- each such stage modifies a square of entries to its right

```
def lu(A):
    U, L = A, eye(m,m)
    for k = 0 to m-2:
        for i = k+1 to m-1:
            L[i][k] = U[i][k] / U[k][k]
            for j = k to m-1:
                U[i][j] = U[i][j] - L[i][k] * U[k][j]
    return U, L
```



scaling of Gaussian elimination

- $A = LU$ requires $\frac{2}{3}m^3 + O(m^2) = O(m^3)$ floating point operations
  - because the volume of an $m \times m \times m$ pyramid is $\frac{1}{3}m^3$,
  - and there are two flops per modified entry




```
def lu(A):
    U, L = A, eye(m,m)
    for k = 0 to m-2:
        for i = k+1 to m-1:
            L[i][k] = U[i][k] / U[k][k]
            for j = k to m-1:
                U[i][j] = U[i][j] - L[i][k] * U[k][j]
    return U, L
```

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & \bullet \end{bmatrix}$$

scaling for solving **dense** linear systems

- the rest of the talk assumes $A \in \mathbb{R}^{m \times m}$ is square and invertible
- one **might** regard the data of "solve $Ax = b$" as the matrix A itself
- then an optimal solver for **dense** A would require $O(m^2)$ flops
  - one cannot do better because generic dense matrices A have m^2 entries, and thus touching each entry is $O(m^2)$
- however, we know GEBS is $O(m^3)$, thus not optimal
- $\exists$ solver algorithms[1] for dense $A \in \mathbb{R}^{m \times m}$ which scale as $O(m^{2.376})$
  - famously starting with V. Strassen (1969). *Gaussian elimination is not optimal*, Numer. Math. 13, 354–356
- however, this "fast dense solver" game is not practical
  - fascinating algebra, but little impact on scientific/engineering software
  - for the applications of greatest interest, $O(m^{2.376})$ is catastophically slow

[1] these "fast solvers" can even be stable with respect to rounding errors? this is not clear to me! see J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg (2007). *Fast matrix multiplication is stable*, Numer. Math. 106 (2), 199–224

scaling for solving **dense** linear systems

- the rest of the talk assumes $A \in \mathbb{R}^{m \times m}$ is square and invertible
- one **might** regard the data of "solve $Ax = b$" as the matrix A itself
- then an optimal solver for **dense** A would require $O(m^2)$ flops
  - one cannot do better because generic dense matrices A have m^2 entries, and thus touching each entry is $O(m^2)$
- however, we know GEBS is $O(m^3)$, thus not optimal
- $\exists$ solver algorithms[1] for dense $A \in \mathbb{R}^{m \times m}$ which scale as $O(m^{2.376})$
  - famously starting with V. Strassen (1969). *Gaussian elimination is not optimal*, Numer. Math. 13, 354–356
- however, this "fast dense solver" game is not practical
  - fascinating algebra, but little impact on scientific/engineering software
  - for the applications of greatest interest, $O(m^{2.376})$ is catastophically slow

[1] these "fast solvers" can even be stable with respect to rounding errors? this is not clear to me! see J. Demmel, I. Dumitriu, O. Holtz, and R. Kleinberg (2007). *Fast matrix multiplication is stable*, Numer. Math. 106 (2), 199–224

optimal solvers for $Ax = b$, arising from applications?

- the rest of the talk assumes the size of the data is m
  - m = (number of unknowns) = (number of rows in A) = (length of b)
  - $A \in \mathbb{R}^{m \times m}$, $b \in \mathbb{R}^m$
- are there special classes of matrices $A \in \mathbb{R}^{m \times m}$, which routinely arise in applications, for which there are $O(m^1)$ solvers?
  - yes
- $\exists O(m^1)$ solvers $\implies$ $\exists$ exploitable matrix structure
  - these are *not* generic matrices
  - most are sparse, but some are dense!
- a *huge* amount of science/engineering-relevant software supports these special matrix classes and optimal solver algorithms

Outline

1. how fast is the basic matrix-vector product $z = Ax$?
2. complexity of Gaussian elimination for linear systems $Ax = b$
3. banded matrices
4. sparse storage?
5. circulant matrices

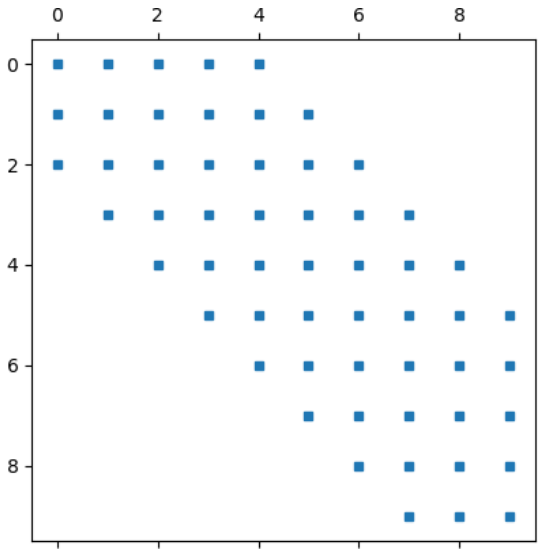
banded matrices

Definition

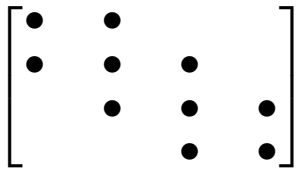
a square matrix A is *banded* with *lower bandwidth* p and *upper bandwidth* q if

$$i > j + p \quad \text{or} \quad j > i + q \quad \Longrightarrow \quad a_{ij} = 0$$

- example non-zero pattern with $p = 2, q = 4$:



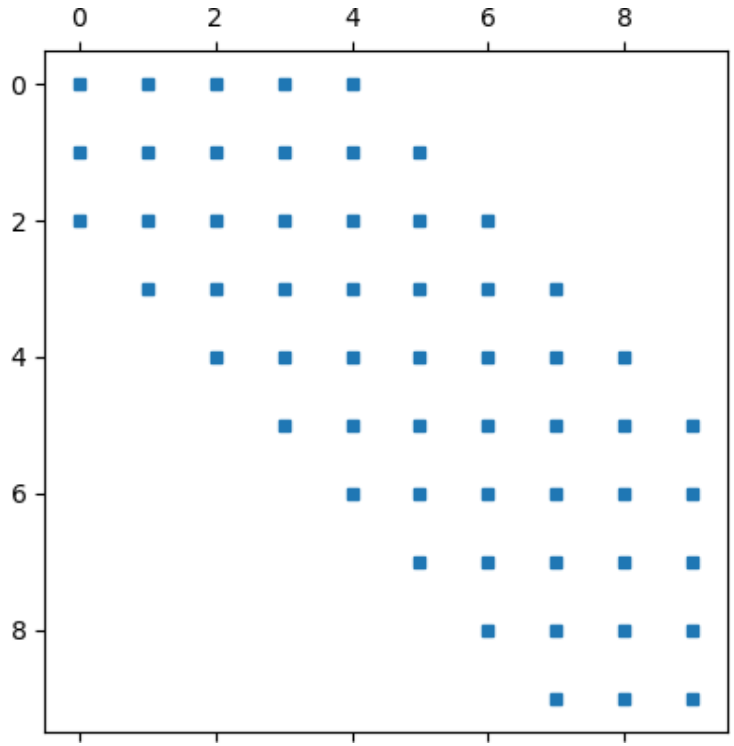
- I have already mentioned tridiagonal matrices
  - they are banded with $p, q = 1, 1$



generating and viewing banded matrices

- random $m = 10$ matrix with bandwidths $p = 2$ (lower) and $q = 4$ (upper):

```
from numpy import tril, triu
from numpy.random import randn
from matplotlib.pyplot import spy, show
m,p,q = 10,2,4
A = tril(triu(randn(m,m),-p),+q)
spy(A,markersize=5.0)
show()
```



banded LU decomposition

- specializing LU decomposition to banded matrices is straightforward
- the pseudocode below computes $A = LU$ *without* pivoting
  - the algorithm below is *in place*; it overwrites A with L and U
  - the tridiagonal case is often called the *Thomas* algorithm
  - skipping pivoting is generally unstable?

```
def lu_banded(A):
    for k = 0 to m-2:
        for i = k+1 to min{k+p,m}:
            A[i][k] = A[i][k] / U[k][k]
        for j = k+1 to min{k+q,m}:
            for i = k+1 to min{k+p,m}:
                A[i][j] = A[i][j] - A[i][k] * A[k][j]
    return A
```

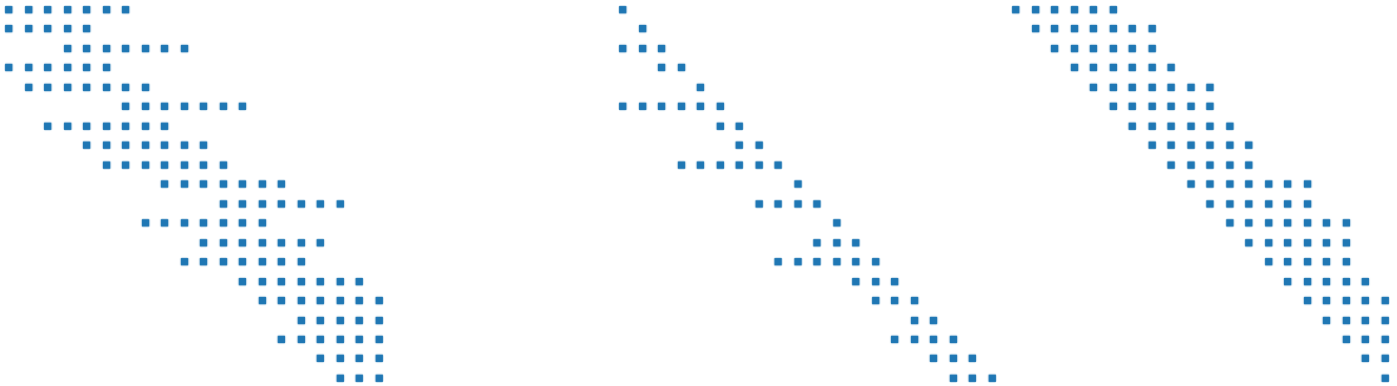
- this algorithm does $2pqm + \text{lower} = O(pqm) = O(m)$ flops
- so solving a banded linear system via LU is optimal $O(m)$
  - with a constant proportional to the product pq of bandwidths

effect of pivoting on banded LU

- pivoting expands the band, but often acceptably

  theorem. [Golub & van Loan, thm 4.3.2] if A is p, q banded then $PA = LU$ where U has upper bandwidth at most $p + q$; though L can have any bandwidth, it has at most $p + 1$ nonzeros per column

- for example, I generated a 20×20 random matrix: $A =$ 
  - $p, q = 2, 4$



- if $p, q = O(1)$ then LU with partial pivoting is optimal $O(m)$

where do banded matrices come from?

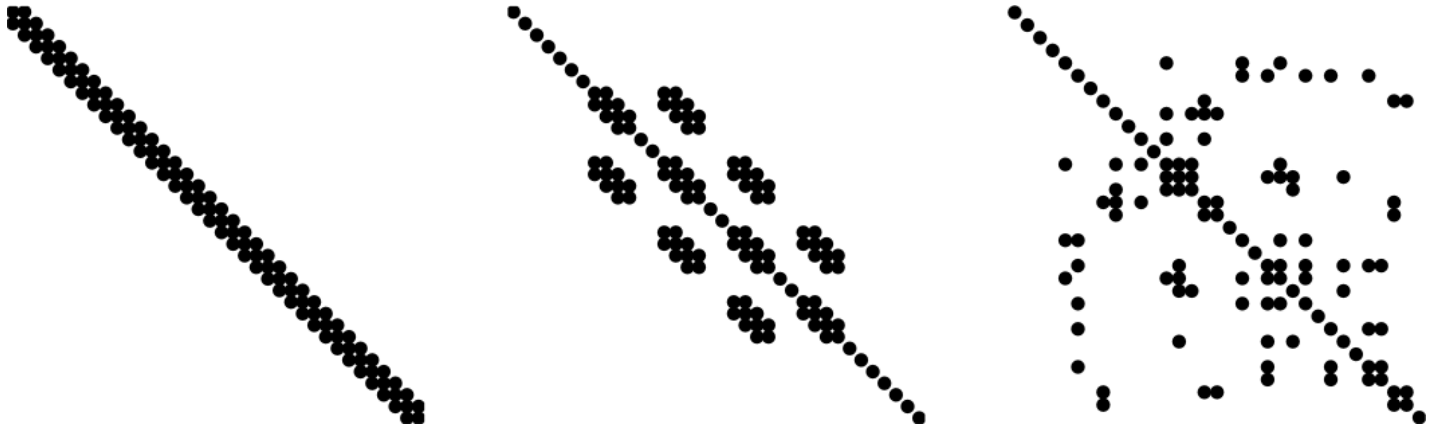
- one source: structured-grid approximations of differential equations
- consider a 2-point boundary value problem

$$u''(x) + p(x)u'(x) = f(x)$$

- discretize it on an n point grid:

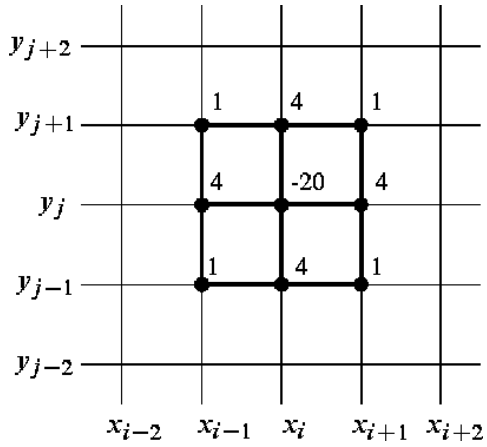
$$\frac{U_{j+1} - 2U_j + U_{j-1}}{h^2} + p(x_j)\frac{U_{j+1} - U_{j-1}}{2h} = f(x_j)$$

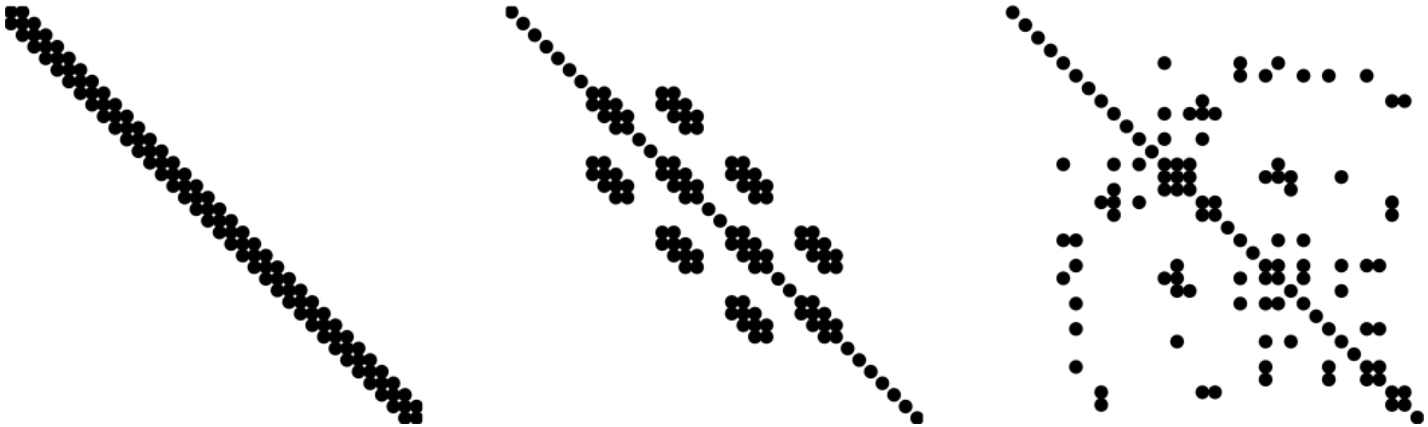
- yields a tridiagonal matrix A (left)



where do banded matrices come from?

- discretizations of 2D and 3D partial differential equations (PDEs) also create banded matrices
- however, the bandwidth grows with the number of grid points in each direction
- for example, a $n_x \times n_y$ 2D grid has $m = n_x n_y$ unknowns and generates A with bandwidth $p = q = \min\{n_x, n_y\} = O(m^{1/2})$ (middle)
- an unstructured triangularization can only generate a banded matrix if the ordering of unknowns is carefully chosen (right; not-so-good ordering)





optimal 2D and 3D PDE solutions?

- FD, FE, and FV discretizations fo 2D and 3D problems generate sparse matrices with $O(1)$ nonzeros per row
  - in the unstructured case this assumes a (realistic!) minimum-degree condition on the mesh
- if the mesh/grid is structured then the matrix is also banded
  - 1D ODE BVPs yield $O(1)$ bandwidth
  - 2D yields $O(m^{1/2})$ bandwidth
  - 3D yields $O(m^{2/3})$ bandwidth
  - a dense matrix has $O(m^1)$ bandwidth
- banded LU solvers for 2D and 3D PDEs are generally *not* optimal
  - 2D: $O(pqm) = O(m^2)$ flops
  - 3D: $O(pqm) = O(m^{7/3})$ flops
- other ideas are needed!
- multigrid is such an idea

Outline

1. how fast is the basic matrix-vector product $z = Ax$?
2. complexity of Gaussian elimination for linear systems $Ax = b$
3. banded matrices
4. sparse storage?
5. circulant matrices

sparse storage basics

- suppose you have a sparse A
  - for example, suppose A has $O(1)$ nonzeros per row
    - e.g. from a PDE discretization scheme
  - you want to apply A to a vector $x \in \mathbb{R}^m$ in an efficient $O(m)$ manner

Definition

sparse storage is a data structure holding only the nonzero matrix entries

- computation of Ax, and extraction of a_{ij}, must be implemented
- for example, `scipy.sparse` has 7 such data structures (object classes):

`bsr_array`(arg1[, shape, dtype, copy, blocksize])	Block Sparse Row array
`coo_array`(arg1[, shape, dtype, copy])	A sparse array in COOrdinate format.
`csc_array`(arg1[, shape, dtype, copy])	Compressed Sparse Column array
`csr_array`(arg1[, shape, dtype, copy])	Compressed Sparse Row array
`dia_array`(arg1[, shape, dtype, copy])	Sparse array with DIAgonal storage
`dok_array`(arg1[, shape, dtype, copy])	Dictionary Of Keys based sparse array.
`lil_array`(arg1[, shape, dtype, copy])	Row-based LIst of Lists sparse array

compressed sparse row (CSR) example

$$A = \begin{bmatrix} 1 & 0 & \pi \\ 0 & 0 & 3.1 \\ \sin(1) & 5 & 6 \end{bmatrix}$$

- store A sparsely by giving (`row`, `col`) indices of nonzero entries `data`:

```
import numpy as np
from scipy.sparse import csr_array
row = np.array([0, 0, 1, 2, 2, 2])
col = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1.0, np.pi, 3.1, np.sin(1.0), 5.0, 6.0])
A = csr_array((data, (row, col)), shape=(3, 3))
```

- look at the result as entered:

```
In [4]: print(A)
  (0, 0)        1.0
  (0, 2)        3.141592653589793
  (1, 2)        3.1
  (2, 0)        0.8414709848078965
  (2, 1)        5.0
  (2, 2)        6.0
```

compressed sparse row (CSR) example

$$A = \begin{bmatrix} 1 & 0 & \pi \\ 0 & 0 & 3.1 \\ \sin(1) & 5 & 6 \end{bmatrix}$$

- store A sparsely by giving (`row`, `col`) indices of nonzero entries `data`:

```
import numpy as np
from scipy.sparse import csr_array
row = np.array([0, 0, 1, 2, 2, 2])
col = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1.0, np.pi, 3.1, np.sin(1.0), 5.0, 6.0])
A = csr_array((data, (row, col)), shape=(3, 3))
```

- look at the result as a "full" matrix:

```
In [5]: print(A.toarray())
[[1.         0.         3.14159265]
 [0.         0.         3.1       ]
 [0.84147098 5.         6.        ]]
```

compressed sparse row (CSR) example

$$A = \begin{bmatrix} 1 & 0 & \pi \\ 0 & 0 & 3.1 \\ \sin(1) & 5 & 6 \end{bmatrix}$$

- store A sparsely by giving (`row`, `col`) indices of nonzero entries `data`:

```
import numpy as np
from scipy.sparse import csr_array
row = np.array([0, 0, 1, 2, 2, 2])
col = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1.0, np.pi, 3.1, np.sin(1.0), 5.0, 6.0])
A = csr_array((data, (row, col)), shape=(3, 3))
```

- look at the result *as actually stored*:

```
In [6]: print(A.indptr),print(A.indices),print(A.data)
[0 2 3 6]
[0 2 2 0 1 2]
[1.  3.14159265  3.1  0.84147098  5.  6. ]
```

how CSR `matvec` works

$$A = \begin{bmatrix} 1 & 0 & \pi \\ 0 & 0 & 3.1 \\ \sin(1) & 5 & 6 \end{bmatrix}$$

```
A.indptr  = [0 2 3 6]
A.indices = [0 2 2 0 1 2]
A.data    = [1.  3.14159265  3.1  0.84147098  5.  6. ]
```

```
def matvec_csr(A,x):
    for i = 0 to m-1:
        kk = A.indptr[i]:A.indptr[i+1]
        z[i] = np.dot(A.data[kk], x[A.indices[kk]])
    return z
```

- if A has $O(1)$ nonzeros per row then this operation is $O(m)$ flops
- note z is constructed sequentially but x is addressed randomly
  - CSC sparse storage reverses this pattern

Outline

1. how fast is the basic matrix-vector product $z = Ax$?
2. complexity of Gaussian elimination for linear systems $Ax = b$
3. banded matrices
4. sparse storage?
5. circulant matrices

circulant matrices

Definition

a square matrix $A \in \mathbb{R}^{m \times m}$ is *circulant* if each wrapped diagonal is constant, that is, there is a vector $c \in \mathbb{R}^m$, the first column of A, so that

$$a_{ij} = c_{i-j \bmod m}$$

- example:

$$A = \begin{bmatrix} 2 & 7 & -1 & 3 & 8 \\ 8 & 2 & 7 & -1 & 3 \\ 3 & 8 & 2 & 7 & -1 \\ -1 & 3 & 8 & 2 & 7 \\ 7 & -1 & 3 & 8 & 2 \end{bmatrix} \qquad \text{is from } c = \begin{bmatrix} 2 \\ 8 \\ 3 \\ -1 \\ 7 \end{bmatrix}$$

- these are *dense* matrices
- however, they can be stored with $O(m)$ storage: store c not A

downshifting to circulate

- let D_m be the *downshift* matrix, for example

$$D_5 = \begin{bmatrix} & & & & 1 \\ 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \end{bmatrix}$$

so that, for example,

$$D_5 c = \begin{bmatrix} & & & & 1 \\ 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \end{bmatrix} \begin{bmatrix} 2 \\ 8 \\ 3 \\ -1 \\ 7 \end{bmatrix} = \begin{bmatrix} 7 \\ 2 \\ 8 \\ 3 \\ -1 \end{bmatrix}$$

recall

$$A = \begin{bmatrix} 2 & 7 & -1 & 3 & 8 \\ 8 & 2 & 7 & -1 & 3 \\ 3 & 8 & 2 & 7 & -1 \\ -1 & 3 & 8 & 2 & 7 \\ 7 & -1 & 3 & 8 & 2 \end{bmatrix}$$

- then, for example: $A = 2I + 8D_5 + 3D_5^2 + (-1)D_5^3 + 7D_5^4$ ↑
- generally, if A is circulant with first column c then

$$A = c_0 I + c_1 D_m + c_2 D_m^2 + \cdots + c_{m-1} D_m^{m-1} = \sum_{k=0}^{m-1} c_k D_m^k$$

eigenvalues and diagonalization

- will return to circulants and downshifts . . . but first recall

Definition

a square matrix A has an *eigenvector* v if v is nonzero and $Av = \lambda v$ for some scalar λ, in which case λ is the *eigenvalue*

- suppose we build V by using eigenvectors as the columns:

$$\left[\begin{array}{c|c|c} Av_0 & \cdots & Av_k \end{array}\right] = \left[\begin{array}{c|c|c} \lambda_0 v_0 & \cdots & \lambda_k v_k \end{array}\right]$$

$$AV = V\Lambda$$

  where Λ is diagonal with $\lambda_0, \ldots, \lambda_k$ on the diagonal
- if A is $m \times m$, and if there are m linearly-independent eigevectors $v_0, \ldots, v_{m-1}$ of A, so V is invertible, then we say A is *diagonalizable*:

$$AV = V\Lambda \qquad \Longleftrightarrow \qquad A = V\Lambda V^{-1}$$

diagonalization and polynomials

- suppose we have a scalar polynomial of degree n:

$$p(\xi) = c_0 + c_1\xi + c_2\xi^2 + \cdots + c_n\xi^n$$

- if $A = V\Lambda V^{-1}$ is diagonalizable then $p(A)$ is easily computed:

$$\begin{aligned}
p(A) &= c_0 + c_1 A + c_2 A^2 + \cdots + c_n A^n \\
&= c_0 I + c_1 V\Lambda V^{-1} + c_2 \left(V\Lambda V^{-1}\right)^2 + \cdots + c_n \left(V\Lambda V^{-1}\right)^n \\
&= c_0 VIV^{-1} + c_1 V\Lambda V^{-1} + c_2 V\Lambda^2 V^{-1} + \cdots + c_n V\Lambda^n V^{-1} \\
&= V\left(c_0 + c_1\Lambda + \cdots + c_n\Lambda^n\right)V^{-1} \\
&= V \begin{bmatrix} p(\lambda_0) & & \\ & \ddots & \\ & & p(\lambda_{m-1}) \end{bmatrix} V^{-1} \\
&= Vp(\Lambda)V^{-1}
\end{aligned}$$

diagonalizing downshift matrices?

- recall: any circulant matrix is a polynomial in the downshift matrix D_m

$$A = c_0 I + c_1 D_m + c_2 D_m^2 + \cdots + c_{m-1} D_m^{m-1} = p(D_m)$$

  - c, the first column of A, gives the coefficients of $p(\xi)$
- can we diagonalize D_m?

can we diagonalize the downshift D_m?

- let us suppose that there is a magic number $\omega_m \neq 1$ such that $(\omega_m)^m = 1$
- then we can generate eigenvectors of D_m
- for example: $m = 5$ and $\omega = \omega_5$

$$D_5 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ 1 & & & & \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 1 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$D_5 \begin{bmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \\ \omega^4 \end{bmatrix} = \begin{bmatrix} & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ 1 & & & & \end{bmatrix} \begin{bmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \\ \omega^4 \end{bmatrix} = \begin{bmatrix} \omega^4 \\ 1 \\ \omega \\ \omega^2 \\ \omega^3 \end{bmatrix} = \omega^{-1} \begin{bmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \\ \omega^4 \end{bmatrix}$$

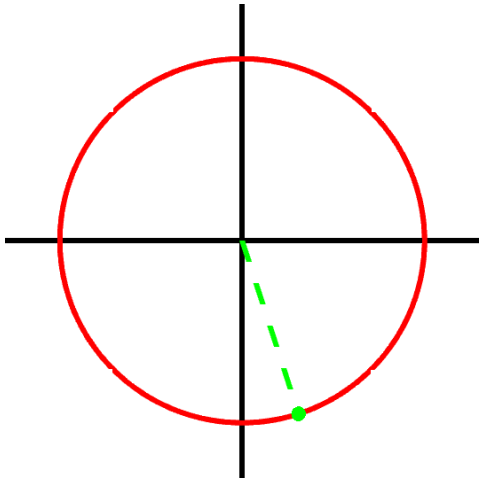
$$D_5 \begin{bmatrix} 1 \\ \omega^2 \\ \omega^4 \\ \omega^6 \\ \omega^8 \end{bmatrix} = \begin{bmatrix} & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ 1 & & & & \end{bmatrix} \begin{bmatrix} 1 \\ \omega^2 \\ \omega^4 \\ \omega^6 \\ \omega^8 \end{bmatrix} = \begin{bmatrix} \omega^8 \\ 1 \\ \omega^2 \\ \omega^4 \\ \omega^6 \end{bmatrix} = \omega^{-2} \begin{bmatrix} 1 \\ \omega^2 \\ \omega^4 \\ \omega^6 \\ \omega^8 \end{bmatrix}$$

$$\vdots$$

the magic exponential

- such a magic number exists!

$$\omega_m = e^{-i2\pi/m}$$



- the columns of the following symmetric $m \times m$ matrix are linearly-independent:

$$F_m = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_m & \omega_m^2 & \dots & \omega_m^{m-1} \\ 1 & \omega_m^2 & \omega_m^4 & & \omega_m^{2(m-1)} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & \omega_m^{m-1} & \omega_m^{2(m-1)} & \dots & \omega_m^{(m-1)(m-1)} \end{bmatrix}$$

- this matrix diagonalizes D_m:

$$D_m = F_m \begin{bmatrix} 1 & & & \\ & \bar{\omega}_m & & \\ & & \ddots & \\ & & & \bar{\omega}_m^{m-1} \end{bmatrix} F_m^{-1}$$

the discrete Fourier transform

- this matrix F_m is called the *discrete Fourier transform* (DFT)
- the *fast Fourier transform* (`FFT`) is an algorithm which applies F_m to any $x \in \mathbb{R}^m$ in $O(m \log m)$ flops
  - this assumes m is "highly composite" ... $m = 2^k$ is most common
  - the same algorithm (`barFFT` applies $\bar{F}_m$) can apply the inverse:

$$F_m^{-1} = \frac{1}{m} \bar{F}_m$$

fast solution of circulant systems

- suppose A is circulant, with first column c, and we want to solve $Ax = b$
- we know

$$A = c_0 I + c_1 D_m + c_2 D_m^2 + \cdots + c_{m-1} D_m^{m-1} = p(D_m)$$

  and $D_m = F_m \Lambda F_m^{-1}$ where $\lambda_j = \bar{\omega}_m^j$
- fast solution process:

$$Ax = b \quad \Longleftrightarrow \quad F_m p(\Lambda) F_m^{-1} x = b \quad \Longleftrightarrow \quad \begin{array}{c} u = F_m^{-1} b \\ v = p(\Lambda)^{-1} u \\ x = F_m v \end{array}$$

```
def solve_circulant(c,x,b):
    z = barFFT(c)
    u = barFFT(b)
    v = u ./ z
    x = FFT(v)
    x /= m
    return x
```

$\leftarrow$ $O(m \log m)$ flops . . . optimal!

summary: some directions suggested by this talk

- sparse storage schemes
  - practicalities?
  - parallelization?
- sparse direct linear algebra
  - how to re-order variables to minimize fill-in in LU
  - this is graph theory (nested-disection, minimum degree, Cuthill-McKee, . . .)
- how does the FFT actually work?
  - why is it $O(m \log m)$?
  - what else is it good for? (signal/image processing, filters, fast Poisson, . . .)
- Krylov methods
  - conjugate gradient, GMRES, . . .
  - matrix-free Krylov
- matrix-based preconditioners for Krylov methods
  - incomplete LU, incomplete Cholesky, . . .
- multigrid for PDE problems
  - geometric multigrid
  - algebraic multigrid, a black-box-ish preconditioner

references

G. Golub & C. van Loan (2013). *Matrix Computations*, 4th ed., Johns Hopkins University Press, Baltimore

- algorithms, banded & circulant matrices, sparse storage

L. Trefethen & D. Bau (2022). *Numerical Linear Algebra*, 25th anniversary ed., SIAM Press, Philadelphia

- clear thinking on matrices and core algorithms