Lecture 3
Scientific Computing: Numerical Linear Algebra

Matthew J. Zahr

CME 292
Advanced MATLAB for Scientific Computing
Stanford University

10th April 2014

1. Dense vs. Sparse Matrices
2. Direct Solvers and Matrix Decompositions
    - Background
    - Types of Matrices
    - Matrix Decompositions
    - Backslash
3. Spectral Decompositions
4. Iterative Solvers
    - Preconditioners
    - Solvers

Outline

1. Dense vs. Sparse Matrices
2. Direct Solvers and Matrix Decompositions
   - Background
   - Types of Matrices
   - Matrix Decompositions
   - Backslash
3. Spectral Decompositions
4. Iterative Solvers
   - Preconditioners
   - Solvers

Assignment

- Create the following matrix (1000 rows/columns)

$$\mathtt{A} = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}$$

- Then, run the following lines of code

```
>> s = who('A');
>> s.bytes
```

- How much storage does your matrix need?

Sparse matrix storage formats

- Sparse matrix = matrix with relatively small number of non zero entries, compared to its size.
- Let $A \in \mathbb{R}^{m \times n}$ be a sparse matrix with n_z nonzeros.
- Dense storage requires mn entries.

Sparse matrix storage formats (continued)

- Triplet format
  - Store nonzero values and corresponding row/column
  - Storage required $= 3n_z$ ($2n_z$ `ints` and n_z `doubles`)
  - Simplest but most inefficient storage format
  - General in that no assumptions are made about sparsity structure
  - Used by MATLAB (column-wise)

$$\begin{bmatrix} 1 & 9 & 0 & 0 & 1 \\ 8 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 5 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix}$$

Sparse matrix storage formats (continued)

- Triplet format
    - Store nonzero values and corresponding row/column
    - Storage required $= 3n_z$ ($2n_z$ `ints` and n_z `doubles`)
    - Simplest but most inefficient storage format
    - General in that no assumptions are made about sparsity structure
    - Used by MATLAB (column-wise)

$$\begin{bmatrix} 1 & 9 & 0 & 0 & 1 \\ 8 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 5 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix}$$

$$\texttt{row} = \begin{bmatrix} 1 & 2 & 1 & 2 & 5 & 3 & 3 & 4 & 1 & 5 \end{bmatrix}$$

$$\texttt{col} = \begin{bmatrix} 1 & 1 & 2 & 2 & 2 & 3 & 4 & 4 & 5 & 5 \end{bmatrix}$$

$$\texttt{val} = \begin{bmatrix} 1 & 8 & 9 & 2 & 4 & 3 & 5 & 7 & 1 & 1 \end{bmatrix}$$

Other sparse storage formats

- Compressed Sparse Row (CSR) format
  - Store nonzero values, corresponding column, and pointer into value array corresponding to first nonzero in each row
  - Storage required $= 2n_z + m$

Break-even point for sparse storage

- For $A \in \mathbb{R}^{m \times n}$ with n_z nonzeros, there is a value of n_z where sparse vs dense storage is more efficient.
- For the triplet format, the cross-over point is defined by $3n_z = mn$
- Therefore, if $n_z \leq \frac{mn}{3}$ use sparse storage, otherwise use dense format
- Cross-over point depends not only on m, n, n_z but also on the data types of `row`, `col`, `val`
- Storage efficiency not only important consideration
  - Data access for linear algebra applications
  - Ability to exploit symmetry in storage

Create Sparse Matrices

- Allocate space for $m \times n$ sparse matrix with n_z nnz
  - `S = spalloc(`m, n, n_z`)`
- Convert full matrix A to sparse matrix S
  - `S = sparse(A)`
- Create $m \times n$ sparse matrix with spare for n_z nonzeros from triplet (`row,col,val`)
  - `S = spalloc(row,col,val,`m, n, n_z`)`
- Create matrix of 1s with sparsity structure defined by sparse matrix S
  - `R = spones(S)`
- Sparse identity matrix of size $m \times n$
  - `I = speye(`m, n`)`

Create Sparse Matrices

- Create sparse uniformly distributed random matrix
  - From sparsity structure of sparse matrix S
    - `R = sprand(S)`
  - Matrix of size $m \times n$ with approximately $mn\rho$ nonzeros and condition number roughly κ (sum of rank 1 matrices)
    - `R = sprand`$(m, n, \rho, \kappa^{-1})$
- Create sparse normally distributed random matrix
  - `R = sprandn(S)`
  - `R = sprandn`$(m, n, \rho, \kappa^{-1})$
- Create sparse symmetric uniformly distributed random matrix
  - `R = sprandn(S)`
  - `R = sprandn`$(m, n, \rho, \kappa^{-1})$
- Import from sparse matrix external format
  - `spconvert`

Create Sparse Matrices (continued)

- Create sparse matrices from diagonals (`spdiags`)
  - Far superior to using `diags`
    - More general
    - Doesn't require creating unnecessary zeros
  - Extract nonzero diagonals from matrix
    - `[B,d] = spdiags(A)`
  - Extract diagonals of A specified by d
    - `B = spdiags(A,d)`
  - Replaces the diagonals of A specified by d with the columns of B
    - `A = spdiags(B,d,A)`
  - Create an $m \times n$ sparse matrix from the columns of B and place them along the diagonals specified by d
    - `A = spdiags(B,d,m,n)`

Assignment

- Create the following matrix (1000 rows/columns)

$$\mathtt{A} = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}$$

  using `spdiags`
- Then, run the following lines of code

```
>> s = who('A');
>> s.bytes
```

- How much storage does your matrix need?

Sparse storage information

Let $S \in \mathbb{R}^{m \times n}$ sparse matrix

- Determine if matrix is stored in sparse format
  - `issparse(S)`
- Number of nonzero matrix elements
  - `nz = nnz(S)`
- Amount of nonzeros allocated for nonzero matrix elements
  - `nzmax(S)`
- Extract nonzero matrix elements
  - If (`row`, `col`, `val`) is sparse triplet of S
  - `val = nonzeros(S)`
  - `[row,col,val] = find(S)`

Sparse and dense matrix functions

Let $S \in \mathbb{R}^{m \times n}$ sparse matrix

- Convert sparse matrix to dense matrix
  - `A = full(S)`
- Apply function (described by function handle `func`) to nonzero elements of sparse matrix
  - `F = ...`
    `spfun(func, S)`
  - Not necessarily the same as `func(S)`
    - Consider `func = @exp`
- Plot sparsity structure of matrix

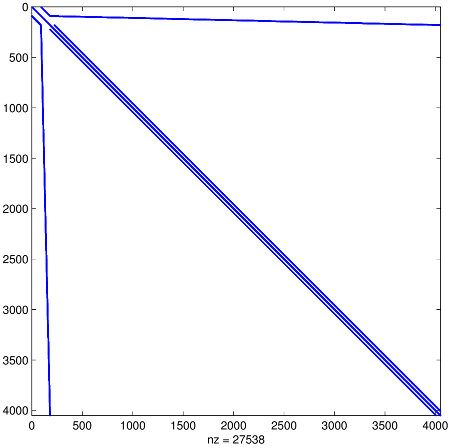


Figure : `spy` plot

Reordering Functions

Command	Description
`amd`	Approximate minimum degree permutation
`colamd`	Column approximate minimum degree permutation
`colperm`	Sparse column permutation based on nonzero count
`dmperm`	Dulmage-Mendelsohn decomposition
`randperm`	Random permutation
`symamd`	Symmetric approximate minimum degree permutation
`symrcm`	Sparse reverse Cuthill-McKee ordering

Sparse Matrix Tips

- Don't change sparsity structure (pre-allocate)
  - Dynamically grows triplet
  - Each component of triplet must be stored *contiguously*
- Accessing values (may be) slow in sparse storage as location of row/columns is not predictable
  - If `S(i,j)` requested, must search through `row`, `col` to find `i`, `j`
- Component-wise indexing to assign values is expensive
  - Requires accessing into an array
  - If `S(i,j)` previously zero, then `S(i,j)= c` changes sparsity structure

Rank

- Rank of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$
  - Defined as the number of linearly independent columns
  - $\text{rank } \mathbf{A} \leq \min\{m, n\}$
  - Full rank $\implies$ $\text{rank } \mathbf{A} = \min\{m, n\}$
  - MATLAB: `rank`
    - Rank determined using SVD

```
>> [rank(rand(100,34)), rank(rand(100,1)*rand(1,34))]
ans =
    34    1
```

Norms

- Gives some notion of size/distance
- Defined for both vectors and matrices
- Common examples for vector, $\mathbf{v} \in \mathbb{R}^m$
  - 2-norm: $||\mathbf{v}||_2 = \sqrt{\mathbf{v}^T \mathbf{v}}$
  - p-norm: $||\mathbf{v}||_p = \left(\sum_{i=1}^m |\mathbf{v}_i|^p\right)^{1/p}$
  - ∞-norm: $||\mathbf{v}||_\infty = \max |\mathbf{v}_i|$
  - MATLAB: `norm(X,type)`
- Common examples for matrices, $\mathbf{A} \in \mathbb{R}^{m \times n}$
  - 2-norm: $||\mathbf{A}||_2 = \sigma_{\max}(\mathbf{A})$
  - Frobenius-norm: $||\mathbf{A}||_{\mathrm{F}} = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |\mathbf{A}_{ij}|^2}$
- MATLAB: `norm(X,type)`
  - Result depends on whether `X` is vector or matrix and on value of `type`
- MATLAB: `normest`
  - Estimate matrix 2-norm
  - For sparse matrices or large, full matrices

Outline

1. Dense vs. Sparse Matrices
2. Direct Solvers and Matrix Decompositions
   - Background
   - Types of Matrices
   - Matrix Decompositions
   - Backslash
3. Spectral Decompositions
4. Iterative Solvers
   - Preconditioners
   - Solvers

Determined System of Equations

Solve linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

by factorizing $\mathbf{A} \in \mathbb{R}^{n \times n}$

- For a general matrix, $\mathbf{A}$, (1) is difficult to solve
- If $\mathbf{A}$ can be decomposed as $\mathbf{A} = \mathbf{B}\mathbf{C}$ then (1) becomes

$$\begin{aligned} \mathbf{B}\mathbf{y} &= \mathbf{b} \\ \mathbf{C}\mathbf{x} &= \mathbf{y} \end{aligned} \tag{2}$$

- If $\mathbf{B}$ and $\mathbf{C}$ are such that (2) are easy to solve, then the difficult problem in (1) has been reduced to two easy problems
- Examples of types of matrices that are "easy" to solve with
  - Diagonal, triangular, orthogonal

Overdetermined System of Equations

Solve the linear least squares problem

$$\min \ \frac{1}{2}||\mathbf{A}\mathbf{x}-\mathbf{b}||_2^2. \tag{3}$$

Define

$$f(\mathbf{x}) = \frac{1}{2}||\mathbf{A}\mathbf{x}-\mathbf{b}||_2^2 = \frac{1}{2}\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{A}\mathbf{x} + \frac{1}{2}\mathbf{b}^T\mathbf{b}$$

Optimality condition: $\nabla f(\mathbf{x}) = 0$ leads to normal equations

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b} \tag{4}$$

Define pseudo-inverse of matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ as

$$\mathbf{A}^\dagger = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T \in \mathbb{R}^{n\times m} \tag{5}$$

Then,

$$\mathbf{x} = \mathbf{A}^\dagger\mathbf{b} \tag{6}$$

Diagonal Matrices

$$\begin{bmatrix} \alpha_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & \alpha_2 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \alpha_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \alpha_{n-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & \alpha_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

$$x_j = \frac{b_j}{\alpha_j}$$

Triangular Matrices

$$\begin{bmatrix} \alpha_1 & 0 & 0 & \cdots & 0 & 0 \\ \beta_1 & \alpha_2 & 0 & \cdots & 0 & 0 \\ \times & \beta_2 & \alpha_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \times & \times & 0 & \cdots & \alpha_{n-1} & 0 \\ \times & \times & \times & \cdots & \beta_{n-1} & \alpha_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

- Solve by forward substitution
  - $x_1 = \frac{b_1}{\alpha_1}$
  - $x_2 = \frac{b_2 - \beta_1 x_1}{\alpha_2}$
  - $\cdots$
- For upper triangular matrices, solve by backward substitution

Additional Matrices

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$

- Symmetric matrix (only for $m = n$)
  - $\mathbf{A} = \mathbf{A}^T$ (transpose)
- Orthogonal matrix
  - $\mathbf{A}^T\mathbf{A} = \mathbf{I}_n$
  - If $m = n$: $\mathbf{A}\mathbf{A}^T = \mathbf{I}_m$
- Symmetric Positive Definite matrix (only for $m = n$)
  - $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \in \mathbb{R}^m$
  - All real, positive eigenvalues
- Permutation matrix (only for $m = n$), $\mathbf{P}$
  - Permutation of rows or columns of identity matrix by permutation vector $\mathbf{p}$
  - For any matrix $\mathbf{B}$, $\mathbf{PB} = \mathbf{B}(\mathbf{p}, :)$ and $\mathbf{BP} = \mathbf{B}(:, \mathbf{p})$

LU Decomposition

Let $\mathbf{A} \in \mathbb{R}^{m \times m}$ be a non-singular matrix.

$$\mathbf{A} = \mathbf{L}\mathbf{U} \tag{7}$$

where $\mathbf{L} \in \mathbb{R}^{m \times m}$ lower triangular and $\mathbf{U} \in \mathbb{R}^{m \times m}$ upper triangular.

LU Decomposition

Let $\mathbf{A} \in \mathbb{R}^{m \times m}$ be a non-singular matrix.

- Gaussian elimination transforms a full linear system into upper triangular one by multiplying (on the left) by a sequence of lower triangular matrices

$$\underbrace{\mathbf{L}_k \cdots \mathbf{L}_1}_{\mathbf{L}^{-1}} \mathbf{A} = \mathbf{U}$$

- After re-arranging, written as

$$\mathbf{A} = \mathbf{L}\mathbf{U} \tag{8}$$

where $\mathbf{L} \in \mathbb{R}^{m \times m}$ lower triangular and $\mathbf{U} \in \mathbb{R}^{m \times m}$ upper triangular.

LU Decomposition - Pivoting

- Gaussian elimination is unstable without pivoting
  - Partial pivoting: $\mathbf{PA} = \mathbf{LU}$
  - Complete pivoting: $\mathbf{PAQ} = \mathbf{LU}$
- Operation count: $\frac{2}{3}m^3$ flops (without pivoting)
- Useful in solving determined linear system of equations, $\mathbf{Ax} = \mathbf{b}$
  - Compute $\mathbf{LU}$ factorization of $\mathbf{A}$
  - Solve $\mathbf{Ly} = \mathbf{b}$ using forward substitution $\implies \mathbf{y}$
  - Solve $\mathbf{Ux} = \mathbf{y}$ using backward substitution $\implies \mathbf{x}$

Theorem

$\mathbf{A} \in \mathbb{R}^{n \times n}$ *has an* $\mathbf{LU}$ *factorization if* $\det \mathbf{A}(1:k, 1:k) \neq 0$ *for* $k \in \{1, \ldots, n-1\}$. *If the* $\mathbf{LU}$ *factorization exists and* $\mathbf{A}$ *is nonsingular, then the* $\mathbf{LU}$ *factorization is unique.*

MATLAB LU factorization

- **LU** factorization, partial pivoting applied to **L**
  - `[L,U] = lu(A)`
    - $\mathbf{A} = \left(\mathbf{P}^{-1}\tilde{\mathbf{L}}\right)\mathbf{U} = \mathbf{L}\mathbf{U}$
    - $\mathbf{U}$ upper tri, $\tilde{\mathbf{L}}$ lower tri, $\mathbf{P}$ row permutation
  - `Y = lu(A)`
    - If $\mathbf{A}$ in sparse format, strict lower triangular of $\mathbf{Y}$ contains $\mathbf{L}$ and upper triangular contains $\mathbf{U}$
    - Permutation information lost
- **LU** factorization, partial pivoting **P** explicit
  - `[L,U,P] = lu(A)`
    - $\mathbf{P}\mathbf{A} = \mathbf{L}\mathbf{U}$
  - `[L,U,p] = lu(A,'vector')`
    - $\mathbf{A}(\mathbf{p},:) = \mathbf{L}\mathbf{U}$

MATLAB LU factorization

- **LU** factorization, complete pivoting $\mathbf{P}, \mathbf{Q}$ explicit
  - `[L,U,P,Q] = lu(A)`
    - $\mathbf{PAQ} = \mathbf{LU}$
  - `[L,U,p,q] = lu(A,'vector')`
    - $\mathbf{A}(\mathbf{p}, \mathbf{q}) = \mathbf{LU}$
- Additional `lu` call syntaxes that give
  - Control over pivoting thresholds
  - Scaling options
  - Calls to UMFPACK vs LAPACK

In-Class Assignment

Use the starter code (`starter_code.m`) below to:

- Compute LU decomposition of using `[L,U] = lu(A);`
  - Generate a spy plot of `L` and `U`
  - Are they both triangular?
- Compute LU decomposition with partial pivoting
  - Create spy plot of `P*A` (or `A(p,:)`), `L`, `U`
- Compute LU decomposition with complete pivoting
  - Create spy plot of `P*A*Q` (or `A(p,q)`), `L`, `U`

```
load matrix1.mat
A = sparse(linsys.row,linsys.col,linsys.val);
b = linsys.b;
clear linsys;
```

Symmetric, Positive Definite (SPD) Matrix

Let $\mathbf{A} \in \mathbb{R}^{m \times m}$ be a symmetric matrix ($\mathbf{A} = \mathbf{A}^T$), then $\mathbf{A}$ is called *symmetric, positive definite* if

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \qquad \forall\, \mathbf{x} \in \mathbb{R}^m.$$

It is called symmetric, positive semi-definite if $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{R}^m$.

Cholesky Factorization

Let $\mathbf{A} \in \mathbb{R}^{m \times m}$ be symmetric positive definite.

- Hermitian positive definite matrices can be decomposed into triangular factors twice as quickly as general matrices
- Cholesky Factorization
  - A variant of Gaussian elimination ($\mathbf{LU}$) that operations on both left and right of the matrix simultaneously
  - Exploits and preserves symmetry

The Cholesky factorization can be written as

$$\mathbf{A} = \mathbf{R}^* \mathbf{R} = \mathbf{L} \mathbf{L}^*$$

where $\mathbf{R} \in \mathbb{R}^{m \times m}$ upper tri and $\mathbf{L} \in \mathbb{R}^{m \times m}$ lower tri.

Theorem

Every hermitian positive definite matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$ has a unique Cholesky factorization. The converse also holds.

Cholesky Decomposition

- Cholesky decomposition algorithm
  - Symmetric Gaussian elmination
- Operation count: $\frac{1}{3}m^3$ flops
- Storage required $\leq \frac{m(m+1)}{2}$
  - Depends on sparsity
- Always stable and pivoting unnecessary
  - Largest entry in $\mathbf{R}$ or $\mathbf{L}$ factor occurs on diagonal
- Pre-ordering algorithms to reduce the amount of fill-in
  - In general, factors of a sparse matrix are dense
  - Pre-ordering attempts to minimize the sparsity structure of the matrix factors
  - Columns or rows permutations applied *before* factorization (in contrast to pivoting)
- Most efficient decomposition for SPD matrices
  - Partial and modified Cholesky algorithms exist for non-SPD
  - Usually just apply Cholesky until problem encountered

Check for symmetric, positive definiteness

For a matrix $\mathbf{A}$, it is not possible to check $\mathbf{x}^T\mathbf{A}\mathbf{x}$ for all $\mathbf{x}$. How does one check for SPD?

- Eigenvalue decomposition

Theorem

If $\mathbf{A} \in \mathbb{R}^{m \times m}$ is a symmetric matrix, $\mathbf{A}$ is SPD if and only if all its eigenvalues are positive.

  - *Very expensive/difficult for large matrices*
- Cholesky factorization
  - If a Cholesky decomposition can be successfully computed, the matrix is SPD
  - *Best option*

MATLAB Functions

- Cholesky factorization
  - `R = chol(A)`
    - Return error if $\mathbf{A}$ not SPD
  - `[R,p] = chol(A)`
    - If $\mathbf{A}$ SPD, $p = 0$
    - If $\mathbf{A}$ not SPD, returns Cholesky factorization of upper $p-1 \times p-1$ block
  - `[R,p,S]=chol(A)`
    - Same as previous, except AMD preordering applied
    - Attempt to maximize sparsity in factor
- Sparse incomplete Cholesky (`ichol`, `cholinc`)
  - `R = cholinc(A,droptol)`
- Rank 1 update to Cholesky factorization
  - Given Cholesky factorization, $\mathbf{R}^T\mathbf{R} = \mathbf{A}$
  - Determine Cholesky factorization of rank 1 update: $\tilde{\mathbf{R}}^T\tilde{\mathbf{R}} = \mathbf{A} + \mathbf{x}\mathbf{x}^T$ using $\mathbf{R}$
  - `R1 = cholupdate(R, x)`

In-Class Assignment

Same starter code (`starter_code.m`) from LU assignment to:

- Compute Cholesky decomposition using `R = chol(A);`
  - Generate a spy plot of `A` and `R`
  - Is `R` triangular?
- Compute Cholesky decomposition *after* reordering the matrix with `p = amd(A)`
  - `Ramd = chol(A(p,p));`
  - Create spy plot of `Ramd`
- Compute incomplete Cholesky decomposition with `cholinc` or `ichol` using drop tolerance of 10^{-2}
  - Create spy plot of `Rinc`
- How do the sparsity pattern and number of nonzeros compare?

QR Factorization

Consider the decomposition of $\mathbf{A} \in \mathbb{R}^{m \times n}$, full rank, as

$$\mathbf{A} = \begin{bmatrix} \mathbf{Q} & \tilde{\mathbf{Q}} \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} = \mathbf{QR} \tag{9}$$

where $\mathbf{Q} \in \mathbb{R}^{m \times n}$ and $\begin{bmatrix} \mathbf{Q} & \tilde{\mathbf{Q}} \end{bmatrix} \in \mathbb{R}^{m \times m}$ are orthogonal and $\mathbf{R} \in \mathbb{R}^{n \times n}$ is upper triangular.

Theorem

Every $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(m \geq n)$ has a QR factorization. If $\mathbf{A}$ is full rank, the decomposition in unique with $\operatorname{diag} \mathbf{R} > 0$.

Full vs. Reduced QR Factorization

$$\mathbf{A} = \begin{bmatrix} \mathbf{Q} & \tilde{\mathbf{Q}} \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} = \mathbf{QR}$$

$$\mathbf{A} = \underbrace{\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix}}_{\begin{bmatrix} \mathbf{Q} & \tilde{\mathbf{Q}} \end{bmatrix}} \underbrace{\begin{pmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}}$$

QR Factorization

- Algorithms for computing QR factorization
  - Gram-Schmidt (numerically unstable)
  - Modified Gram-Schmidt
  - Givens rotations
  - Householder reflections
- Operation count: $2mn^2 - \frac{2}{3}n^3$ flops
- Storage required: $mn + \frac{n(n+1)}{2}$
- May require pivoting in the rank-deficient case

Uses of QR Factorization

Let $\mathbf{A} = \mathbf{QR}$ be the QR factorization of $\mathbf{A}$

- Pseudo-inverse
  - $\mathbf{A}^{\dagger} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T = \left(\mathbf{R}^T\mathbf{R}\right)^{-1}\mathbf{R}^T\mathbf{Q}^T = \mathbf{R}^{-1}\mathbf{Q}^T$
- Solution of least squares
  - $\mathbf{x} = \mathbf{A}^{\dagger}\mathbf{b} = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{b}$
  - Very popular *direct* method for linear least squares
- Solution of linear system of equations
  - $\mathbf{x} = \mathbf{A}^{-1}\mathbf{x} = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{b}$
  - Not best option as $\mathbf{Q} \in \mathbb{R}^{m \times m}$ is dense and $\mathbf{R} \in \mathbb{R}^{m \times m}$
- Extraction of orthogonal basis for column space of $\mathbf{A}$

MATLAB QR function

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$, full rank

- For general matrix, $\mathbf{A}$ (dense or sparse)
  - Full QR factorization
    - `[Q,R] = qr(A)`: $\mathbf{A} = \mathbf{QR}$
    - `[Q,R,E] = qr(A)`: $\mathbf{AE} = \mathbf{QR}$
    - $\mathbf{Q} \in \mathbb{R}^{m \times m}$, $\mathbf{R} \in \mathbb{R}^{m \times n}$, $\mathbf{E} \in \mathbb{R}^{n \times n}$ permutation matrix
  - Economy QR factorization
    - `[Q,R]=qr(A,0)`: $\mathbf{A} = \mathbf{QR}$
    - `[Q,R,E] = qr(A,0)`: $\mathbf{A}(:,\mathbf{E}) = \mathbf{QR}$
    - $\mathbf{Q} \in \mathbb{R}^{m \times n}$, $\mathbf{R} \in \mathbb{R}^{n \times n}$, $\mathbf{E} \in \mathbb{R}^{n}$ permutation vector
- For $\mathbf{A}$ sparse format
  - Q-less QR factorization
    - `R= qr(A), R = qr(A,0)`
  - Least-Squares
    - `[C,R] = qr(A,B), [C,R,E] = qr(A,B),`
      `[C,R] = qr(A,B,0), [C,R,E] = qr(A,B,0)`
    - $\min ||\mathbf{Ax} - \mathbf{b}|| \implies \mathbf{x} = \mathbf{ER}^{-1}\mathbf{C}$

Other MATLAB QR algorithms

Let $\mathbf{A} = \mathbf{QR}$ be the QR factorization of $\mathbf{A}$

- QR of $\mathbf{A}$ with a column/row removed
  - `[Q1,R1] = qrdelete(Q,R,j)`
    - QR of $\mathbf{A}$ with column j removed (without re-computing QR from scratch)
  - `[Q1,R1] = qrdelete(Q,R,j,'row')`
    - QR of $\mathbf{A}$ with row j removed (without re-computing QR from scratch)
- QR of $\mathbf{A}$ with vector $\mathbf{x}$ inserted as jth column/row
  - `[Q1,R1] = qrinsert(Q,R,j,x)`
    - QR of $\mathbf{A}$ with $\mathbf{x}$ inserted in column j (without re-computing QR from scratch)
  - `[Q1,R1] = qrinsert(Q,R,j,x,'row')`
    - QR of $\mathbf{A}$ with $\mathbf{x}$ inserted in row j (without re-computing QR from scratch)

Assignment

Suppose we wish to fit an m degree polynomial, or the form (10) to n data points, (x_i, y_i) for $i = 1, \ldots, n$.

$$a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0 \tag{10}$$

One way to approach this is by solving a linear least squares problem of the form

$$\min ||\mathbf{Va} - \mathbf{y}|| \tag{11}$$

where $\mathbf{x} = [a_m, a_{m-1}, \ldots, a_0]$, $\mathbf{y} = [y_1, \ldots y_n]$, and $\mathbf{V}$ is the Vandermonde matrix

$$\mathbf{V} = \begin{bmatrix} x_1^m & x_1^{m-1} & \cdots & x_1 & 1 \\ x_2^m & x_2^{m-1} & \cdots & x_2 & 1 \\ \vdots & \ddots & \ddots & \vdots & 1 \\ x_n^m & x_n^{m-1} & \cdots & x_n & 1 \end{bmatrix}$$

Assignment

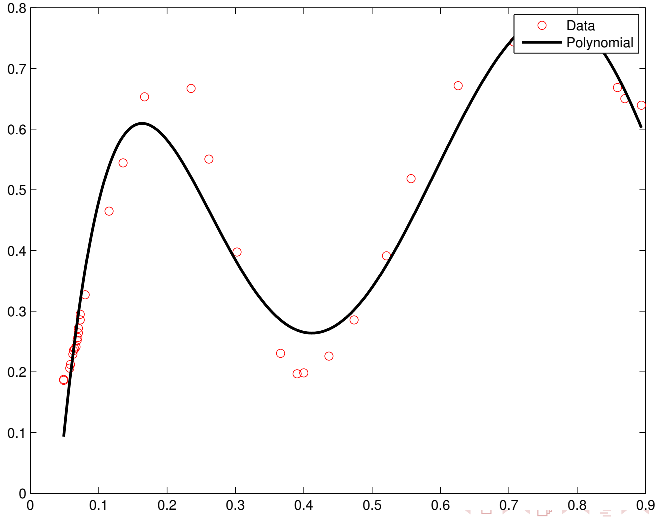
Given the starter code (`qr_ex.m`) below,

- Fit a polynomial of degree 5 to the data in `regression_data.mat`
- Plot the data and polynomial

```
%% QR (regression)
load('regression_data.mat'); %Defines x,y
xfine = linspace(min(x),max(x),1000);
order = 5;

VV = vander(x);
V = VV(:,end-order:end);
```

Assignment


Data
Polynomial
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

De-mystify MATLAB's `mldivide` (\)

- Diagnostics for square matrices
  - Check for triangularity (or permuted triangularity)
    - Check for zeros
    - Solve with substitution or permuted substitution
  - If **A** symmetric with positive diagonals
    - Attempt Cholesky factorization
    - If fails, performs symmetric, indefinite factorization
  - **A** Hessenberg
    - Gaussian elimination to reduce to triangular, then solve with substitution
  - Otherwise, **LU** factorization with partial pivoting
- For rectangular matrices
  - Overdetermined systems solved with **QR** factorization
  - Underdetermined systems, MATLAB returns solution with maximum number of zeros

De-mystify MATLAB's `mldivide` (\)

- Singular (or nearly-singular) *square* systems
  - MATLAB issues a warning
  - For singular systems, least-squares solution may be desired
    - Make system rectangular: $\mathbf{A} \leftarrow \begin{bmatrix} \mathbf{A} \\ \mathbf{0} \end{bmatrix}$ and $\mathbf{b} \leftarrow \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}$
    - From `mldivide` diagnostics, rectangular system immediately initiates least-squares solution
- Multiple Right-Hand Sides (RHS)
  - Given matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and given k RHS, $\mathbf{B} \in \mathbb{R}^{n \times k}$
    - `X = A\B`
    - Superior to `X(:,j)= A\B(:,j)` as matrix only needs to be factorized once, regardless of k
- In summary, *use backslash* to solve $\mathbf{Ax}= \mathbf{b}$ with a direct method

Outline

1. Dense vs. Sparse Matrices
2. Direct Solvers and Matrix Decompositions
   - Background
   - Types of Matrices
   - Matrix Decompositions
   - Backslash
3. Spectral Decompositions
4. Iterative Solvers
   - Preconditioners
   - Solvers

Eigenvalue Decomposition (EVD)

Let $\mathbf{A} \in \mathbb{R}^{m \times m}$, the Eigenvalue Decomposition (EVD) is

$$\mathbf{A} = \mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1} \tag{12}$$

where $\mathbf{\Lambda}$ is a diagonal matrix with the eigenvalues of $\mathbf{A}$ on the diagonal and the columns of $\mathbf{X}$ contain the eigenvectors of $\mathbf{A}$.

Theorem

If $\mathbf{A}$ has distinct eigenvalues, the EVD exists.

Theorem

If $\mathbf{A}$ is hermitian, eigenvectors can be chosen to be orthogonal.

Eigenvalue Decomposition (EVD)

- Only defined for square matrices
  - Does not even exist for all square matrices
    - *Defective* - EVD does not exist
    - *Diagonalizable* - EVD exists
- All EVD algorithms *must* be iterative
- Eigenvalue Decomposition algorithm
  - Reduction to upper Hessenberg form (upper tri + subdiag)
  - Iterative transform upper Hessenberg to upper triangular
- Operation count: $\mathcal{O}(m^3)$
- Storage required: $m(m+1)$
- Uses of EVD
  - Matrix powers ($\mathbf{A}^k$) and exponential ($e^{\mathbf{A}}$)
  - Stability/perturbation analysis

MATLAB EVD algorithms (`eig` and `eigs`)

- Compute eigenvalue decomposition of $\mathbf{AX} = \mathbf{XD}$
  - Eigenvalues only: `d = eig(X)`
  - Eigenvalues and eigenvectors: `[X,D] = eig(X)`
- `eig` also used to computed generalized EVD: $\mathbf{Ax} = \lambda\mathbf{Bx}$
  - `E = eig(A,B)`
  - `[V,D] = eig(A,B)`
- Use ARPACK to find largest eigenvalues and corresponding eigenvectors (`eigs`)
  - By default returns 6 largest eigenvalues/eigenvectors
  - Same calling syntax as `eig` (or EVD and generalized EVD)
  - `eigs(A,k)`, `eigs(A,B,k)` for k largest eigenvalues/eigenvectors
  - `eigs(A,k,sigma)`, `eigs(A,B,k,sigma)`
    - If `sigma` a number, e-vals closest to `sigma`
    - If `'LM'` or `'SM'`, e-vals with largest/smallest e-vals

Singular Value Decomposition (SVD)

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ have rank r. The SVD of $\mathbf{A}$ is

$$\mathbf{A} = \begin{bmatrix} \mathbf{U} & \tilde{\mathbf{U}} \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V} & \tilde{\mathbf{V}} \end{bmatrix}^* = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{13}$$

where $\mathbf{U} \in \mathbb{R}^{m \times r}$ and $\tilde{\mathbf{U}} \in \mathbb{R}^{m \times (m-r)}$ orthogonal, $\mathbf{\Sigma} \in \mathbb{R}^{r \times r}$ diagonal with real, positive entries, and $\mathbf{V} \in \mathbb{R}^{n \times r}$ and $\tilde{\mathbf{V}} \in \mathbb{R}^{n \times (n-r)}$ orthogonal.

Theorem

Every matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has a singular value decomposition. The singular values $\{\sigma_j\}$ are uniquely determined, and, if $\mathbf{A}$ is square and the σ_j are distinct, the left and right singular vectors $\{\mathbf{u}_j\}$ and $\{\mathbf{v}_j\}$ are uniquely determined up to complex signs.

Full vs. Reduced SVD

$$\mathbf{A} = \begin{bmatrix} \mathbf{U} & \tilde{\mathbf{U}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V} & \tilde{\mathbf{V}} \end{bmatrix}^* = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$$

$$\mathbf{A} = \underbrace{\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix}}_{\begin{bmatrix} \mathbf{U} & \tilde{\mathbf{U}} \end{bmatrix}} \underbrace{\begin{pmatrix} \times & 0 & 0 \\ 0 & \times & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\begin{bmatrix} \boldsymbol{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}} \underbrace{\begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix}}_{\begin{bmatrix} \mathbf{V}^* \\ \tilde{\mathbf{V}}^T \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- SVD algorithm
  - Bi-diagonalization of $\mathbf{A}$
  - Iteratively transform bi-diagonal to diagonal
- Operation count (depends on outputs desired):
  - Full SVD: $4m^2n + 8mn^2 + 9n^3$
  - Reduced SVD: $14mn^2 + 8n^3$
- Storage for SVD of $\mathbf{A}$ of rank r
  - Full SVD: $m^2 + n^2 + r$
  - Reduced SVD: $(m + n + 1)r$
- Applications
  - Low-rank approximation (compression)
  - Pseudo-inverse/Least-squares
  - Rank determination
  - Extraction of orthogonal subspace for range and null space

MATLAB SVD algorithm

- Compute SVD of $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^* \in \mathbb{R}^{m \times n}$
  - Singular vales only: `s = svd(A)`
  - Full SVD: `[U,S,V] = svd(A)`
  - Reduced SVD
    - `[U,S,V] = svd(A,0)`
    - `[U,S,V] = svd(A,'econ')`
    - Equivalent for $m \geq n$
- `[U,V,X,C,S] = gsvd(A,B)` to compute generalized SVD
  - $\mathbf{A} = \mathbf{U}\mathbf{C}\mathbf{X}^*$
  - $\mathbf{B} = \mathbf{V}\mathbf{S}\mathbf{X}^*$
  - $\mathbf{C}^*\mathbf{C} + \mathbf{S}^*\mathbf{S} = \mathbf{I}$
- Use ARPACK to find largest singular values and corresponding singular vectors (`svds`)
  - By default returns 6 largest singular values/vectors
  - Same calling syntax as `eig` (or EVD and generalized EVD)
  - `svds(A,k)` for k largest singular values/vectors
  - `svds(A,k,sigma)`

Condition Number, κ

- The condition number of a matrix, $\mathbf{A} \in \mathbb{R}^{m \times n}$, is defined as

$$\kappa = \frac{\sigma_{\max}}{\sigma_{\min}} = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}} \tag{14}$$

  where $\sigma_{\min}$ and $\sigma_{\max}$ are the smallest and largest singular vales of $\mathbf{A}$ and $\lambda_{\min}$ and $\lambda_{\max}$ are the smallest and largest eigenvalues of $\mathbf{A}^T\mathbf{A}$.
- $\kappa = 1$ for orthogonal matrices
- $\kappa = \infty$ for singular matrices
- A matrix is *well-conditioned* for κ close to 1; *ill-conditioned* for κ large
  - `cond`: returns 2-norm condition number
  - `condest`: lower bound for 1-norm condition number
  - `rcond`: LAPACK estimate of inverse of 1-norm condition number (estimate of $||A^{-1}||_1$)

Outline

1. Dense vs. Sparse Matrices
2. Direct Solvers and Matrix Decompositions
   - Background
   - Types of Matrices
   - Matrix Decompositions
   - Backslash
3. Spectral Decompositions
4. Iterative Solvers
   - Preconditioners
   - Solvers

Iterative Solvers

Consider the linear system of equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{15}$$

where $\mathbf{A} \in \mathbb{R}^{m \times m}$, nonsingular.

- Direct solvers
  - $\mathcal{O}(m^3)$ operations required
  - $\mathcal{O}(m^2)$ storage required (depends on sparsity)
  - Factorization of sparse matrix not necessarily sparse
  - Not practical for large-scale matrices
  - Factorization only needs to be done once, regardless of $\mathbf{b}$
- Iterative solvers
  - Solve linear system of equations iteratively
  - $\mathcal{O}(m^2)$ operations required, $\mathcal{O}(nnz(\mathbf{A}))$ storage
  - *Do not need entire matrix* $\mathbf{A}$, only products $\mathbf{A}\mathbf{v}$
  - Preconditioning usually required to keep iterations low
    - Intended to modify matrix to improve condition number

Preconditioning

Suppose $\mathbf{L} \in \mathbb{R}^{m \times m}$ and $\mathbf{R} \in \mathbb{R}^{m \times m}$ are *easily* invertible.

- Preconditioning replaces the original problem ($\mathbf{A}\mathbf{x} = \mathbf{b}$) with a different problems with the same (or similar) solution.
  - Left preconditioning
    - Replace system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ with

$$\mathbf{L}^{-1}\mathbf{A}\mathbf{x} = \mathbf{L}^{-1}\mathbf{b} \tag{16}$$

  - Right preconditioning
    - Define $\mathbf{y} = \mathbf{R}\mathbf{x}$

$$\mathbf{A}\mathbf{R}^{-1}\mathbf{y} = \mathbf{b} \tag{17}$$

  - Left and right preconditioning
    - Combination of previous preconditioning techniques

$$\mathbf{L}^{-1}\mathbf{A}\mathbf{R}^{-1}\mathbf{y} = \mathbf{L}^{-1}\mathbf{b} \tag{18}$$

Preconditioners

Preconditioner $\mathbf{M}$ for $\mathbf{A}$ ideally a cheap approximation to $\mathbf{A}^{-1}$, intended to drive condition number, κ, toward 1

Typical preconditioners include

- Jacobi
    - $\mathbf{M} = \text{diag } \mathbf{A}$
- Incomplete factorizations
    - **LU**, Cholesky
    - Level of fill-in (beyond sparsity structure)
        - Fill-in 0 $\implies$ sparsity structure of incomplete factors same as that $\mathbf{A}$ itself
        - Fill-in > 0 $\implies$ incomplete factors more dense that $\mathbf{A}$
        - Higher level of fill-in $\implies$ better preconditioner
        - No restrictions on fill-in $\implies$ exact decomposition $\implies$ perfect preconditioner $\implies$ single iteration to solve $\mathbf{Ax} = \mathbf{b}$

MATLAB preconditioners

Given square matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$

- Jacobi preconditioner
  - Simple implementation: `M = diag(diag(A))`
  - Careful of 0s on the diagonal ($\mathbf{M}$ nonsingular)
    - If $\mathbf{A}_{jj} = 0$, set $\mathbf{M}_{jj} = 1$
  - Sparse storage (use `spdiags`)
  - Function handle that returns $\mathbf{M}^{-1}\mathbf{v}$ given $\mathbf{v}$
- Incomplete factorization preconditioners
  - `[L,U] = ilu(A,SETUP)`, `[L,U,P] = ilu(A,SETUP)`
    - `SETUP`: `TYPE`, `DROPTOL`, `MILU`, `UDIAG`, `THRESH`
    - Most popular and cheapest: no fill-in, ILU(0) (`SETUP.TYPE='nofill'`)
  - `R = cholinc(X,OPTS)`
    - OPTS: DROPTOL, MICHOL, RDIAG
  - `R = cholinc(X,'0')`, `[R,p] = cholinc(X,'0')`
    - No fill-in incomplete Cholesky
    - Two outputs will not raise error for non-SPD matrix

Common Iterative Solvers

- Linear system of equations $\mathbf{Ax} = \mathbf{b}$
  - Symmetric Positive Definite matrix
    - Conjugate Gradients (CG)
  - Symmetric matrix
    - Symmetric LQ Method (SYMMLQ)
    - Minimum-Residual (MINRES)
  - General, Unsymmetric matrix
    - Biconjugate Gradients (BiCG)
    - Biconjugate Gradients Stabilized (BiCGstab)
    - Conjugate Gradients Squared (CGS)
    - Generalized Minimum-Residual (GMRES)
- Linear least-squares $\min \;||\mathbf{Ax} - \mathbf{b}||_2$
  - Least-Squares Minimum-Residual (LSMR)
  - Least-Squares QR (LSQR)

MATLAB Iterative Solvers

- MATLAB's built-in iterative solvers for $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{A} \in \mathbb{R}^{m \times m}$
  - `pcg`, `bicg`, `bicgstab`, `bicgstabl`, `cgs`, `minres`, `gmres`, `lsqr`, `qmr`, `symmlq`, `tmqmr`
- Similar call syntax for each
  - `[x,flag,relres,iter,resvec] = ...`
    `solver(A,b,restart,tol,maxit,M1,M2,x0)`
  - Outputs
    - `x` - attempted solution to $\mathbf{Ax} = \mathbf{b}$
    - `flag` - convergence flag
    - `relres` - relative residual $\frac{||\mathbf{b}-\mathbf{Ax}||}{||\mathbf{b}||}$ at convergence
    - `iter` - number of iterations (inner and outer iterations for certain algorithms)
    - `resvec` - vector of residual norms at each iteration $||\mathbf{b} - \mathbf{Ax}||$, including preconditioners if used $(||\mathbf{M}^{-1}(\mathbf{b} - \mathbf{Ax})||)$

MATLAB Iterative Solvers

- Similar call syntax for each
  - `[x,flag,relres,iter,resvec] = ...`
    `solver(A,b,restart,tol,maxit,M1,M2,x0)`
  - Inputs (only `A`, `b` required, defaults for others)
    - `A` - full or sparse (recommended) square matrix *or* function handle returning $\mathbf{A}\mathbf{v}$ for any $\mathbf{v} \in \mathbb{R}^m$
    - `b` - m vector
    - `restart` - restart frequency (GMRES)
    - `tol` - relative convergence tolerance
    - `maxit` - maximum number of iterations
    - `M1, M2` - full or sparse (recommended) preconditioner matrix *or* function handler returning $\mathbf{M}_2^{-1}\mathbf{M}_1^{-1}\mathbf{v}$ for any $\mathbf{v} \in \mathbb{R}^m$ (can specify only $\mathbf{M}_1$ or not precondition system by not specifying `M1,M2` or setting `M1 = []` and `M2=[]`)
    - `x0` - initial guess at solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$

Assignment

`iterative_ex.m`