CME292: Advanced MATLAB for Scientific Computing

Homework #2
NLA & Opt

Due: Tuesday, April 28, 2015

Instructions

This problem set will be combined with Homework #3. For this combined problem set, 2 out of the 6 problems are required. You are free to choose the problems you complete.

Before completing problem set, please see `HomeworkInstructions` *on Coursework for general homework instructions, grading policy, and number of required problems per problem set.*

Problem 1

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ can be compressed using the low-rank approximation property of the SVD. The approximation algorithm is given in Algorithm 1.

Algorithm 1 Low-Rank Approximation using SVD

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$ of rank r and approximation rank k ($k \leqslant r$)
Output: $\mathbf{A}_k \in \mathbb{R}^{m \times n}$, optimal rank k approximation to $\mathbf{A}$
1: Compute (thin) SVD of $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, where $\mathbf{U} \in \mathbb{R}^{m \times r}$, $\mathbf{\Sigma} \in \mathbb{R}^{r \times r}$, $\mathbf{V} \in \mathbb{R}^{n \times r}$
2: $\mathbf{A}_k = \mathbf{U}(:, 1 : k)\mathbf{\Sigma}(1 : k, 1 : k)\mathbf{V}(:, 1 : k)^T$

Notice the low-rank approximation, $\mathbf{A}_k$, and the original matrix, $\mathbf{A}$, are the same size. To actually achieve compression, the truncated singular factors, $\mathbf{U}(:, 1 : k) \in \mathbb{R}^{m \times k}$, $\mathbf{\Sigma}(1 : k, 1 : k) \in \mathbb{R}^{k \times k}$, $\mathbf{V}(:, 1 : k) \in \mathbb{R}^{n \times k}$, should be stored. As $\mathbf{\Sigma}$ is *diagonal*, the required storage is $(m+n+1)k$ doubles. The original matrix requires storing mn doubles. Therefore, the compression is useful only if $k < \frac{mn}{m+n+1}$.

Recall from lecture that the SVD is among the most expensive matrix factorizations in numerical linear algebra. Many SVD approximations have been developed; a particularly interesting one from [1] is given in Algorithm 2. This algorithm computes a rank p approximation to the SVD of a matrix. This approximation rank is *different* than the compression rank from Algorithm 1.

Algorithm 2 Low-Rank Probabilistic SVD Approximation

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$ (usually $n \ll m$), approximation rank p, and number of power iterations q
Output: Approximate SVD of $\mathbf{A} \approx \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$
1: Generate $n \times p$ Gaussian test matrix $\mathbf{\Omega}$
2: Form $\mathbf{Y} = (\mathbf{A}\mathbf{A}^T)^q\mathbf{A}\mathbf{\Omega}$
3: Compute QR factorization of $\mathbf{Y}$: $\mathbf{Y} = \mathbf{Q}\mathbf{R}$
4: Form $\mathbf{B} = \mathbf{Q}^T\mathbf{A}$
5: Compute SVD of $\mathbf{B} = \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^T$
6: Set $\mathbf{U} = \mathbf{Q}\tilde{\mathbf{U}}$

In this problem, you will use the Singular Value Decomposition (SVD) to compress an image of your choosing. You will be given the function `get_rgb.m` that accepts the filename of an image and returns the RGB matrices defining the image, stacked vertically to form a single skinny matrix. Also, the function `plot_image_rgb.m` will take stacked RGB matrix and (optionally) a handle to an `axes` graphics object (default will use `gca`) and plot the corresponding image (the input to `plot_image_rgb.m` is the output of `get_rgb.m`). To "compress" the image, compress the stacked RGB matrix and pass to `plot_image_rgb.m`. Another possibility involves compressing the RGB matrices individually. For this assignment, use the stacked version for the compression.

```
1  function  [RGB] = get_rgb(fname)
2
3  % Parse filename to determine extension
4  if isempty(find(fname=='.',1))
5      error('Filename must include extension.');
6  else
7      ext = fname(find(fname=='.',1,'last')+1:end);
8  end
9
10 A = double(imread(fname,ext));
11 RGB = [A(:,:,1);A(:,:,2);A(:,:,3)];
12
13 end
```

```
1  function  [] = plot_image_rgb(RGB,axHan)
2
3  if nargin < 4, axHan=gca; end
4
5  m = size(RGB,1)/3; n = size(RGB,2);
6  A = zeros(m,n,3);
7  A(:,:,1)=RGB(1:m,:);
8  A(:,:,2)=RGB(m+1:2*m,:);
9  A(:,:,3)=RGB(2*m+1:end,:);
10 image(uint8(A),'Parent',axHan);
11
12 end
```

- Consider an image of $m \times n$ pixels. The `R, G, B` matrices will be of dimension $m \times n$.
- Compress to ranks $[5, 10, 15, 20, 25, 50, 100, 200, \min(3m, n)]$ and visualize resulting images. Your result should be similar to Figure 1 for the image of Palm Drive.
  - Use `svd` for the SVD step in Algorithm 1
  - (extra credit) Implement the probabilistic SVD algorithm from Algorithm 2 and use this for the SVD step in Algorithm 1. Good values for the approximation rank will depend on the image you choose to compress. Notice that a rank p approximation to the SVD in Algorithm 2 only admits low-rank approximations of up to rank p (cannot go up to $\min(3m, n)$). In this case, the ranks $[5, 10, 15, 20, 25, 50, 100, 200, \min(3m, n)]$ cannot all be used. Choose any 9 interesting ranks to present.
- Plot $\nu(k)$ for $k \in \{1, 2, \dots, \min(3m, n)\}$ for the stacked RGB matrix, where

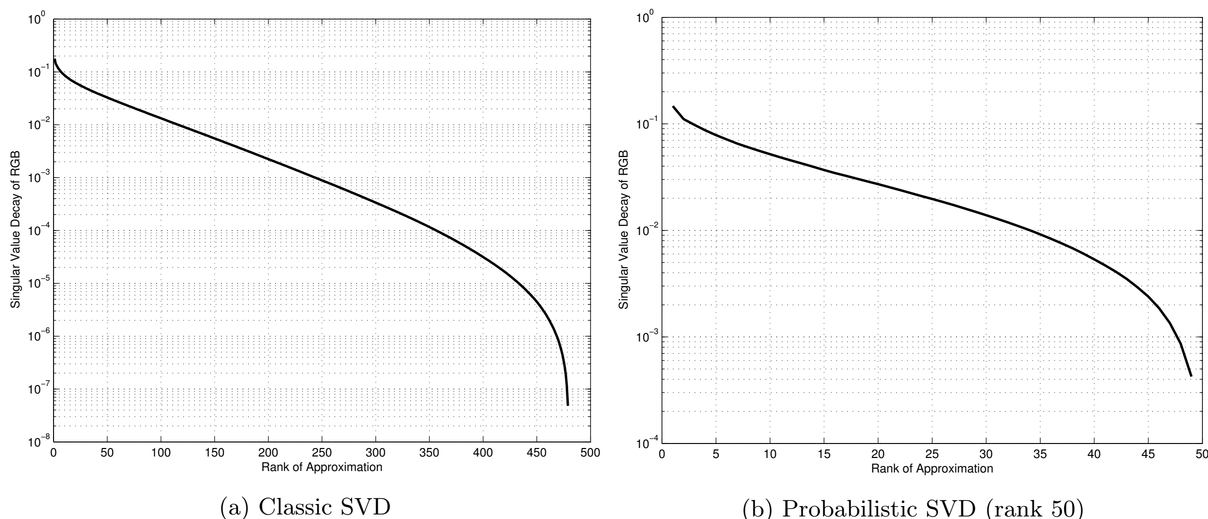
$$\nu(k) = 1 - \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{j=1}^{\min(3m,n)} \sigma_j^2}. \tag{1}$$

Problem 1 continued on next page. . .

and σ_i is the ith singular value of the matrix. The result should be similar to Figure 2a for the image of Palm Drive. This gives an indication of the compressibility of the image. A fast singular value decay implies the image can be compressed to a small rank with minimal loss in quality.

- Use the singular values generated by `svd` to complete this part.
- (extra credit) Repeat using the singular values generated by Algorithm 2.

Figure 1: Classic SVD, Varying degrees of compression



(a) Classic SVD

(b) Probabilistic SVD (rank 50)

Figure 2: Singular value decay

Problem 2

In this problem, you will gain experience with sparse matrices, matrix decompositions, and linear system solvers. You will be given a medium-scale sparse matrix in triplet format `(row, col, val)` as well as a right-hand side vector, `b` (`matrix_medium.mat`). For this matrix, complete the following tasks.

(a) Convert sparse triplet `(row, col, val)` into a matrix `A`

(b) Determine number of nonzeros in `A`

(c) Compute approximate minimum degree permutation of `A`: `p = amd(A)`

(d) Plot sparsity structure of matrix `A`

(e) Compute LU decomposition of `L*U = A(p,p)` and plot sparsity structure of both matrix factors

(f) Determine whether matrix is symmetric positive definite (don't use `eig` as it my be very slow). If matrix is SPD,

- compute Cholesky factorization of `A(p,p)= R'*R` and plot sparsity structure of factor `R`
- compute incomplete Cholesky factorization of `A` → `Rinc` with no fill-in
  - `ichol` or `cholinc` depending on your version of MATLAB
- Plot the sparsity structure of `Rinc`. Briefly comment.

(g) Solve linear system of equations `A*x = b`. Define `M = Rinc'*Rinc`.

- Using `mldivide` (backslash); this will be considered the exact solution of `A*x = b`
- Using the following iterative schemes
  - Conjugate Gradient without preconditioning
  - Preconditioned Conjugate Gradient, preconditioned with `M`
  - MINRES without preconditioning
  - MINRES, preconditioned with `M`

- Create a table containing the CPU time and number of iterations required for convergence of each algorithm and the relative error in converged solution, where (`error = norm(x_backslash - ... x_iterative)/norm(x_backslash)`).
- Create a convergence plot of residual vector norm for each algorithm.

Additionally, you will also be given a large-scale sparse matrix in triplet format (`matrix_large.mat`). Repeat parts (a), (b), (c), (f), (g), (h) for the large matrix. Generate a spy plot for only the matrix itself (not the factors as some may be dense). *Warning: The large matrix will be fairly CPU intensive. I recommend running on corn.*

Problem 3

In this problem, you will write a nonlinear equation solver using Newton-Raphson method. The Newton-Raphson method solves a nonlinear system iteratively by linearizing about the current iterate and finding the root of the linear system. To be explicit, consider the nonlinear function $\mathbf{R} : \mathbb{R}^n \to \mathbb{R}^n$. The goal is to solve $\mathbf{R}(\mathbf{x}) = \mathbf{0}$, given some starting value $\mathbf{x}_0$. The truncated Taylor series of $\mathbf{R}$ about point $\bar{\mathbf{x}}$ is

$$\mathbf{R}(\mathbf{x}) = \mathbf{R}(\bar{\mathbf{x}}) + \frac{\partial \mathbf{R}}{\partial \mathbf{x}}(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + \mathcal{O}(||\mathbf{x} - \bar{\mathbf{x}}||^2). \tag{2}$$

Substituting the truncated Taylor series into the equation $\mathbf{R}(\mathbf{x}) = 0$ we have

$$\mathbf{x} = \bar{\mathbf{x}} - \left[\frac{\partial \mathbf{R}}{\partial \mathbf{x}}(\bar{\mathbf{x}})\right]^{-1} \mathbf{R}(\bar{\mathbf{x}}) \tag{3}$$

Applying (2) iteratively, starting from $\mathbf{x}_0$, until some convergence criteria is met is the Newton-Raphson method for solving nonlinear equations. Convergence is only guaranteed if $\mathbf{x}_0$ is *close* to the actual solution.

- The convergence criteria for the nonlinear iterations is an important consideration for algorithmic speed and reliability. Most convergence criteria are based on the norm of the residual $||\mathbf{R}(\mathbf{x})||$. In this problem, we consider the nonlinear iterations to be converged if *either* of the following are satified:
  - $||\mathbf{R}(\mathbf{x}_n)|| \leqslant \epsilon_{\text{res}}^{\text{rel}} ||\mathbf{R}(\mathbf{x}_0)||$
  - $||\mathbf{R}(\mathbf{x}_n)|| \leqslant \epsilon_{\text{res}}^{\text{abs}}$ AND $||\mathbf{x}_n - \mathbf{x}_{n-1}|| \leqslant \epsilon_{\text{inc}}^{\text{abs}}$

  where $\epsilon_{\text{res}}^{\text{rel}}$, $\epsilon_{\text{res}}^{\text{abs}}$, $\epsilon_{\text{inc}}^{\text{abs}}$ are convergence tolerances. Use the 2-norm for this assignment (`norm(x)`).
- For many situations in computational science, the computation of the Jacobian matrix $\frac{\partial \mathbf{R}}{\partial \mathbf{x}}$ is very expensive and it is desirable to reduce the number of times it must be computed. Therefore variants of Newton's method have considered *freezing* the Jacobian matrix at some iteration and reusing it until an iteration fails to reduce the residual. Namely, the Jacobian matrix is only recomputed when
  - $||\mathbf{R}(\mathbf{x}_k)|| > \epsilon ||\mathbf{R}(\mathbf{x}_{k-1})||$ for $\epsilon \in [0, 1]$
- As Newton's method is *not* guaranteed to converge for an arbitrary starting point, it is good practice to specify a maximum number of nonlinear iterations. If the nonlinear iterations do not converge, a warning message should be issued. It is up to you to decide whether the final iterate $\mathbf{x}_{N_{\max}}$ is an acceptable solution.

Newton-Raphson's method is summarized in Algorithm 3 (without the "frozen" Jacobian feature). It should be emphasized that Algorithm 3 is just pseudo-code, not an optimized algorithm.
In this problem, your tasks are

Algorithm 3 Newton-Raphson Method

Input: $\mathbf{x}_0 \in \mathbb{R}^n$, $\epsilon_{\text{res}}^{\text{rel}}$, $\epsilon_{\text{res}}^{\text{abs}}$, $\epsilon_{\text{inc}}^{\text{abs}}$, $N_{\max}$
Output: $\mathbf{x}^* \in \mathbb{R}^n$ such that $\mathbf{R}(\mathbf{x}^*) \approx \mathbf{0}$

1: **for** $k = 0, 1, 2, ..., N_{\max}$ **do**
2: $\quad \Delta\mathbf{x}_k = -\frac{\partial \mathbf{R}}{\partial \mathbf{x}}(\mathbf{x}_k)^{-1}\mathbf{R}(\mathbf{x}_k)$
3: $\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$
4: $\quad$ **if** $||\mathbf{R}(\mathbf{x}_{k+1})|| \leqslant \epsilon_{\text{res}}^{\text{rel}}||\mathbf{R}(\mathbf{x}_0)||$ or $(||\mathbf{R}(\mathbf{x}_{k+1})|| \leqslant \epsilon_{\text{res}}^{\text{abs}}$ AND $||\mathbf{x}_{k+1} - \mathbf{x}_k|| \leqslant \epsilon_{\text{inc}}^{\text{abs}})$ **then**
5: $\qquad \mathbf{x}^* = \mathbf{x}$
6: $\qquad$ return
7: $\quad$ **end if**
8: $\quad$ Issue convergence warning
9: $\quad \mathbf{x}^* = \mathbf{x}$
10: **end for**

(1) Implement Newton-Raphson's method (Algorithm 3). The input for your Newton-Raphson implementation should be similar to that of `fsolve`, namely, a function handle, a starting point $\mathbf{x}_0$, and various tolerances. The function handle should accept a vector and return two outputs, the nonlinear function and Jacobian evaluated at the input vector. Starter code (`newton_raphson`) has been provided in the code distribution for this assignment. Feel free to use this as a starting point or start from scratch.

(2) Implement the modified Newton-Raphson method that allows for

- freezing the Jacobian matrix until $||\mathbf{R}(\mathbf{x}_k)|| > 0.1||\mathbf{R}(\mathbf{x}_{k-1})||$

(3) Make sure you algorithm returns the following output information (I recommend storing it in a structure, but it is not required)

- the number of nonlinear iterations required for convergence, k
- the residual norm at convergence, $||\mathbf{R}(\mathbf{x}_k)||$
- the number of Jacobian computations required
- the residual norm convergence history, i.e. $||\mathbf{R}(\mathbf{x}_j)||$ for $j \in \{0, \ldots, k\}$

(4) The piece of code below will setup a nonlinear system of equations of dimension 122 and define a function handle whose *first* argument is the nonlinear function and the *second* is the Jacobian. The solution of this nonlinear equation will solve an instance of the nonlinear truss problem defined in Problem 4 of this assignment. *To run this code, the folder `nltruss` must be on your MATLAB search path. Either copy all files in `nltruss` to your current folder or preferably use `addpath('/path/to/nltruss')` to add `nltruss` to your search path.* Starter code provided in `nonlinear_study.m`.

```
1 mesh = 'mesh3';
2 nel = eval([mesh,'()']);
3 [msh,bcs,mat] = setup_problem(0.1*ones(nel,1),mesh);
4 func = @(W) staticResidual(W,msh,mat,bcs);
```

Using the function handle defined above and your Newton-Raphson solver, conduct the following study:

- Solve the nonlinear system using `fsolve`. Turn the `Jacobian` option `on` for a fair comparison with your Newton-Raphson implementation. Make sure the tolerances, `TolX` and `TolFun`, and maximum iterations, `MaxIter`, correspond to those you use below. Record the number of iterations. Assume there is one Jacobian evaluation per iteration.

Problem 3 continued on next page. . .

  - Plot the mesh and its deformation (the solution of the nonlinear system) using `visualize_truss_loads(msh,bcs,mat,x,1)` where `x` is the solution of the nonlinear system of equations (output of `fsolve`)

- Solve the nonlinear system using your Newton-Raphson implementation with the following variations. For each, record the residual norm history, number of iterations required for convergence, and number of Jacobian evaluations.
  - Recompute the Jacobian at every iteration and use backslash to solve the linear system
  - Repeat the above three tasks, but reuse the Jacobian until $||\mathbf{R}(\mathbf{x}_k)|| > 0.1||\mathbf{R}(\mathbf{x}_{k-1})||$
- I recommend all tolerances be set to 10^{-8} and $N_{\max} = 1000$.

(5) Create a table that summarizes the results of the above study, comparing number of nonlinear iterations, residual at convergence, and number of Jacobian evaluations required for each algorithm (3 in total - `fsolve` and your Newton-Raphson solver with and without recycling the Jacobian).

(6) Create a plot of residual norm versus nonlinear iteration for both Newton-Raphson algorithms (not necessary for `fsolve`).

(7) Briefly comment on the results.

Problem 4

In this problem, you will gain experience with MATLAB's optimization functions, in particular `fmincon`. You will be given a general, *nonlinear* 2D truss code that will be used to define the optimization problems. Prior knowledge of truss analysis or mechanics is not necessary. A brief exposition of the truss analysis framework is provided below.

First, we begin with some terminology and nomenclature. A truss structure will be composed of n_v nodes or vertices connected by n_{el} truss elements. Each element e has an initial length L_e, cross-sectional area A_e, elastic modulus E_e, density ρ_e, and deformed length (i.e. after loads applied) ℓ_e. A truss element can only carry *normal* forces, not shear or moments. This means the direction of the force within an element must align with the element. By definition, the intersection of two truss elements will be a *pinned* connection (otherwise the members would carry shear/moments). The force carried by element e will be denoted $\mathbf{N}_e$. Each vertex will be subject to forces that are applied externally or transferred from the truss elements. Let $\mathbf{f}_i^{\text{int}} \in \mathbb{R}^2$ denote the force on vertex i due to the truss elements, usually called the *internal* force. Also, let $\mathbf{f}_i^{\text{ext}} \in \mathbb{R}^2$ denote the force on vertex i due to external loads. Finally, $\mathbf{u}_i \in \mathbb{R}^2$ denotes the displacement of vertex i induced by the loads.

Boundary conditions for truss elements can either be *displacement* or *force* boundary conditions. Force boundary conditions correspond to external loads, while displacement boundary conditions indicate prescribed displacements. Every node must have *exactly one* boundary condition for *each* degree of freedom (i.e. the $x-$ and $y-$directions).

Static equilibrium at the nodes leads to the governing equation

$$\mathbf{f}^{\text{int}}(\mathbf{u}) = \mathbf{f}^{\text{ext}}. \tag{4}$$

There are several important quantities that will be used in this assignment as objective or constraints.

- Stress
$$\sigma_e = E_e \frac{\ell_e}{L_e} \tag{5}$$

- Weight
$$W = \sum_{e=1}^{n_{el}} \rho_e A_e \ell_e \tag{6}$$
- Compliance
$$C = \sum_{i=1}^{n_v} \mathbf{f}_i^T \mathbf{u}_i \tag{7}$$

In the truss program you are given, there will be three relevant data structures.

- `msh` - contains all mesh information
- `bcs` - contains all boundary condition information
- `mat` - contains all material information (including area)

The below piece of code demonstrates how to call use the nonlinear truss code. Pay particular attention to the commands up to `solve_truss` as this describes how to solve the nonlinear truss problem given a new vector of elemental areas. Additional starter code (`optimize_truss.m`) has been provided for you in the code distribution.

```
% Bridge mesh
mesh = 'mesh2';

% Determine number of elements in mesh and define vector of areas
nel = eval([mesh,'()']);
A = ones(nel,1);

% Setup mesh, BCs, and materials
[msh,bcs,mat] = setup_problem(A,mesh);

% Solve nonlinear truss equations (and sensitivity equations)
[U,dUdA,F,dFdA] = solve_truss(msh,mat,bcs);

% Plot mesh and deformation
% Vary last argument to scale deformation
visualize_truss_loads(msh,bcs,mat,U,10);

% Compute deformed lengths and sensitivities
[l,dldA] = compute_element_lengths(U,dUdA,msh)

% Compute weight and sensitivities
[W,dWdA] = compute_weight(U,dUdA,msh,mat);

% Compute compliance and sensitivities
[C,dCdA] = compute_compliance(U,dUdA,F,dFdA);

% Compute element force and sensitivities
[N,dNdA] = compute_internal_force(U,dUdA,msh,mat);

% Compute stress and sensitivities
[s,dsdA] = compute_stress(U,dUdA,msh,mat);
```

(1) Solve the following minimization problem

$$\begin{aligned} \underset{\mathbf{A}\in\mathbb{R}^{n_{el}}}{\text{minimize}} \quad & W(\mathbf{A}) \\ \text{subject to} \quad & |\sigma_e(\mathbf{A})| \leqslant \sigma_{\text{fail}} \\ & 0.01 \leqslant \mathbf{A} \leqslant 1, \end{aligned} \tag{8}$$

for the mesh in Figure 4. Use an initial guess of $\mathbf{A} = 0.1$. This is accomplished using `mesh = ... 'mesh2'` in the code provided above. I recommend starting (and debugging) with `mesh = 'mesh1'` as the corresponding mesh has fewer elements and is faster to solve/optimize (Figure 3). Also, feel free to play around with `nseg` in `mesh2.m`, which controls the number of spans in the bridge.

- Use `fmincon` - try active set, interior point, and SQP algorithms
- Use both user-defined gradients (provided in `nltruss`) and gradients automatically computed with finite differences (`'GradObj'`, `'GradConstr'`).
- Comment on differences in optimal solution and timings.
- Plot the deformed truss with optimal areas (`visualize_truss_loads.m`)

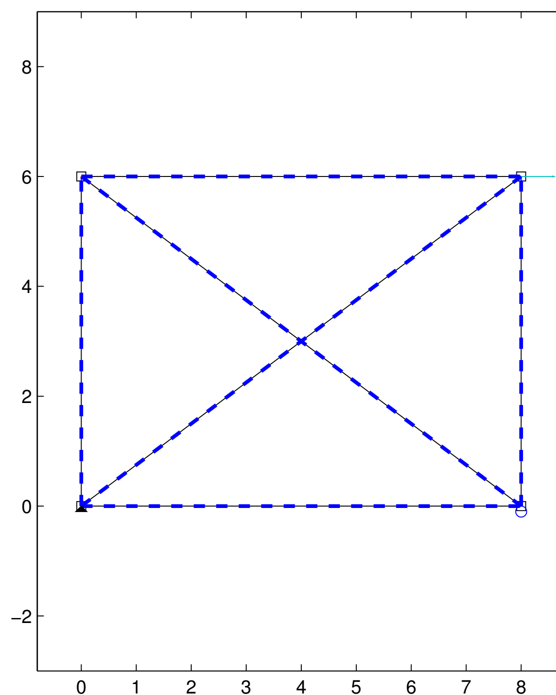


Figure 3: Simple Mesh (`mesh1`)

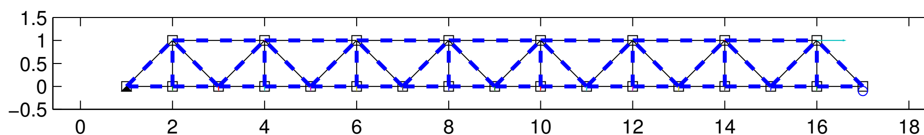


Figure 4: Truss Bridge Mesh (`mesh2`)

(2) Solve the following optimization problem

$$\begin{aligned} \underset{\mathbf{A}\in\mathbb{R}^{n_{el}}}{\text{minimize}} \quad & C(\mathbf{A}) \\ \text{subject to} \quad & W(\mathbf{A}) \leqslant W(\mathbf{A}_0) \\ & |\sigma_e(\mathbf{A})| \leqslant \sigma_{\text{fail}} \\ & 0.01 \leqslant \mathbf{A} \leqslant 1, \end{aligned} \tag{9}$$

where $\mathbf{A}_0 = 0.1$ is the initial guess.

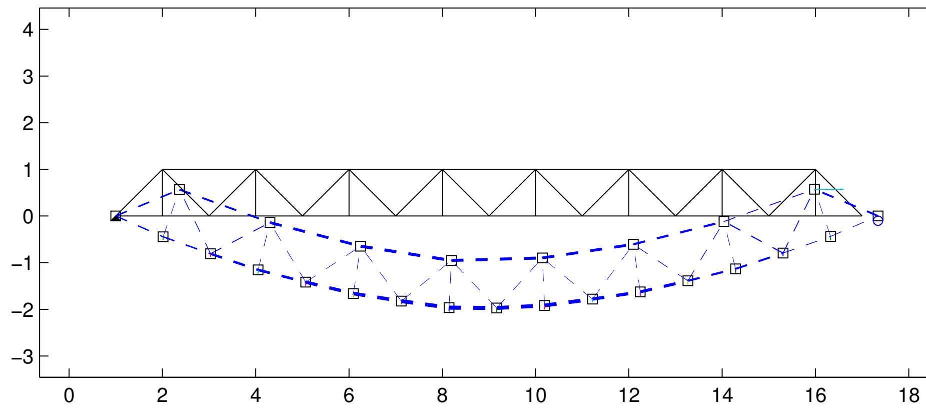


Figure 5: Truss Bridge - Optimized (Min Weight), Deformed (`mesh2`)

- Use `fmincon` - any optimization algorithm you choose
- Use user-defined gradients (provided in `nltruss`) (`'GradObj'`, `'GradConstr'`).
- Plot the deformed truss with optimal areas (`visualize_truss_loads.m`)

References

[1] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.