Homework 2 - Least Squares and Truss Simulation

Handout date: October 28, 2024.
Submission deadline: November 14, 2024, 11:59pm.

This assignment is a little longer than the previous ones and is ***worth 130 points***. However, a large portion of it involves translating the pseudocode from the lecture slides into Python code. It also includes an opportunity for an additional 10 bonus points (Section 2.4).

1 Theory

Problem 1 (10 pts). Consider a Cholesky factorization of the matrix appearing in the normal equations for a least-squares problem involving matrix $A \in \mathbb{R}^{m\times n}$:

$$A^\top A = LL^\top$$

when A is full rank and $m \geq n$ (so that the least-squares solution is unique). Show that the columns of the matrix $AL^{-\top}$ are orthonormal. In other words, this matrix is orthogonal apart from the technicality that orthogonal matrices are square by definition. Next, based on this fact, describe how to construct a reduced QR factorization of A using the Cholesky factorization of $A^\top A$.

Problem 2 (10 pts). Show that the stiffness matrix K defined below in Section 3 is positive *semi*definite provided that the spring constants k_e are all nonnegative. In other words, show that $\mathbf{u}^\top K\mathbf{u} \geq 0$ for all $\mathbf{u} \neq 0$. Finally, find a nonzero vector $\mathbf{u} \neq 0$ so that $\mathbf{u}^\top K\mathbf{u} = 0$. This means that K is singular (and not quite positive definite).

2 Curve Fitting by Solving Least Squares Problems

So far we have considered two different approaches for solving the least-squares problem:

$$\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2 .$$

1) Solving the normal equations $A^\top A\mathbf{x} = A^\top\mathbf{b}$ using the Cholesky factorization.
2) Using the reduced QR factorization $A = \hat{Q}\hat{R}$:

$$\underbrace{\hat{R}^\top\hat{Q}^\top}_{A^\top}\underbrace{\hat{Q}\hat{R}}_{A}\mathbf{x} = \underbrace{\hat{R}^\top\hat{Q}^\top}_{A^\top}\mathbf{b} \quad\Longleftrightarrow\quad \hat{R}\mathbf{x} = \hat{Q}^\top\mathbf{b}$$

Soon we will learn a third approach based on the singular value decomposition. In this part of the assignment, you will compare the two methods above on the problem of polynomial approximation. (curve fitting) by completing the implementations in the newly provided `linear_systems.py`, `curvefit.py`, and `qr.py` files. *First, copy your implementations from the previous assignment's `linear_systems.py` file into the new version.*

Problem 3 (5 pts). Implement the Cholesky factorization algorithm in the `cholesky` function of `linear_systems.py`. *Warning: you may find the indexing of the innermost loop a little tricky to translate into Python; note that the upper limit of `range` is **noninclusive***

Problem 4 (5 pts). Implement the `solve_cholesky` function in `linear_systems.py` by using the `cholesky` function from above to solve the linear system $A\mathbf{x} = \mathbf{b}$ (assuming A is positive definite).

Before proceeding to the next step, you should confirm that your implementation passes all unit tests in `linear_systems.py` by running `python3 linear_systems.py`.

Problem 5 (5 pts). Complete the function `vandermonde_matrix` in `curvefit.py` to build the Vandermonde matrix for polynomials of degree d and the sample coordinates specified in vector $\mathbf{x}$. Note that in the least squares case, this matrix is rectangular, not square; its entries are obtained by evaluating basis monomials x^j at the sample coordinates x_i:

$$V := \begin{bmatrix} 1 & x_1 & \cdots & x_1^d \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \cdots & x_n^d \end{bmatrix}.$$

Problem 6 (5 pts). Complete the function `curvefit.evaluate_polynomial` to evaluate the polynomial with coefficients `a` at the sample points `x`. Assume that the order of the coefficients in `a` matches the order of the columns in your Vandermonde matrix.

Problem 7 (10 pts). Implement the Cholesky factorization approach to solving the least-squares problem "V $\mathbf{a}$= $\mathbf{y}$" in the `curvefit_cholesky` function of `curvefit.py`, which should call your `linear_systems.solve_cholesky` function from above.

Running the command `python3 curvefit.py 5` will generate several data sets, run your curve fitting implementations on each one, and plot the resulting best-fit polynomials for degree 5. Verify that the output plots `result_*_deg_5.pdf` look reasonable.

Problem 8 (5 pts). Run `python3 curvefit.py <DEG> --method Cholesky` to generate plots for the normal-equations-based fit only, where `<DEG>` is the degree of the polynomial you want to fit. Include in your report the plot for the "lecture 8" data set (the points from this slide) for degrees 7 and 10. Also include the degree 8 plot for the "linear" data set. Finally, include the printout of the following command in your report:

```
python3 curvefit.py 12 --example quadratic --method Cholesky
```

This printout reports the coefficients of the best-fit polynomial of degree 12 as well as the relative residual achieved. Comment on how accurate you think the coefficients from the normal equations are. Furthermore, what happens when you try a degree of 15? Report on what you think might be going wrong. *Hint: the Vandermonde matrix becomes nearly rank deficient at high degree.*

2.1 Using the Modified Gram-Schmidt QR Factorization

In this part, you will solve the least-squares problem using the reduced QR factorization as computed by the *modified* Gram-Schmidt algorithm.

Problem 9 (5 pts). Implement the modified Gram-Schmidt algorithm in `qr.modified_gram_schmidt` to compute the reduced QR decomposition $A = \hat{Q}\hat{R}$.

Problem 10 (5 pts). Implement the branch of `qr.least_squares` corresponding to `ModifiedGramSchmidt` to solve the least squares problem. Then run

```
python3 curvefit.py 12 --example quadratic --method ModifiedGramSchmidt
```

and include the printout in your report. How accurate are the coefficients?

Even though the modified Gram-Schmidt algorithm is more stable than classical Gram-Schmidt, it still suffers from loss of orthogonality, especially on ill-conditioned inputs. However, the factorization can be salvaged (see the bonus problem in Section 2.4).

2.2 Using the Householder QR Factorization

In this part, you will solve the least-squares problem using the *full* QR factorization computed by the Householder reflection algorithm.

Problem 11 (15 pts). Implement the Householder reflection algorithm to compute the full QR decomposition $A = QR$ (although with Q represented implicitly as a list of Householder reflection hyperplane normals $\mathbf{v}_k$). You will need to decide how to store these vectors $\mathbf{v}_k$, which are all of different sizes.

The easiest (but not most efficient) approach in Python is to allocate a new vector to hold each $\mathbf{v}_k$ append it to a list as suggested in the pseudocode. Feel free to use this approach if you like. However, notice that lower triangle of the R matrix is unused, and it is therefore tempting to store vector $\mathbf{v}_k$ below the diagonal in column k. The tricky part is that $\mathbf{v}_k$ is of length $m - k$, while there are only $m - k - 1$ spaces *below* the diagonal. One work-around is to use a working matrix that has

$m+1$ rows (one more than the input matrix). Another is to scale the vector $\mathbf{v}_k$ so that its first entry is always 1 (and therefore need not be stored explicitly). This last approach is actually what the standard implementation in LAPACK does, but it slightly complicates the code applying the implied reflectors Q_k.

Problem 12 (15 pts). Compute the right-hand-side vector $\hat{Q}^\top \mathbf{b}$. Since you should have not explicitly constructed the matrix Q, you will need to do this by applying the Householder reflections represented by $\mathbf{v}_k$ to $\mathbf{b}$ one at a time. The result will be $Q^\top \mathbf{b}$, holding $\hat{Q}^\top \mathbf{b}$ in its first n rows.

Finally, solve the linear system $\hat{R}\mathbf{x} = \hat{Q}^\top \mathbf{b}$ using your `backsub` implementation. *Warning: remember that Householder computes a FULL QR factorization, while $\hat{R}$ in this system is the reduced factor.*

Problem 13 (5 pts). Run

```
python3 curvefit.py 12 --example quadratic --method Householder
```

and include the printout in your report. How accurate are the coefficients? Also run `python3 curvefit.py 14 --example quadratic` and include the plot it generates `result_quadratic_deg_14.pdf` in your report. The Cholesky solve is likely to fail, and so its curve is likely missing from this plot.

2.3 Unit Tests

Files `curvefit.py` and `qr.py` each contain unit tests that should thoroughly check your implementations for problems 5-10 above. I recommend running the tests after completing each problem to catch any bugs before moving on, since the implementations build on each other. Run the tests by executing `python3 curvefit.py --test` and `python3 qr.py`, respectively.

2.4 Salvaging the Modified Gram-Schmidt Approach

Bonus (10 pts). The loss-of-orthogonality issues that led to poor results in Section 2.1 can be mitigated with a careful calculation of the right-hand-side vector $\mathbf{y} := \hat{Q}^\top \mathbf{b}$. Specifically, each entry y_i of $\mathbf{y}$ should be computed as the dot product $y_i = \mathbf{q}_i \cdot \mathbf{b}$, and then $y_i \mathbf{q}_i$ should be subtracted from $\mathbf{b}$ so that the component along $\mathbf{q}_i$ does not leak into subsequent entries. This subtraction step is the key to improved stability (without it, we would just be computing $\hat{Q}^\top \mathbf{b}$ directly).

This process is equivalent to applying modified Gram-Schmidt to the augmented system, generating the block QR factorization $\begin{bmatrix}\hat{Q} & \mathbf{q}_{n+1}\end{bmatrix}\begin{bmatrix}\hat{R} & \mathbf{y} \\ 0 & y_{n+1}\end{bmatrix} = \begin{bmatrix}A & \mathbf{b}\end{bmatrix}$.

Implement this "improved" version of the modified Gram-Schmidt least-squares solver in the `ModifiedGramSchmidtImproved` branch of `qr.least_squares` whichever way you see fit. Include the printout of the command

```
python3 curvefit.py 12 --example quadratic --method ModifiedGramSchmidtImproved
```

and comment on its accuracy. Also generate and include `result_linear_deg_10.pdf` (degree 10).

Bonus point policy: your average grade across all homework assignments will be capped to 100% before combining with the final exam, meaning these bonus points cannot be used to make up for points lost on the exam.

3 Application: Truss Simulation

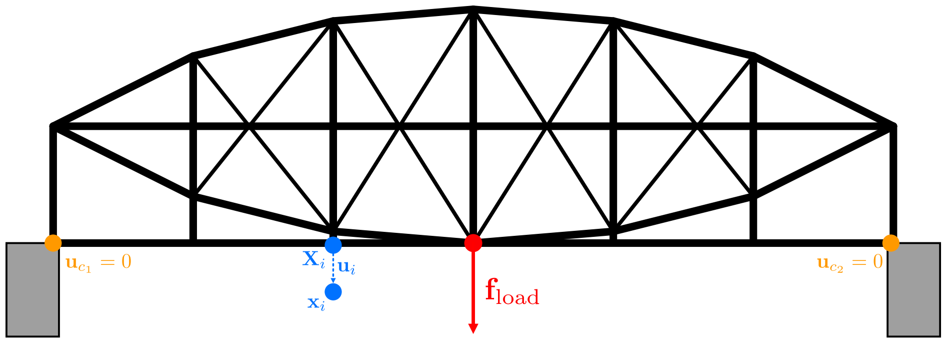

Figure 1: A truss bridge subjected to a force $\mathbf{f}_{\text{load}}$ applied to one of its joints and glued to supports at its bottom left and bottom right corners. Each node (joint) of the truss will displace by an unknown vector $\mathbf{u}_i$ that your simulation will compute.

In this part, you will implement a simple simulation of *truss structures*: structures composed of elastic beams attached at pivot joints. This simulation model is appropriate for stiff beam structures like the one shown in Figure 1, where we want loads to be carried fully by the beams themselves rather than the relatively weak bolts used to join them together. The goal is to solve for the *displacement vector* $\mathbf{u}_i \in \mathbb{R}^2$ for each joint, describing how the joint moves from its initial position $\mathbf{X}_i$ to its deformed position $\mathbf{x}_i = \mathbf{X}_i + \mathbf{u}_i$ due to forces $\mathbf{f}_i \in \mathbb{R}^2$ applied at each joint.

The simplest way to formulate the simulation is in terms of minimizing the *total potential energy* of the system. This total potential energy consists of both the elastic energy stored in the stretched/compressed beams and the potential energy associated with the applied forces:

$$E(\mathbf{U}) = E_{\text{elastic}}(\mathbf{U}) - \sum_{i=1}^{n} \mathbf{f}_i \cdot \mathbf{u}_i, \tag{1}$$

where $\mathbf{U} \in \mathbb{R}^{2n}$ stacks all of the 2D displacements $\mathbf{u}_i$ of each of the n nodes into a single long vector of unknowns. Note that $\mathbf{f}_i \cdot \mathbf{u}_i$ is the work done by the force $\mathbf{f}_i$ while displacing the i^{th} joint by $\mathbf{u}_i$.

For small displacements, the elastic energy stored in the beam for edge $e = (i, j)$ connecting joints i and j can be computed from Hooke's law as:

$$\frac{k_e}{2}(\Delta L_e)^2 = \frac{k_e}{2}(\|\mathbf{x}_i - \mathbf{x}_j\| - \|\mathbf{X}_i - \mathbf{X}_j\|)^2 \approx \frac{k_e}{2}(\hat{\mathbf{a}}_e \cdot (\mathbf{u}_i - \mathbf{u}_j))^2$$

where $\hat{\mathbf{a}}_e := \frac{\mathbf{X}_i - \mathbf{X}_j}{\|\mathbf{X}_i - \mathbf{X}_j\|}$ is the unit axis vector of the edge before the deformation. This expression can be derived formally by linearizing the deformed length calculation using a Taylor expansion. Intuitively, it approximates the change in length ΔL_e by measuring the change in length of its shadow along (projection onto) the original beam axis $\hat{\mathbf{a}}_e$.

We obtain the total energy by summing over edges in the truss:

$$E_{\text{elastic}}(\mathbf{U}) = \sum_{e=(i,j)} \frac{k_e}{2}(\hat{\mathbf{a}}_e \cdot (\mathbf{u}_i - \mathbf{u}_j))^2 = \frac{1}{2}\mathbf{U}^\top \underbrace{B^\top D B}_{K} \mathbf{U}, \tag{2}$$

where $B \in \mathbb{R}^{m \times 2n}$ is the matrix such that $B\mathbf{U}$ computes the projected edge length change for each edge (where m is the number of edges), and diagonal matrix $D \in \mathbb{R}^{m \times m}$ collects the beam stiffnesses k_e on its diagonals. In other words, we define B such that $[B\mathbf{U}]_e = \hat{\mathbf{a}}_e^\top(\mathbf{u}_i - \mathbf{u}_j)$. It follows that each row of B (corresponding to a single edge (i, j) of the truss) has two nonzero blocks: one holding row vector $\hat{\mathbf{a}}_e^\top$ that corresponds to node i and one holding $-\hat{\mathbf{a}}_e^\top$ that corresponds to node j.

The full matrix appearing in the quadratic expression of vector $\mathbf{U}$ in (2) is the *stiffness matrix* $K := B^\top D B$. With it, the total potential energy in (1) can be rewritten as:

$$E(\mathbf{U}) = \frac{1}{2}\mathbf{U}^\top K \mathbf{U} - \mathbf{F} \cdot \mathbf{U},$$

where vector $\mathbf{F} \in \mathbb{R}^{2n}$ stacks the input force vectors $\mathbf{f}_i$ in the same way that $\mathbf{U}$ stacks the displacements $\mathbf{u}_i$. Notice how $\frac{1}{2}\mathbf{U}^\top K \mathbf{U}$ can be thought of as a generalization of the standard 1D spring energy expression given by Hooke's law.

In order to simulate the full structure's displacement, we want to minimize this potential energy with respect to $\mathbf{U}$. But first we must apply some constraints to prevent the structure from flying off to infinity the moment that forces are applied. In the example of Figure 1, we see that the bottom two corners are glued to supports. These supports apply the constraints $\mathbf{u}_c = 0$ for each glued joint with index $c \in \mathcal{C}$. In other words, we wish to solve the constrained problem:

$$\min_{\substack{U \\ \mathbf{u}_c = 0 \,\forall\, c \in \mathcal{C}}} \frac{1}{2}\mathbf{U}^\top K \mathbf{U} - \mathbf{F} \cdot \mathbf{U}.$$

It can be shown that this quadratic optimization problem is equivalent to solving the linear system

$$\tilde{K}\mathbf{U} = \tilde{\mathbf{F}},$$

where $\tilde{K}$ is obtained by replacing the rows and columns corresponding to the joints in $\mathcal{C}$ with rows and columns of the identity matrix (zeroing out all entries in these rows/cols and then putting ones on their diagonal entries), and $\tilde{\mathbf{F}}$ is obtained by overwriting the entries corresponding to joints in $\mathcal{C}$ with zeros.

Whereas K is only *positive semidefinite* (Problem 2), as long as enough joints of the truss are constrained to prevent the whole structure from rigidly translating or rotating, $\tilde{K}$ will be *positive definite*. Therefore Cholesky factorization is the ideal approach for solving this linear system.

Problem 14 (15 pts). Implement `construct_K` in `truss_simulation.py` to build the stiffness matrix for the truss structure described by arguments `X`, `E`, and `k`. Input `X` is a matrix of size $n \times 2$ holding the initial coordinates of each node. Input `E` is a matrix of size $m \times 2$ where each row `E[e]` holds the indices of the two joints connected by edge `e`. Finally, input `k` is a vector of size m holding the stiffness of each beam.

Start by completing `construct_B` to build the $m \times 2n$ matrix B one row at a time. In other words, loop over each edge index e, construct the unit axis vector $\hat{\mathbf{a}}_e$, and then put a positive copy in block `B[e, 2 i:2(i + 1)]` and a negative copy in block `B[e, 2 j:2(j + 1)]`. Next, compute DB by scaling each row of B by its corresponding stiffness (which can be done without explicitly constructing the diagonal matrix). Finally, compute $K = B^\top DB$.

Problem 15 (10 pts). Implement `simulate` in `truss_simulation.py` by building the modified linear system $\tilde{K}\mathbf{U} = \tilde{\mathbf{F}}$ and solving it using the Cholesky factorization approach. This function is passed `X`, `E`, and `k` as described above, as well as inputs `F` and `C`. Input `F` is the stacked force vector of length $2n$ called $\mathbf{F}$ above, and input `C` is a list holding the indices of the constrained joints.

First construct the stiffness matrix K by calling `construct_K` and then modify it by calling `apply_bcs` (implement this function too!) to obtain $\tilde{K}$ as described above. Next, construct the right-hand-side vector $\tilde{\mathbf{F}}$ by zeroing out the entries corresponding to the constrained joints.

Finally apply your `linear_systems.solve_cholesky` routine and return the solution vector $\mathbf{U}$.

Run `python3 truss_simulation.py data/bridge.pkl` to simulate a truss structure like the one in Figure 1. This will generate plots `bridge_init.pdf` and `bridge_deformed.pdf` showing the initial and deformed configurations, respectively; include those in your report.

You can also try running the code on the other examples provided in the `data` folder, or attempt to set up your own simulation!

Problem 16 (5 pts). Count the average number of nonzero entries in each row of the matrix K constructed for the bridge example above and report it in your `report.pdf`. Does storing and operating on all the entries of K seem like an efficient approach for this problem?

Simulation unit tests: You should check that each step of your implementation is working by running the unit tests: `python3 truss_simulation.py --test`. I'd again recommend that you you check this after finishing each function to catch bugs at the source.

4 What to Turn In

Submit a `zip` archive to Canvas containing:

1) a **single PDF** named `report.pdf` with your written solutions to the theory part as well as the plots and program output requested above; and
2) your edited source files `qr.py`, `curvefit.py`, `linear_systems.py`, and `truss_simulation.py`.

Your PDF document can be handwritten, but to obtain credit it must be *easily readable*.

5 Late Work and Collaboration Policy

Late work will be accepted with a deduction of 10pts per day up until graded homework is returned. The deadline is strict: homework submitted after 11:59pm will be considered one day late.

Discussing the problems with classmates is allowed, but you may not collaborate as you write up your solutions, and you must not at any point look at another student's written solutions. You also must not search for solutions online: if you are stuck on a problem, please message me or post a question to the Canvas discussion board.