EE263 Homework 5 Solutions

Fall 2023

6.790. Identifying a system from input/output data. We consider the standard setup:

$$y = Ax + v,$$

where $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$ is the input vector, $y \in \mathbb{R}^m$ is the output vector, and $v \in \mathbb{R}^m$ is the noise or disturbance. We consider here the problem of estimating the matrix A, given some input/output data. Specifically, we are given the following:

$$x^{(1)}, \ldots, x^{(N)} \in \mathbb{R}^n, \qquad y^{(1)}, \ldots, y^{(N)} \in \mathbb{R}^m.$$

These represent N samples or observations of the input and output, respectively, possibly corrupted by noise. In other words, we have

$$y^{(k)} = Ax^{(k)} + v^{(k)}, \quad k = 1, \ldots, N,$$

where $v^{(k)}$ are assumed to be small. The problem is to estimate the (coefficients of the) matrix A, based on the given input/output data. You will use a least-squares criterion to form an estimate $\hat{A}$ of A. Specifically, you will choose as your estimate $\hat{A}$ the matrix that minimizes the quantity

$$J = \sum_{k=1}^{N} \|Ax^{(k)} - y^{(k)}\|^2$$

over A.

a) Explain how to do this. If you need to make an assumption about the input/output data to make your method work, state it clearly. You may want to use the matrices $X \in \mathbb{R}^{n\times N}$ and $Y \in \mathbb{R}^{m\times N}$ given by

$$X = \left[\, x^{(1)} \;\; \cdots \;\; x^{(N)} \,\right], \qquad Y = \left[\, y^{(1)} \;\; \cdots \;\; y^{(N)} \,\right]$$

in your solution.

b) On the course web site you will find some input/output data for an instance of this problem in the file `sysid_data.json`. Executing this Julia file will assign values to m, n, and N, and create two matrices that contain the input and output data, respectively. The $n \times N$ matrix variable `X` contains the input data $x^{(1)}, \ldots, x^{(N)}$ (*i.e.*, the first column of `X` contains $x^{(1)}$, etc.). Similarly, the $m \times N$ matrix `Y` contains the output data $y^{(1)}, \ldots, y^{(N)}$. You must give your final estimate $\hat{A}$, your source code, and also give an explanation of what you did.

Solution.

a) We start by expressing the objective function J as

$$
\begin{aligned}
J &= \sum_{k=1}^{N} \|Ax^{(k)} - y^{(k)}\|^2 \\
&= \sum_{k=1}^{N} \sum_{i=1}^{m} (Ax^{(k)} - y^{(k)})_i^2 \\
&= \sum_{k=1}^{N} \sum_{i=1}^{m} (a_i^\mathsf{T} x^{(k)} - y_i^{(k)})^2 \\
&= \sum_{i=1}^{m} \left(\sum_{k=1}^{N} (a_i^\mathsf{T} x^{(k)} - y_i^{(k)})^2 \right),
\end{aligned}
$$

where a_i^T is the ith row of A. The last expression shows that J is a sum of expressions J_i (shown in parentheses), each of which only depends on a_i. This means that to minimize J, we can minimize each of these expressions separately. That makes sense: we can estimate the rows of A separately. Now let's see how to minimize

$$
J_i = \sum_{k=1}^{N} (a_i^\mathsf{T} x^{(k)} - y_i^{(k)})^2,
$$

which is the contribution to J from the ith row of A. First we write it as

$$
J_i = \left\| \begin{bmatrix} x^{(1)T} \\ \vdots \\ x^{(N)T} \end{bmatrix} a_i - \begin{bmatrix} y_i^{(1)} \\ \vdots \\ y_i^{(N)} \end{bmatrix} \right\|^2 .
$$

Now that we have the problem in the standard least-squares format, we're pretty much done. Using the matrix $X \in \mathbb{R}^{n \times N}$ given by

$$
X = \begin{bmatrix} x^{(1)} & \cdots & x^{(N)} \end{bmatrix},
$$

we can express the estimate as

$$
\hat{a}_i = (XX^\mathsf{T})^{-1} X \begin{bmatrix} y_i^{(1)} \\ \vdots \\ y_i^{(N)} \end{bmatrix}.
$$

Using the matrix $Y \in \mathbb{R}^{m \times N}$ given by

$$
Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(N)} \end{bmatrix},
$$

we can express the estimate of A as

$$
\hat{A}^\mathsf{T} = (XX^\mathsf{T})^{-1} X Y^\mathsf{T}.
$$

Transposing this gives the final answer:

$$
\hat{A} = Y X^\mathsf{T} (XX^\mathsf{T})^{-1}.
$$

b) Once you have the neat formula found above, it's easy to compute the estimate. It's a little inefficient, but perfectly correct, to simply use

```
Ahat = Y*X'*inv(X*X');
```

This yields the estimate

$$\hat{A} = \begin{bmatrix} 2.03 & 5.02 & 5.01 \\ 0.01 & 7 & 1.01 \\ 7.04 & 0 & 6.94 \\ 7 & 3.98 & 4 \\ 9.01 & 1.04 & 7 \\ 4.01 & 3.96 & 9.03 \\ 4.99 & 6.97 & 8.03 \\ 7.94 & 6.09 & 3.02 \\ 0.01 & 8.97 & -0.04 \\ 1.06 & 8.02 & 7.03 \end{bmatrix}.$$

Once you've got $\hat{A}$, it's a good idea to check the residuals, just to make sure it's reasonable, by comparing it to

$$\sum_{k=1}^{N} \|y^{(k)}\|^2.$$

Here we get $(64.5)^2$, around 4.08%. There are several other ways to compute $\hat{A}$ in Julia. You can calculate the rows of $\hat{A}$ one at a time:

```
A = zeros(10,3)
for i=1:10
A[i,:] = X' \ Y[i,:]
end
```

In fact, the backslash operator in Julia solves multiple least-squares problems at once, so you can use

```
A = transpose(X' \ Y')
```

In any case, it's not exactly a long program ...

6.810. Estimating a signal with interference. This problem concerns three proposed methods for estimating a signal, based on a measurement that is corrupted by a small noise and also by an interference, that need not be small. We have

$$y = Ax + Bv + w,$$

where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{m\times p}$ are known. Here $y \in \mathbb{R}^m$ is the measurement (which is known), $x \in \mathbb{R}^n$ is the signal that we want to estimate, $v \in \mathbb{R}^p$ is the interference, and w is a noise. The noise is unknown, and can be assumed to be small. The interference is unknown, but cannot be assumed to be small. You can assume that the matrices A and B are skinny and full rank (*i.e.*, $m > n$, $m > p$), and that the ranges of A and B intersect only at 0. (If this last condition does not hold, then there is no hope of finding x, even when $w = 0$, since a nonzero interference can masquerade as a signal.) Each of the EE263 TAs proposes a method for estimating x. These methods, along with some informal justification from their proposers, are given below. Nikola proposes the **ignore and estimate method.** He describes it as follows:

> We don't know the interference, so we might as well treat it as noise, and just ignore it during the estimation process. We can use the usual least-squares method, for the model $y = Ax + z$ (with z a noise) to estimate x. (Here we have $z = Bv + w$, but that doesn't matter.)

Almir proposes the **estimate and ignore method.** He describes it as follows:

> We should simultaneously estimate both the signal x *and* the interference v, based on y, using a standard least-squares method to estimate $[x^\mathsf{T}\ v^\mathsf{T}]^\mathsf{T}$ given y. Once we've estimated x and v, we simply ignore our estimate of v, and use our estimate of x.

Miki proposes the **estimate and cancel method.** He describes it as follows:

> Almir's method makes sense to me, but I can improve it. We should simultaneously estimate both the signal x and the interference v, based on y, using a standard least-squares method, exactly as in Almir's method. In Almir's method, we then throw away $\hat{v}$, our estimate of the interference, but I think we should use it. We can form the "pseudo-measurement" $\tilde{y} = y - B\hat{v}$, which is our measurement, with the effect of the estimated interference subtracted off. Then, we use standard least-squares to estimate x from $\tilde{y}$, from the simple model $\tilde{y} = Ax + z$. (This is exactly as in Nikola's method, but here we have subtracted off or cancelled the effect of the estimated interference.)

These descriptions are a little vague; part of the problem is to translate their descriptions into more precise algorithms.

a) Give an explicit formula for each of the three estimates. (That is, for each method give a formula for the estimate $\hat{x}$ in terms of A, B, y, and the dimensions n, m, p.)

b) Are the methods really different? Identify any pairs of the methods that coincide (*i.e.*, always give exactly the same results). If they are all three the same, or all three different, say so. Justify your answer. To show two methods are the same, show that the formulas given in part (a) are equal (even if they don't appear to be at first). To show two methods are different, give a specific numerical example in which the estimates differ.

c) Which method or methods do you think work best? Give a very brief explanation. (If your answer to part (b) is "The methods are all the same" then you can simply repeat here, "The methods are all the same".)

Solution. Let's analyze the three methods. For Nikola's method, we have

$$\hat{x}_{\rm ie} = (A^\mathsf{T} A)^{-1} A^\mathsf{T} y.$$

(The subscript 'ie' stands for 'ignore and estimate'.) Almir's method is more complicated. We start with

$$y = \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + w.$$

We need to say something about the rank of the matrix $\begin{bmatrix} A & B \end{bmatrix}$. Each is full rank, *i.e.*, has independent columns. We are also told that the ranges of the matrices A and B only intersect at 0. This means that the range of $\begin{bmatrix} A & B \end{bmatrix}$ has dimension $n+p$, which must be no more than m. Thus, the matrix $\begin{bmatrix} A & B \end{bmatrix}$ is also skinny (or square) and full rank. Forming a least-squares estimate of x and v we get

$$\begin{aligned} \begin{bmatrix} \hat{x}_{\rm ei} \\ \hat{v}_{\rm ei} \end{bmatrix} &= \left(\begin{bmatrix} A & B \end{bmatrix}^\mathsf{T} \begin{bmatrix} A & B \end{bmatrix} \right)^{-1} \begin{bmatrix} A & B \end{bmatrix}^\mathsf{T} y \\ &= \begin{bmatrix} A^\mathsf{T} A & A^\mathsf{T} B \\ B^\mathsf{T} A & B^\mathsf{T} B \end{bmatrix}^{-1} \begin{bmatrix} A & B \end{bmatrix}^\mathsf{T} y. \end{aligned}$$

(The subscript 'ei' stands for 'estimate and ignore'.) Finally, let's work out Miki's method. The pseudo-measurement is given by $\tilde{y} = y - B\hat{v}$, and then we have

$$\hat{x}_{\rm ec} = (A^\mathsf{T} A)^{-1} A^\mathsf{T} (y - B\hat{v}) = (A^\mathsf{T} A)^{-1} (A^\mathsf{T} y - A^\mathsf{T} B\hat{v}),$$

where $\hat{v} = \hat{v}_{\rm ei}$. (The subscript 'ec' stands for 'estimate and cancel'.) So far it's not clear that any of these are the same. But it turns out that Almir and Miki's method are identical. To see this, we note that

$$\begin{bmatrix} A^\mathsf{T} A & A^\mathsf{T} B \\ B^\mathsf{T} A & B^\mathsf{T} B \end{bmatrix} \begin{bmatrix} \hat{x}_{\rm ei} \\ \hat{v}_{\rm ei} \end{bmatrix} = \begin{bmatrix} A & B \end{bmatrix}^\mathsf{T} y.$$

Expanding this we get

$$A^\mathsf{T} A\hat{x}_{\rm ei} + A^\mathsf{T} B\hat{v}_{\rm ei} = A^\mathsf{T} y, \qquad B^\mathsf{T} A\hat{x}_{\rm ei} + B^\mathsf{T} B\hat{v}_{\rm ei} = B^\mathsf{T} y.$$

This means that

$$A^\mathsf{T} y - A^\mathsf{T} B\hat{v}_{\rm ei} = A^\mathsf{T} A\hat{x}_{\rm ei},$$

and plugging this into our formula above for $\hat{x}_{\rm ec}$, we get

$$\hat{x}_{\rm ec} = (A^\mathsf{T} A)^{-1} (A^\mathsf{T} y - A^\mathsf{T} B\hat{v}_{\rm ei}) = (A^\mathsf{T} A)^{-1} A^\mathsf{T} A\hat{x}_{\rm ei} = \hat{x}_{\rm ei}.$$

Thus, we have shown that Almir's and Miki's methods are the same. In other words, the extra step of cancelling off the effect of v does *nothing*. Finally, we need to show that Nikola's is different from the other two. Here is a simple example that shows it's different:

$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

We then have $\hat{x}_{\rm ie} = [1\ 0]y$, and

$$\begin{bmatrix} \hat{x}_{\rm ei} \\ \hat{v}_{\rm ei} \end{bmatrix} = \begin{bmatrix} A^\mathsf{T} A & A^\mathsf{T} B \\ B^\mathsf{T} A & B^\mathsf{T} B \end{bmatrix}^{-1} \begin{bmatrix} A & B \end{bmatrix}^\mathsf{T} y = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} y,$$

so $\hat{x}_{\rm ei} = [-1\ 1]y$. Clearly the estimates differ; for $y = [1\ 1]$ we have $\hat{x}_{\rm ie} = 1$, but $\hat{x}_{\rm ie} = 0$.

7.1040. Fitting a Gaussian function to data. A Gaussian function has the form

$$f(t) = ae^{-(t-\mu)^2/\sigma^2}.$$

Here $t \in \mathbb{R}$ is the independent variable, and $a \in \mathbb{R}$, $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}$ are parameters that affect its shape. The parameter a is called the *amplitude* of the Gaussian, μ is called its *center*, and σ is called the *spread* or *width*. We can always take $\sigma > 0$. For convenience we define $p \in \mathbb{R}^3$ as the vector of the parameters, *i.e.*, $p = [a \ \mu \ \sigma]^\mathsf{T}$. We are given a set of data,

$$t_1, \ldots, t_N, \qquad y_1, \ldots, y_N,$$

and our goal is to fit a Gaussian function to the data. We will measure the quality of the fit by the root-mean-square (RMS) fitting error, given by

$$E = \left(\frac{1}{N} \sum_{i=1}^{N} (f(t_i) - y_i)^2 \right)^{1/2}.$$

Note that E is a function of the parameters a, μ, σ, *i.e.*, p. Your job is to choose these parameters to minimize E. You'll use the Gauss-Newton method.

a) Work out the details of the Gauss-Newton method for this fitting problem. Explicitly describe the Gauss-Newton steps, including the matrices and vectors that come up. You can use the notation $\Delta p^{(k)} = [\Delta a^{(k)} \ \Delta \mu^{(k)} \ \Delta \sigma^{(k)}]^\mathsf{T}$ to denote the update to the parameters, *i.e.*,

$$p^{(k+1)} = p^{(k)} + \Delta p^{(k)}.$$

(Here k denotes the kth iteration.)

b) Get the data t, y (and N) from the file `gauss_fit_data.json`, available on the class website. Implement the Gauss-Newton (as outlined in part (a) above). You'll need an initial guess for the parameters. You can visually estimate them (giving a short justification), or estimate them by any other method (but you must explain your method). Plot the RMS error E as a function of the iteration number. (You should plot enough iterations to convince yourself that the algorithm has nearly converged.) Plot the final Gaussian function obtained along with the data on the same plot. Repeat for another reasonable, but different initial guess for the parameters. Repeat for another set of parameters that is *not* reasonable, *i.e.*, not a good guess for the parameters. (It's possible, of course, that the Gauss-Newton algorithm doesn't converge, or fails at some step; if this occurs, say so.) Briefly comment on the results you obtain in the three cases.

Solution.

a) Minimizing E is the same as minimizing NE^2, which is a nonlinear least-squares problem. The first thing to do is to find the first-order approximation of the Gaussian function, with respect to the parameters a, μ, and σ. This approximation is

$$f(t) + \frac{\partial}{\partial a} f(t) \Delta a + \frac{\partial}{\partial \mu} f(t) \Delta \mu + \frac{\partial}{\partial \sigma} f(t) \Delta \sigma,$$

where all the partial derivatives are evaluated at the current parameter values. In matrix form, this first-order approximation is

$$f(t) + (\nabla_p f(t))^\mathsf{T} \Delta p,$$

where ∇_p denotes the gradient with respect to p. These partial derivatives are:

$$\frac{\partial}{\partial a} f(t) = e^{-(t-\mu)^2/\sigma^2}$$
$$\frac{\partial}{\partial \mu} f(t) = \frac{2a(t-\mu)}{\sigma^2} e^{-(t-\mu)^2/\sigma^2}$$
$$\frac{\partial}{\partial \sigma} f(t) = \frac{2a(t-\mu)^2}{\sigma^3} e^{-(t-\mu)^2/\sigma^2}$$

The Gauss-Newton method proceeds as follows. We find Δp that minimizes

$$\sum_{i=1}^{N} \left(f(t_i) + \nabla_p f(t_i)^\mathsf{T} \Delta p - y_i \right)^2,$$

and then set the new value of p to be $p := p + \Delta p$. Finding Δp is a (linear) least-squares problem. We can put this least-squares problem in a more conventional form by defining

$$A = \begin{bmatrix} \nabla_p f(t_1)^\mathsf{T} \\ \vdots \\ \nabla_p f(t_N)^\mathsf{T} \end{bmatrix}, \qquad b = \begin{bmatrix} y_1 - f(t_1) \\ \vdots \\ y_N - f(t_N) \end{bmatrix}.$$

Then, Δp is found by minimizing $\|A\Delta p - b\|$. Thus, we have

$$\Delta p = (A^\mathsf{T} A)^{-1} A^\mathsf{T} b.$$

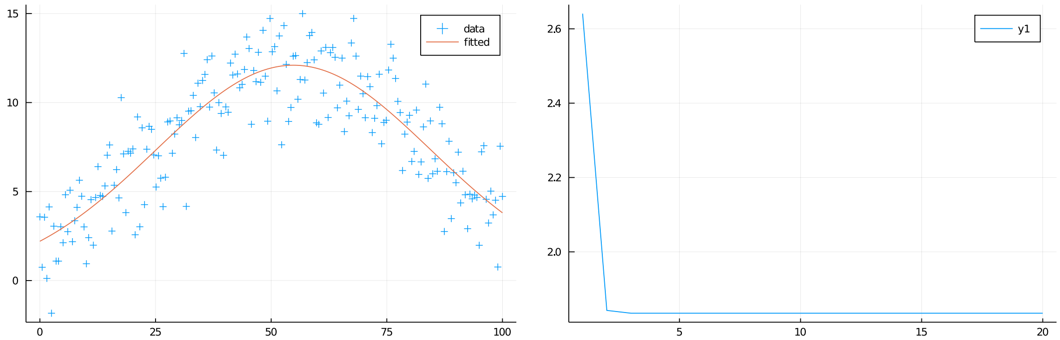
To summarize, the algorithm repeats the following steps:

- Evaluate the vector b (which is the vector of fitting residuals.) Evaluate the partial derivatives to form the matrix A.
- Solve the least-squares problem to get Δp.
- Update the parameter vector: $p := p + \Delta p$.

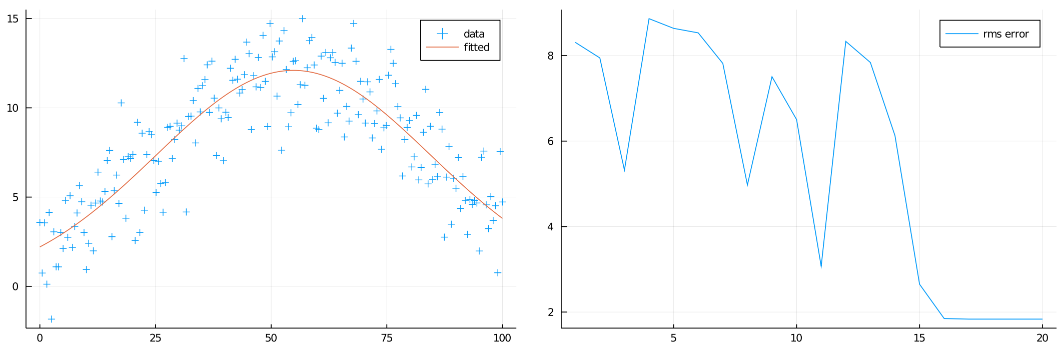
This can be repeated until the update Δp is small, or the improvement in E is small.

b) We used the starting parameter values $p = [11, 50, 35]^\mathsf{T}$, estimated visually. The amplitude $a = 11$ was estimated as a guess for the (noise-free) peak of the graph, $\mu = 50$ was estimated as its center, and $\sigma = 35$ was estimated from its spread.

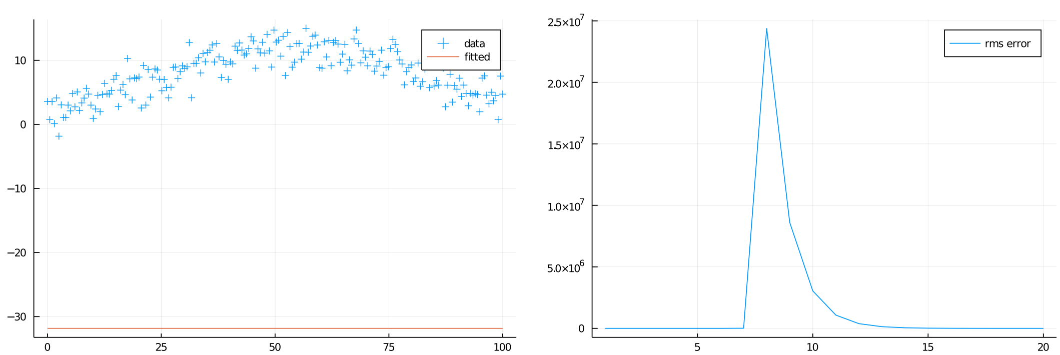
The results are shown below. The final fit clearly is good (at least, visually), at $a \approx 12.10, \mu \approx 54.81, \sigma \approx 42.02$. The final RMS fit level is around $E \approx 1.83$, and convergence happens very quickly, in just a handful of iterations.



Now we try with another starting point, $p = (10, 20, 10)$. The final fit is the same (well, σ landed on -42.02, but that doesn't matter). It does tend to bounce around a bit more before converging, which is indicative of its nonlinearity. That it landed in the same place bolsters our confidence that the fit found in our first run (the same as this one) is probably the best fit possible.



For other poor initial guesses, however, the algorithm fails to converge. For example, with initial parameter estimate $p = (5, 20, 10)$, there's a miserable spike before coming back to $E \approx 105$, a clearly not optimal fit.



The Julia code for the Gauss-Newton method is given below.

Note: Julia supports Unicode characters, so if you type something like `\sigma` *then Tab, Jupyter and Julia's REPL will convert it to the Greek letter, in place of the Latin-spelled* `sigma`*. But LATEX doesn't support Unicode characters, so we Latinized the Greek letters to print the code here.*

```
using LinearAlgebra
using Plots
include("readclassjson.jl")
data = readclassjson("gauss_fit_data.json")

N = data["N"]
t = data["t"]
y = data["y"]

function fit_gaussian(p_init)
    p = p_init
    E = Float64[]
    for i = 1:20
        a, mu, sigma = p
        w = exp.(-(t .- mu).^2 / sigma^2)
        A =[w  2*a*(t.-mu)/sigma^2 .* w  2*a*(t.-mu).^2/sigma^3 .* w]
        f = a .* w
        b = y - f
        Deltap = A \ b
        p += Deltap
        push!(E, sqrt(sum((f - y).^2) / N))
    end
    p..., E
end

fit_gaussian(a_init, mu_init, sigma_init) = fit_gaussian([a_init, mu_init, sigma_init])

a, mu, sigma, E = fit_gaussian(20, 50, 15)
f = a * exp.(-(t .- mu).^2 / sigma^2)
scatter(t, y, label="data", marker=:+)
plot!(t, f, label="fitted")

plot(E, label="rms error")
```

8.1110. Simultaneous left inverse of two matrices. Consider a system where

$$y = Gx, \quad \tilde{y} = \tilde{G}x$$

where $G \in \mathbb{R}^{m\times n}$, $\tilde{G} \in \mathbb{R}^{m\times n}$. Here x is some variable we wish to estimate or find, y gives the measurements with some set of (linear) sensors, and $\tilde{y}$ gives the measurements with some *alternate* set of (linear) sensors. We want to find a *reconstruction matrix* $H \in \mathbb{R}^{n\times m}$ such that $HG = H\tilde{G} = I$. Such a reconstruction matrix has the nice property that it recovers x perfectly from *either* set of measurements (y or $\tilde{y}$), *i.e.*, $x = Hy = H\tilde{y}$. Consider the specific case

$$G = \begin{bmatrix} 2 & 3 \\ 1 & 0 \\ 0 & 4 \\ 1 & 1 \\ -1 & 2 \end{bmatrix}, \quad \tilde{G} = \begin{bmatrix} -3 & -1 \\ -1 & 0 \\ 2 & -3 \\ -1 & -3 \\ 1 & 2 \end{bmatrix}.$$

Either find an explicit reconstruction matrix H, or explain why there is no such H.

Solution. The requirements $HG = I$ and $H\tilde{G} = I$ are a set of linear equations in the variables H_{ij}. Since $H \in \mathbb{R}^{n\times m}$ there are mn unknowns; each equation of the form $HG = I$ gives n^2 equations, so all together we have $2n^2$ equations. Roughly speaking, it's reasonable to expect a solution to exist when there are more variables than equations, *i.e.*, $mn \geq 2n^2$, which implies that $m \geq 2n$. This condition makes sense: to invert two different sensor measurements we need a redundancy factor of two. Now let's look at the specific case given. Suppose that

$$H = \begin{bmatrix} h_1^\mathsf{T} \\ h_2^2 \end{bmatrix}$$

where $h_1, h_2 \in \mathbb{R}^5$. $HG = I$ implies that $h_1^\mathsf{T} G = e_1^\mathsf{T}$ and $h_2^\mathsf{T} G = e_2^\mathsf{T}$ where e_1 and e_2 are the unit vectors in $\mathbb{R}^2$. Similarly we should have $h_1^\mathsf{T}\tilde{G} = e_1^\mathsf{T}$ and $h_2^\mathsf{T}\tilde{G} = e_2^\mathsf{T}$. In block matrix form

$$\begin{bmatrix} G^\mathsf{T} & 0 \\ \tilde{G}^\mathsf{T} & 0 \\ 0 & G^\mathsf{T} \\ 0 & \tilde{G}^\mathsf{T} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} e_1 \\ e_1 \\ e_2 \\ e_2 \end{bmatrix}.$$

Now by defining

$$A = \begin{bmatrix} G^\mathsf{T} & 0 \\ \tilde{G}^\mathsf{T} & 0 \\ 0 & G^\mathsf{T} \\ 0 & \tilde{G} \end{bmatrix}, \quad x = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}, \quad b = \begin{bmatrix} e_1 \\ e_2 \\ e_1 \\ e_2 \end{bmatrix}$$

we get the standard form $Ax = b$. If $b \in \text{range}(A)$ a solution exists and H can be found. In this case, A happens to be full rank so $(AA^\mathsf{T})^{-1}$ exists and we can set $x = A^\mathsf{T}(AA^\mathsf{T})^{-1}b$.

Here is the Julia code to solve this problem:

```
using LinearAlgebra;

G = [2 3; 1 0; 0 4; 1 1; -1 2]
G_tilde = [-3 -1; -1 0; 2 -3; -1 -3; 1 2]
m, n = size(G)

# Try to find simultaneous left inverse for G and G_tilde
A = [G' zeros(n, m);
     G_tilde' zeros(n, m);
```

```
     zeros(n, m) G';
     zeros(n, m) G_tilde']
b = [I(n)[:, 1]; I(n)[:, 1]; I(n)[:, 2]; I(n)[:, 2]]

if rank([A b]) != rank(A)
    print("There is no simultaneous left inverse.\n")
else
    x = A \ b
    H = [x[begin:m]';
         x[(m + 1):end]']
    print("The simultaneous left inverse is given by:\n")
    display(H)

    # Confirm that H is a simulatenous left inverse
    print("HG = I: ", all(abs.(I(n) - H * G) .< 1e-14), "\n")
    print("HG_tilde = I: ", all(abs.(I(n) - H * G_tilde) .< 1e-14), "\n")
end
```

Using this code, you can see that G and $\tilde{G}$ *do* have a simulatenous left inverse H; the specific solution H that we found running the code above is given by

$$H = \begin{bmatrix} -0.278166 & 2.09441 & 0.860917 & -1.22844 & -0.690365 \\ 0.161616 & 0.227273 & 0.0808081 & -0.30303 & 0.247475 \end{bmatrix}. \tag{1}$$

Of course, there are other solutions as well. Indeed, since there are 10 variables and 8 independent equations, there is a 2 dimensional set of H's that satisfy the required condition.