Lab 4: Numerical linear algebra

MCHA4400

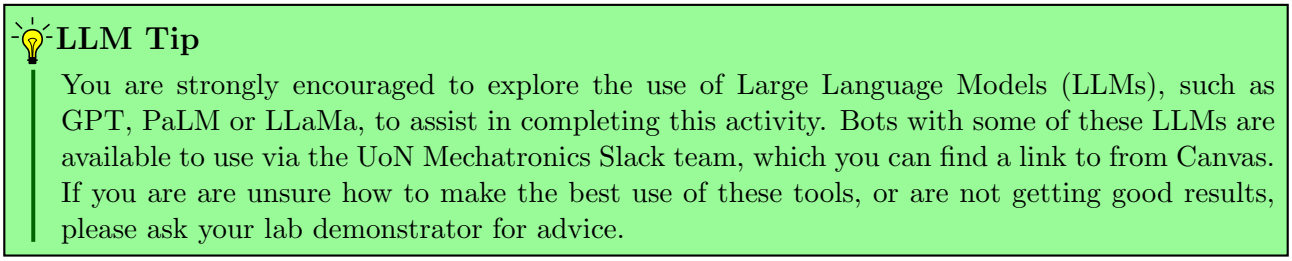
Semester 2 2023

Introduction

In this lab, you will perform matrix and array manipulation and some linear algebra operations using the C++ template library, Eigen. In addition, you will write some unit tests using doctest to validate the solutions.

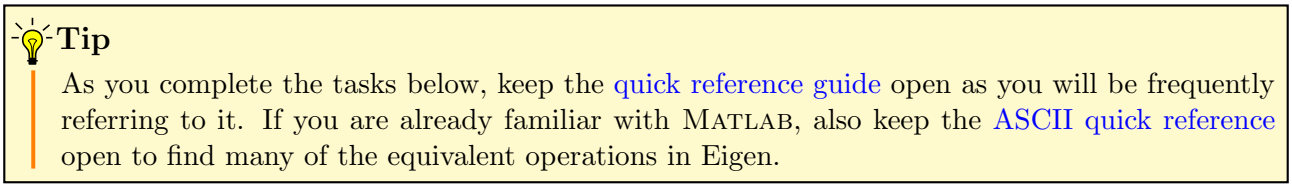
The lab should be completed within 4 hours. The assessment will be done in the lab at the end of your enrolled lab session(s). Once you complete the tasks, call the lab demonstrator to start your assessment.

The lab is worth 5% of your course grade and is graded from 0–5 marks.



1 Preliminaries (1 mark)

In this task you will become acquainted with basic Eigen matrix and array operations. Complete the following tasks in the `src/main.cpp` file:



a) Using the `EIGEN_WORLD_VERSION`, `EIGEN_MAJOR_VERSION`, and `EIGEN_MINOR_VERSION` constants, print out the version of Eigen installed on your computer.

Terminal

```
nerd@basement:~/MCHA4400/lab4$ cmake -G Ninja -B build -DCMAKE_BUILD_TYPE=Debug && cd build
nerd@basement:~/MCHA4400/lab4/build$ ninja && ./lab4
[Ninja-ing intensifies]
[Unit test results]
Eigen version: 3.4.0
... Output from other tasks
nerd@basement:~/MCHA4400/lab4/build$
```

b) Create a vector $\mathbf{x} \in \mathbb{R}^3$, of type `Eigen::VectorXd`, such that $\mathbf{x} = \begin{bmatrix} 1 & 3.2 & 0.01 \end{bmatrix}^\mathsf{T}$. Use the `<<` assignment operator to populate $\mathbf{x}$. Generate the following output:

Terminal

```
nerd@basement:~/MCHA4400/lab4/build$ ninja && ./lab4
Output from other tasks
Create a column vector:
x =
   1
 3.2
0.01
Output from other tasks
nerd@basement:~/MCHA4400/lab4/build$
```

c) Create a matrix $\mathbf{A} \in \mathbb{R}^{4\times3}$ with elements $A_{i,j} = ij$ for $i = 1, 2, 3, 4$ and $j = 1, 2, 3$. Print the `size`, `rows`, `cols` of $\mathbf{A}$ as well as $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ to the console.

Terminal

```
nerd@basement:~/MCHA4400/lab4/build$ ninja && ./lab4
Output from other tasks
Create a matrix:
A.size() = ?
A.rows() = ?
A.cols() = ?
A =
 ?  ?  ?
 ?  ?  ?
 ?  ?  ?
 ?  ?  ?

A.transpose() =
 ?  ?  ?  ?
 ?  ?  ?  ?
 ?  ?  ?  ?
Output from other tasks
nerd@basement:~/MCHA4400/lab4/build$
```

d) Use the $*$ operator to compute the matrix-vector product $\mathbf{Ax}$ to generate the following output:

Terminal

```
nerd@basement:~/MCHA4400/lab4/build$ ninja && ./lab4
Output from other tasks
Matrix multiplication:
A*x =
 7.43
14.86
22.29
29.72
Output from other tasks
nerd@basement:~/MCHA4400/lab4/build$
```

e) Using the `<<` operator, create a matrix $\mathbf{B} \in \mathbb{R}^{4\times6}$ such that $\mathbf{B} = \begin{bmatrix}\mathbf{A} & \mathbf{A}\end{bmatrix}$. Create a matrix $\mathbf{C} \in \mathbb{R}^{8\times3}$ such that $\mathbf{C} = \begin{bmatrix}\mathbf{A} \\ \mathbf{A}\end{bmatrix}$. Output the both $\mathbf{B}$ and $\mathbf{C}$ to the console.

f) Extract a submatrix $\mathbf{D} \in \mathbb{R}^{1\times3}$ from $\mathbf{B}$ such that

$$D_{i,j} = B_{1+i,\ 2+j}, \qquad \forall i, j, \tag{1}$$

and output $\mathbf{D}$ to the console using each of the following methods:

i) a block operation,

ii) a slicing operation.

g) Use `rowwise` to compute the matrix $\mathbf{E}$ such that,

$$E_{i,j} = B_{i,j} + v_j, \qquad \forall i, j \tag{2}$$

where $\mathbf{v} = \begin{bmatrix} 1 & 3 & 5 & 7 & 4 & 6 \end{bmatrix}^\mathsf{T}$. Output $\mathbf{E}$ to the console.

h) Generate the matrix $\mathbf{F} \in \mathbb{R}^{4\times 6}$ from $\mathbf{B}$ such that

$$F_{i,j} = B_{r_i,c_j}, \qquad \forall i, j, \tag{3}$$

where $\mathbf{r} = \begin{bmatrix} 1 & 3 & 2 & 4 \end{bmatrix}$ and $\mathbf{c} = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \end{bmatrix}$ are arrays of indices. Output $\mathbf{F}$ to the console.

i) An existing array has elements `array` $= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}$ in memory. Use an `Eigen::Map` to create a view of this memory as a 3×3 matrix, $\mathbf{G}$, assuming row-major storage. Then, write -3 to the third element of `array` and write -7 to $G_{3,1}$ and print $\mathbf{G}$ to the console to verify that $\mathbf{G}$ and `array` use the same memory.

> **Info**
> `Eigen::Map` can also be used to create a view of OpenCV matrices via `cv::Mat::ptr` (for dynamically-sized matrices) or `cv::Matx<>::val` (for fixed-size matrices, e.g., `cv::Matx33d` or `cv::Vec3d`). This is convenient, since it enables non-Eigen objects access to the full power of the Eigen library and its interface without unnecessarily duplicating data.

2 Pythagorean QR (2 marks)

> **Definition**
> Given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, the QR decomposition produces matrices $\mathbf{Q} \in \mathbb{R}^{m\times m}$ and $\mathbf{R} \in \mathbb{R}^{m\times n}$ such that $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix.

The QR decomposition can be used to propagate a square-root covariance matrix through affine transformations without needing to explicitly compute a matrix square root. As an example, consider the sum of two covariance matrices,

$$\mathbf{P} = \mathbf{P}_1 + \mathbf{P}_2,$$

where $\mathbf{P}, \mathbf{P}_1, \mathbf{P}_2 \in \mathbb{R}^{n\times n}$ are symmetric positive-definite matrices. To implement this operation in square-root form, we seek the upper-triangular matrix $\mathbf{S}$ such that $\mathbf{S}^\mathsf{T}\mathbf{S} = \mathbf{P}$. Assuming that similar factorisations are available for $\mathbf{P}_1$ and $\mathbf{P}_2$, we obtain the following matrix quadratic equation:

$$\begin{aligned} \underbrace{\mathbf{S}^\mathsf{T}\mathbf{S}}_{\mathbf{P}} &= \underbrace{\mathbf{S}_1^\mathsf{T}\mathbf{S}_1}_{\mathbf{P}_1} + \underbrace{\mathbf{S}_2^\mathsf{T}\mathbf{S}_2}_{\mathbf{P}_2} \\ &= \begin{bmatrix} \mathbf{S}_1^\mathsf{T} & \mathbf{S}_2^\mathsf{T} \end{bmatrix} \begin{bmatrix} \mathbf{S}_1 \\ \mathbf{S}_2 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{S}_1 \\ \mathbf{S}_2 \end{bmatrix}^\mathsf{T} \begin{bmatrix} \mathbf{S}_1 \\ \mathbf{S}_2 \end{bmatrix}, \end{aligned} \tag{4}$$

where $\mathbf{S}_1$ and $\mathbf{S}_2$ are any matrices with n columns[1] such that $\mathbf{P}_1 = \mathbf{S}_1^\mathsf{T}\mathbf{S}_1$ and $\mathbf{P}_2 = \mathbf{S}_2^\mathsf{T}\mathbf{S}_2$.

To find an upper triangular solution for $\mathbf{S}$, let $\mathbf{A} = \begin{bmatrix}\mathbf{S}_1 \\ \mathbf{S}_2\end{bmatrix} \in \mathbb{R}^{m\times n}$, where m is the sum of the number of rows in $\mathbf{S}_1$ and $\mathbf{S}_2$. Typically, $\mathbf{A}$ has more rows than columns ($m > n$), which we call a *tall* matrix, and the QR decomposition $\mathbf{A} = \mathbf{QR}$ also has a tall $\mathbf{R}$. Since $\mathbf{R}$ is upper triangular and tall, it must contain $m - n$ rows of zeros at the bottom. Therefore, we can partition the QR decomposition as follows:

$$\underbrace{\begin{bmatrix}\mathbf{S}_1 \\ \mathbf{S}_2\end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix}\mathbf{Y} & \mathbf{Z}\end{bmatrix}}_{\mathbf{Q}} \underbrace{\begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}\end{bmatrix}}_{\mathbf{R}},$$

where $\mathbf{Y} \in \mathbb{R}^{m\times n}$, $\mathbf{Z} \in \mathbb{R}^{m\times(m-n)}$ and $\mathbf{R}_1 \in \mathbb{R}^{n\times n}$. Evaluating $\mathbf{A}^\mathsf{T}\mathbf{A}$ reveals

$$\mathbf{A}^\mathsf{T}\mathbf{A} = \mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{QR} = \mathbf{R}^\mathsf{T}\mathbf{R},$$

where $\mathbf{Q}^\mathsf{T}\mathbf{Q} = \mathbf{I}$ since $\mathbf{Q}$ is orthogonal, and substituting this result into (4) yields

$$\mathbf{S}^\mathsf{T}\mathbf{S} = \begin{bmatrix}\mathbf{S}_1 \\ \mathbf{S}_2\end{bmatrix}^\mathsf{T}\begin{bmatrix}\mathbf{S}_1 \\ \mathbf{S}_2\end{bmatrix} = \begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}\end{bmatrix}^\mathsf{T}\begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}\end{bmatrix} = \mathbf{R}_1^\mathsf{T}\mathbf{R}_1.$$

Since $\mathbf{R}_1$ is a square upper triangular matrix, a solution to (4) is given by setting $\mathbf{S} = \mathbf{R}_1$.

The operation of finding a matrix square root of a sum of squared matrices[2] can be considered a matrix generalisation of Pythagoras' theorem. We colloquially refer to the solution of this problem using QR composition the *Pythagorean QR* algorithm. This operation appears frequently when implementing numerically robust state estimators that propagate covariance via a matrix square root.

In this task, you will create a function that implements Pythagorean QR to solve (4) by finding a QR decomposition using Householder reflections (see both the `householderQr` member function and the `HouseholderQR` class).

> **Tip**
>
> Since the Pythagorean QR algorithm only needs the $\mathbf{R}$ matrix from the QR decomposition, we do not need to evaluate the $\mathbf{Q}$ matrix.

Complete the following tasks:

a) Read through the very short doctest tutorial and then review the BDD-style unit tests you were given in `lab3/test/src/camera.cpp`.

> **Info**
>
> In the BDD-style `SCENARIO`s, note that
>
> - `GIVEN` blocks enclose data,
> - `WHEN` blocks enclose the execution of operations,
> - `THEN` blocks enclose the assertion of expected results.

[1] The matrices $\mathbf{S}_1$ and $\mathbf{S}_2$ need not be upper triangular nor even square in this example, but they do need the same number of columns.

[2] not to be confused with square matrices

> In addition to the assertions used to test the expected results, further `REQUIRE` assertions are used throughout the test scenario to terminate tests early to avoid *crashing* at runtime. This can occur, for example, if an expected file isn't present or if a non-existent element of an `std::vector` is accessed.

b) Create an empty function called `pythagoreanQR` in `src/gaussian.cpp` and place its forward declaration in `src/gaussian.h` with the following prototype:

```
void pythagoreanQR(const Eigen::MatrixXd & S1, const Eigen::MatrixXd & S2, Eigen::MatrixXd & S);
```

c) Review the `GIVEN` blocks within the `SCENARIO` block in `test/src/pythagoreanQR.cpp` that correspond to the following test data:

i) $\mathbf{S}_1 = \mathbf{I}_{3\times3}$ and $\mathbf{S}_2 = \mathbf{0}_{3\times3}$

ii) $\mathbf{S}_1 = \mathbf{0}_{3\times3}$ and $\mathbf{S}_2 = \mathbf{I}_{3\times3}$

iii) $\mathbf{S}_1 = 3\,\mathbf{I}_{3\times3}$ and $\mathbf{S}_2 = 4\,\mathbf{I}_{3\times3}$

iv) $\mathbf{S}_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 16 \end{bmatrix}$ and $\mathbf{S}_2 = \begin{bmatrix} 0 & 10 & -1 & -3 \end{bmatrix}$

d) For each of the cases above, create BDD-style unit tests using the `doctest` API to assert that

i) **S** is upper triangular,

> **Tip**
>
> Testing that a matrix `S` is upper triangular can be achieved using the `isUpperTriangular()` member function and the `CHECK` macro as follows:
>
> ```
> CAPTURE_EIGEN(S);        // Capture S to be printed on test failure
> CHECK(S.isUpperTriangular());
> ```
>
> In the event that this test fails, the knowledge that `S` isn't upper triangular alone doesn't help identify which elements are non-zero in the lower triangle. To provide extra diagnostic information, the `CAPTURE_EIGEN` macro ensures that the entire `S` matrix is printed in full only if the test fails.

ii) **S** has the correct dimensions,

iii) **S** satisfies (4).

> **Tip**
>
> Testing that the elements of a matrix `M` match expected values in `M_exp` may be performed using the `isApprox()` member function and the `CHECK` macro in the following way,
>
> ```
> CAPTURE_EIGEN(M);        // Capture M to be printed on test failure
> CAPTURE_EIGEN(M_exp);    // Capture M_exp to be printed on test failure
> CHECK(M.isApprox(M_exp));
> ```

> In the event that this test fails, the knowledge that `M` is not equal to `M_exp` alone doesn't help diagnose which elements in those matrices are different. To provide extra diagnostic information, the `CAPTURE_EIGEN` macro ensures that both the `M` and `M_exp` matrices are printed in full only if the test fails.

e) Run `ninja` and verify that appropriate the tests *fail*, since `pythagoreanQR` has not yet been implemented. If your tests *crash*, include appropriate `REQUIRE` assertions in the tests as needed.

f) Implement `pythagoreanQR`.

g) Run `ninja` and verify that all the tests pass.

3 Triangular system solve (2 marks)

Recall from MCHA4100 that the joint Gaussian distribution

$$p(\mathbf{x}, \mathbf{y}) = p\left(\begin{bmatrix}\mathbf{y}\\ \mathbf{x}\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\mathbf{y}\\ \mathbf{x}\end{bmatrix}; \begin{bmatrix}\boldsymbol{\mu}_\mathbf{y}\\ \boldsymbol{\mu}_\mathbf{x}\end{bmatrix}, \begin{bmatrix}\mathbf{P}_{\mathbf{yy}} & \mathbf{P}_{\mathbf{yx}}\\ \mathbf{P}_{\mathbf{xy}} & \mathbf{P}_{\mathbf{xx}}\end{bmatrix}\right). \tag{5}$$

has the following conditional distribution:

$$p(\mathbf{x} \mid \mathbf{y}) = \frac{p(\mathbf{x}, \mathbf{y})}{p(\mathbf{y})} = \frac{p(\mathbf{y}, \mathbf{x})}{p(\mathbf{y})} = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_{\mathbf{x}|\mathbf{y}}, \mathbf{P}_{\mathbf{x}|\mathbf{y}}),$$

where

$$\boldsymbol{\mu}_{\mathbf{x}|\mathbf{y}} = \boldsymbol{\mu}_\mathbf{x} + \mathbf{P}_{\mathbf{xy}}\mathbf{P}_{\mathbf{yy}}^{-1}(\mathbf{y} - \boldsymbol{\mu}_\mathbf{y}), \tag{6a}$$

$$\mathbf{P}_{\mathbf{x}|\mathbf{y}} = \mathbf{P}_{\mathbf{xx}} - \mathbf{P}_{\mathbf{xy}}\mathbf{P}_{\mathbf{yy}}^{-1}\mathbf{P}_{\mathbf{yx}}. \tag{6b}$$

To implement this calculation in terms of square-root covariance, we need to find an upper triangular matrix $\mathbf{S}_{\mathbf{x}|\mathbf{y}}$ such that $\mathbf{S}_{\mathbf{x}|\mathbf{y}}^\mathsf{T}\mathbf{S}_{\mathbf{x}|\mathbf{y}} = \mathbf{P}_{\mathbf{x}|\mathbf{y}}$. To do this, assume we have the following factorisation of the joint covariance:

$$\begin{bmatrix}\mathbf{P}_{\mathbf{yy}} & \mathbf{P}_{\mathbf{yx}}\\ \mathbf{P}_{\mathbf{xy}} & \mathbf{P}_{\mathbf{xx}}\end{bmatrix} = \begin{bmatrix}\mathbf{S}_1^\mathsf{T} & \mathbf{0}\\ \mathbf{S}_2^\mathsf{T} & \mathbf{S}_3^\mathsf{T}\end{bmatrix}\begin{bmatrix}\mathbf{S}_1 & \mathbf{S}_2\\ \mathbf{0} & \mathbf{S}_3\end{bmatrix}, \tag{7}$$

where $\mathbf{S}_1$ and $\mathbf{S}_3$ are upper triangular. Evaluating each block of the above reveals the following equations:

$$\begin{aligned}\mathbf{S}_1^\mathsf{T}\mathbf{S}_1 &= \mathbf{P}_{\mathbf{yy}},\\ \mathbf{S}_1^\mathsf{T}\mathbf{S}_2 &= \mathbf{P}_{\mathbf{yx}},\\ \mathbf{S}_2^\mathsf{T}\mathbf{S}_1 &= \mathbf{P}_{\mathbf{xy}},\\ \mathbf{S}_2^\mathsf{T}\mathbf{S}_2 + \mathbf{S}_3^\mathsf{T}\mathbf{S}_3 &= \mathbf{P}_{\mathbf{xx}}.\end{aligned}$$

Then, the term $\mathbf{P}_{\mathbf{xy}}\mathbf{P}_{\mathbf{yy}}^{-1}$ that appears in (6a) can be written as follows:

$$\mathbf{P}_{\mathbf{xy}}\mathbf{P}_{\mathbf{yy}}^{-1} = \mathbf{S}_2^\mathsf{T}\mathbf{S}_1(\mathbf{S}_1^\mathsf{T}\mathbf{S}_1)^{-1} = \mathbf{S}_2^\mathsf{T}\mathbf{S}_1\mathbf{S}_1^{-1}\mathbf{S}_1^{-\mathsf{T}} = \mathbf{S}_2^\mathsf{T}\mathbf{S}_1^{-\mathsf{T}},$$

and the conditional covariance (6b) can be written as

$$\begin{aligned}\mathbf{P}_{\mathbf{x}|\mathbf{y}} &= \mathbf{P}_{\mathbf{xx}} - \mathbf{P}_{\mathbf{xy}}\mathbf{P}_{\mathbf{yy}}^{-1}\mathbf{P}_{\mathbf{yx}} \\ &= \mathbf{P}_{\mathbf{xx}} - \mathbf{P}_{\mathbf{xy}}\mathbf{S}_1^{-1}\mathbf{S}_1^{-\mathsf{T}}\mathbf{P}_{\mathbf{yx}} \\ &= (\mathbf{S}_2^\mathsf{T}\mathbf{S}_2 + \mathbf{S}_3^\mathsf{T}\mathbf{S}_3) - \mathbf{S}_2^\mathsf{T}\mathbf{S}_1\mathbf{S}_1^{-1}\mathbf{S}_1^{-\mathsf{T}}\mathbf{S}_1^\mathsf{T}\mathbf{S}_2 \\ &= \mathbf{S}_2^\mathsf{T}\mathbf{S}_2 + \mathbf{S}_3^\mathsf{T}\mathbf{S}_3 - \mathbf{S}_2^\mathsf{T}\mathbf{S}_2 \\ \mathbf{S}_{\mathbf{x}|\mathbf{y}}^\mathsf{T}\mathbf{S}_{\mathbf{x}|\mathbf{y}} &= \mathbf{S}_3^\mathsf{T}\mathbf{S}_3.\end{aligned}$$

Since $\mathbf{S}_3$ is upper triangular, we can set $\mathbf{S}_{\mathbf{x}|\mathbf{y}} = \mathbf{S}_3$. Therefore, the measurement correction (6) can be expressed in square-root form using a single triangular solve as follows:

$$\boldsymbol{\mu}_{\mathbf{x}|\mathbf{y}} = \boldsymbol{\mu}_{\mathbf{x}} + \mathbf{S}_2^\mathsf{T}\mathbf{S}_1^{-\mathsf{T}}\left(\mathbf{y} - \boldsymbol{\mu}_{\mathbf{y}}\right), \tag{8a}$$

$$\mathbf{S}_{\mathbf{x}|\mathbf{y}} = \mathbf{S}_3. \tag{8b}$$

Complete the following tasks:

a) Create an empty function called `conditionGaussianOnMarginal` in `src/gaussian.cpp` and place its forward declaration in `src/gaussian.h` with the following function prototype:

```
void conditionGaussianOnMarginal(const Eigen::VectorXd & muyxjoint, const Eigen::
    MatrixXd & Syxjoint, const Eigen::VectorXd & y, Eigen::VectorXd & muxcond, Eigen::
    MatrixXd & Sxcond);
```

b) Review the `GIVEN` blocks within the `SCENARIO` block in `test/src/conditionGaussianOnMarginal.cpp` that correspond to the following cases:

i) `ny` $= 1$ and `nx` $= 1$

ii) `ny` $= 3$ and `nx` $= 1$

iii) `ny` $= 1$ and `nx` $= 3$

iv) `ny` $= 3$ and `nx` $= 3$

c) For each of the cases above, create BDD-style unit tests to assert that

i) The dimensions of $\boldsymbol{\mu}_{\mathbf{x}|\mathbf{y}}$ and $\mathbf{S}_{\mathbf{x}|\mathbf{y}}$ are valid.

ii) $\mathbf{S}_{\mathbf{x}|\mathbf{y}}$ is upper triangular.

iii) $\boldsymbol{\mu}_{\mathbf{x}|\mathbf{y}}$ and $\mathbf{P}_{\mathbf{x}|\mathbf{y}} = \mathbf{S}_{\mathbf{x}|\mathbf{y}}^\mathsf{T}\mathbf{S}_{\mathbf{x}|\mathbf{y}}$ match expected values `muxcond_exp` and `Pxcond_exp` respectively.

d) Run `ninja` and verify that appropriate the tests *fail*, since `conditionGaussianOnMarginal` has not yet been implemented. If your tests *crash*, include appropriate `REQUIRE` assertions in the tests as needed.

e) Implement `conditionGaussianOnMarginal` from (8).

Tip

Use the solve member function for triangular views to implement the conditional mean.

f) Run `ninja` and verify that all unit tests pass.