We discussed several topics during this week and here is a short reminder note with some references to the MatLab course notes 'Matlab course.pdf'. All explicit chapter, page and statement references in this short note will refer to that document. Although the notes are intended for MatLab coding, the vast majority of the concepts therein are easily ported to Python with minor syntax changes (that will be highlighted when necessary). The MatLab notes are quite a bit more mathematical in nature and Chapter 3 ('A crash course about matrices') provides a refresher on some of the fundamental linear algebra concepts that should come in handy to any mathematical programmer. Note, however, that sub-chapters 3.2 and especially 3.3 should become understandable after a bit more linear algebra course work.

**The correspondence between matrices and linear maps** This is discussed in pages 12-14 and the key idea is contained in Theorem 3.0.3, namely that any linear map between (finite dimensional) vector spaces can be uniquely identified with a matrix, i.e., one can obtain the same outcome from applying a linear map to vectors in the domain as from applying a matrix to the same vectors.

**The notion of matrix multiplication** This is discussed starting with Definition 3.0.4 until sub-chapter 3.1. The crucial idea is that what is referred to as matrix multiplication comes with a very nice property: composing linear maps is equivalent to multiplying their associated matrices. The examples on pages 16 and 17 and particularly nice to think about. Proposition 3.0.5 and Lemma 3.0.6 contain another criterion for checking if a matrix is invertible (in terms of its determinant) and a discussion reminiscent about the idea that a non-trivial linear combination of rows prevents a matrix from being invertible which we discussed in the context of linear systems.

In terms of Python, the important methods to remember here are *matmul* for multiplying two arrays (e.g. two matrices or a matrix and a vector; note that this is the correct way to multiply matrices in Pyton, using the asterisk symbol or the *multiply* command will perform pointwise multiplication) and *matrix_power* for computing powers of a matrix with simple syntax. For example, given a square matrix (array) $A$, the following two options will both compute $A^3$:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.matmul(np.matmul(A, A), A)
C = np.linalg.matrix_power(A, 3)
print(np.array_equal(B, C))
```

**Adjacency matrix of a simple graph as an application of matrix multiplication** A more in depth discussion of the notion of a simple graph and the construction of the associated adjacency matrix is contained in sub-chapter 3.4.

The interesting application is Proposition 3.4.5 which gives a mathematically simple (but usually suboptimal from an implementation complexity point of view) way of counting paths of length $k$ in a graph by inspecting the entries of the $k$-th power of the adjacency matrix. This can be extended (with the same complexity caveat) to a simple algorithm for checking if a graph is connected (see Problem 3.4.6).

**Solving linear systems** The various types of linear systems (consistent, over- and under-determined) are discussed in sub-chapter 3.1. The crucial insight is that if we are dealing with a square coefficient matrix for a system, linear algebra gives us a full characterization of when we can find solutions to the system:

- If the coefficient matrix $A$ is invertible, then we can always find a unique solution which is given by $A^{-1}b$, where $b$ is the result vector. We can compute this in Python either with an explicit computation of the invers (not recommended)

  ```
  import numpy as np
  A = np.array([[1, 2], [3, 4]])
  b = np.array([[5], [6]])
  print(np.matmul(np.linalg.inv(A), b))
  ```

  or by making use of the built-in linear function solver (recommended)

  ```
  print(np.linalg.solve(A, b))
  ```

- If the coefficient matrix $A$ is not invertible, this implies that the rows of $A$ satisfy some non-trivial linear combination and we have two subcases:
  - If the linear combination is also satisfied by the result vector $b$ (i.e., the system is consistent), then we can disregard one or more of the equations (one or more of the rows of $A$) as they do not add any information useful for finding the unknowns. The reduced system will therefore be under-determined and will have infinitely many solutions.
  - Otherwise the system is not consistent and cannot have any solutions.

In short, the linear algebra insight is that a system with $n$ equations and $n$ unknowns admits a unique solution if and only if the coefficient matrix is invertible.

As an additional note, if we are dealing with an under-determined system, we should expect to have infinitely many solutions. However, just like in the case of least squares minimization, there is a way of finding 'an optimal solution' for the system, in the sense of a solution that has minimal Euclidean norm (the analogy comes from the fact that in the case of least squares minimization, we were looking for the solution that minimized the Euclidean norm of the error vector). Again, this has a precise linear algebra formulation in terms of the Moore-Penrose pseudoinverse (discussed in detail in sub-chapter 3.2, although that might be difficult to follow without some more linear algebra experience).

Consider Problem 3.1.3. With some simple geometry, it is clear that the two planes intersect along the line with equation $2x + 5y = -7$. We can write this under-determined system (a single equation but two unknowns) in matrix form as

$$\begin{pmatrix}2 & 5\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix} = -7$$

with the coefficient matrix $A = \begin{pmatrix}2 & 5\end{pmatrix}$ and the result vector $b = -7$. Clearly, $A$ is not a square matrix, so it will not have an inverse. It will however have a pseudoinverse which can be computed in Python with

```
import numpy as np
A = np.array([[2, 5]])
Apinv = np.linalg.pinv(A)
print(Apinv)
```

Now we can find the 'optimal' solution (i.e., the one with minimal Euclidean norm or equivalently closest to the origin in Euclidean distance) in an analogous fashion to how we would compute it if we had an actual inverse for $A$:

```
solution = Apinv*b
```

Note that since $b$ is a scalar, we use the asterisk operator instead of *matmul*. Finally, to get the minimal distance, we just need to compute the Euclidean (or 2-norm) of the *solution*

```
print(np.sqrt(solution[0]**2 + solution[1]**2))
print(np.linalg.norm(solution, 2))
```

From a purely mathematical point of view, finding a solution in the case of an invertible coefficient matrix is clear. Numerically, however, things can go wrong if we have a coefficient matrix that is 'almost not invertible'. Consider the following two systems:

$$\begin{cases} x + y & = 2 \\ x + 1.001y & = 2 \end{cases} \qquad \begin{cases} x + y & = 2 \\ x + 1.001y & = 2.001 \end{cases}$$

They have the same invertible coefficient matrix $A = \begin{pmatrix}1 & 1\\1 & 1.001\end{pmatrix}$ and two very similar (i.e., very close, just $10^{-3}$ apart) result vectors $b_1 = \begin{pmatrix}2\\2\end{pmatrix}$ and $b_2 = \begin{pmatrix}2\\2.001\end{pmatrix}$. The solutions to these systems are quite different though.

```
A = np.array([[1, 1], [1, 1.001]])
b1 = np.array([[2], [2]])
b2 = np.array([[2], [2.001]])
print(np.linalg.solve(A, b1))
print(np.linalg.solve(A, b2))
```

Namely, for the first system the solution is $\begin{pmatrix}2\\0\end{pmatrix}$ and for the second the solution is $\begin{pmatrix}1\\1\end{pmatrix}$. The two solutions differ in each coordinate by 1, despite the fact that the coordinate difference between the result vectors was only $10^{-3}$. The issue here is that the coefficient matrix $A$ is 'almost' not invertible. Indeed, its rows are 'almost' linearly dependent, since if we subtract the first row from the second, we get 'almost' 0. This is measured by the so-called *condition number* of the coefficient matrix. In loose terms, the condition number measures the worse possible change in the solution if we perturb the result vector by a unit. It has a precise definition in terms of the 2-norms of the matrix and its inverse, but in Python it can simply be computed by

```
print(np.linalg.cond(A, 2))
```

In this case, the condition number of $A$ is just above 4000, explaining the thousand-fold discrepancy between the solutions given by a single unit of change in the result vectors.

From a numerical point of view, dealing with *ill conditioned* systems (i.e., systems that have coefficient matrices with large condition number) is very tricky, since result vectors are often obtained via empirical measurements (that come with non-negligible error bars) and therefore the solutions can vary wildly depending on the accuracy of the results. There is no universal fix to this issue, but good practice involves performing computations in the most numerically stable ways (for example using *linalg.solve* instead of computing the inverse and multiplying it with the result vector) and repeating measurements whenever possible to get a more robust average value of the result vectors.