

Numerical methods for PDEs

Stefano Piani

December 30, 2020

“ ... Partial differential equations are ubiquitous in mathematically-oriented scientific fields, such as physics and engineering. For instance, they are foundational in the modern scientific understanding of sound, heat, diffusion, electrostatics, electrodynamics, fluid dynamics, elasticity, general relativity, and quantum mechanics. They also arise from many purely mathematical considerations, such as differential geometry and the calculus of variations...”¹

¹Taken from the Wikipedia page about PDEs

What methods do we have if we want to solve a PDE?

- ▶ Finite difference methods
- ▶ Finite volume methods
- ▶ Finite element methods

FINITE DIFFERENCE METHODS

The main idea of this method is to approximate the derivatives with some expressions that involve the values of the function at some points

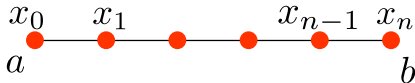
$$u'(x_0) \approx \frac{u(x_0 + h) - u(x_0)}{h}$$

$$u''(x_0) \approx \frac{u(x_0 + h) + u(x_0 - h) - 2u(x_0)}{h^2} \quad (1)$$

Let us suppose that $f(x) \in C^0[a, b]$ is a continuous function, that $k_1, k_2 \in \mathbb{R}$ are two real numbers, and that you want to solve the problem

$$\begin{cases} u''(x) = f(x) \\ u(a) = k_1 \\ u(b) = k_2 \end{cases}$$

Then you may choose some equally spaced points $x_0, \dots, x_n \in [a, b]$ with $x_0 = a$ and $x_n = b$.

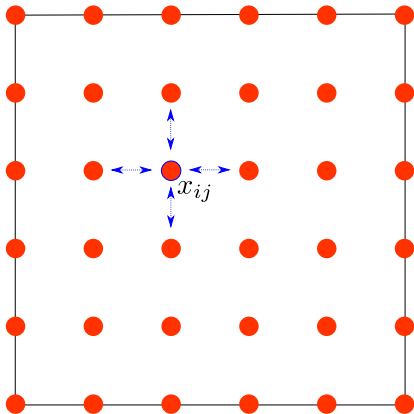


If you use the approximation defined in 1 then the differential problem can be approximate by the following linear system

$$\left\{ \begin{array}{l} \frac{u_0 + u_2 - 2u_1}{h^2} = f(x_1) \\ \frac{u_1 + u_3 - 2u_2}{h^2} = f(x_2) \\ \vdots \\ \frac{u_{n-2} + u_n - 2u_{n-1}}{h^2} = f(x_{n-1}) \\ u_0 = k_1 \\ u_n = k_2 \end{array} \right.$$

We consider u_k an approximation of $u(x_k)$ (and therefore we know $u(x)$ on $n + 1$ points).

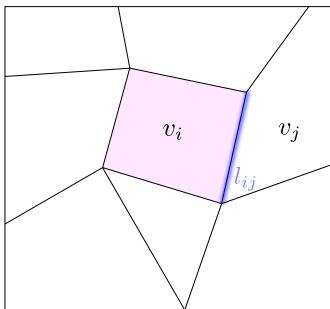
Of course, this can be also extended in more than one dimension



FINITE VOLUME METHODS

Let us suppose that the physical system that we are modeling follows a conservation law (for example, conservation of mass, of energy, of charge...). For this example, we will use the thermal energy and, therefore, we are looking for the function $u(x)$ that describes the temperature of the point x .

We divide our domain in some regions (or intervals, if we are in dimension 1) v_0, \dots, v_n called *volumes*



The boundary between region v_i and region v_j is called l_{ij} . We will use $\mu(v_i)$ for the area of the volume v_i .

We are interested in a value u_i for each volume that is an approximation of the average of the temperature function u on the volume v_i

$$u_i \approx \frac{1}{\mu(v_i)} \int_{v_i} u(x) dx$$

Let us suppose that the temperature $u(x)$ does not depend on the time (i.e. the temperature is fixed). This means that u_i and u_j do not depend on t .

If u_i is smaller than u_j (v_i is colder than v_j) some thermal energy will flow from v_j to v_i .

On the other side, we now that u_i is fixed. This means that the same amount of energy (per time unit) that goes inside through l_{ij} must go outside through the other sides.

If we call l_{i0}, \dots, l_{i3} the sides of v_i and we indicate with L_{ij} the amount of energy that goes through l_{ij} , we can say that

$$L_{i0} + L_{i1} + L_{i2} + L_{i3} = 0$$

If, instead, there is a source of heating f inside v_i , then the right hand side of the previous equation becomes

$$L_{i0} + L_{i1} + L_{i2} + L_{i3} = \int_{v_i} f(x) \, dx$$

To apply the finite volume method:

- ▶ Approximate every L_{ij} with an expression that depends linearly on the values of u_i and u_j (and on the geometry of the grid). For example, it can be

$$L_{ij} = \alpha \frac{u_j - u_i}{d_{ij}}$$

where d_{ij} is the distance between the centers of the volumes v_i and v_j .

- ▶ Write a linear system for u_0, \dots, u_n putting together the equations

$$L_{i0} + L_{i1} + L_{i2} + L_{i3} = 0$$

for every volume v_i (L_{ij} must be written as a function of u_i and u_j).

- ▶ Solve the linear system

FINITE ELEMENT METHODS

To apply the finite element method, you need to rewrite your problem in a weak form, i.e. it must be something like

Find a function $u(x) \in S$ such that for every function $v(x) \in R$ the following holds

$$A(u, v) = F(v)$$

where S and R are functional spaces, $A: S \times R \rightarrow \mathbb{R}$ is a bilinear coercive continuous function and $F: R \rightarrow \mathbb{R}$ is a linear continuous functional.

The idea of the method is to restrict the spaces S and R to some finite dimensional spaces S_h and R_h .

A common choice for the spaces S_h and R_h is to divide the domain in some regions called *elements* and choose S_h and R_h as the spaces of the functions that are continuous and whose restriction on each element is a polynomial of a fixed degree d .

Finite element methods are a little bit too complex to be summarized here. Maybe we will spend some time on them during the next tutorial

WHAT METHOD SHOULD I CHOOSE?

It depends: each method has its own strengths and weaknesses.

Finite difference method:

- Easy to understand
- Easy to implement
- Possibility to achieve high order accuracy
- Not suitable for complex geometry
- Not suitable for adaptivity
- You get the value of the solution only in a fixed number of points

Usually, it is used for meteorological and astrophysical simulations, where the geometry of the domain is simple.

Finite volume method:

- Requires a conservative law
- It is conservative (by construction)
- Works with complex geometry
- Extremely stable
- Allows adaptivity
- Low order accuracy
- You get only the average of the solution on the volumes (but not the values)

It is used everywhere there is a conservative law. In particular, it is the most common method used for CFD simulations (Computational Fluid Dynamics) and for industrial applications.

Finite element method:

- Requires a problem in a weak form
- Extremely versatile
- A lot of possibilities
- Difficult to implement
- Difficult to tune
- Suitable to complex geometry
- Allows adaptivity
- Possibility to achieve high order accuracy
- You get an approximation of the solution in each point of the domain
- Easy to prove that they converge (it is written in any book)
- The previous point is false!

They are well known for mechanical simulations, but they are used more or less in any field. In particular, when the problem is very complex (for example, multiphysics simulations), they are a good choice.

Time for some Python!

You should be already familiar with the heat equation for a one-dimensional object

$$\frac{du}{dt} = \alpha \frac{d^2u}{dx^2}$$

where α is a positive real number called *thermal diffusivity*. For example, if you put $\alpha = 1$ you are solving the problem for a rod made of ice (length is in mm, time is in seconds). For iron, set $\alpha = 23\text{mm}^2/\text{s}$.

We will wait until the temperature of the rod is constant. In this way,

$$\frac{du}{dt} = 0$$

Therefore, our equation will become

$$\frac{d^2u}{dx^2} = 0$$

We also add some considerations about the boundary conditions: instead, we suppose that we have measured the temperature of the rod at the top and at the bottom and that we discovered that the temperature is k_0 at the bottom $u(0)$ and it is k_l at the top $u(l)$.

Therefore, our problem is

$$\begin{cases} \frac{d^2u}{dx^2} = 0 \\ u(0) = k_1 \\ u(l) = k_2 \end{cases}$$

You already solved this problem with two different methods: using FDM and FEM.

What I propose today is to solve the same problem on a plate

Of course, in this case, we must consider the derivatives respect to x but also respect to y . Indeed, our equations will become

$$\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = 0$$

The sum of all the second derivatives usually is indicated with the symbol Δu or $\nabla^2 u$ (we will use the first one) and is called *laplacian* operator.

Let $D \subseteq \mathbb{R}^2$ be a square whose bottom left corner is the point $(0, 0)$ and let Γ be the boundary of the domain D . Let $d_\Gamma \in C^0(\Gamma)$ be a continuous function on the boundary Γ . We consider the following problem

$$\begin{cases} \Delta u = 0 & \text{on } D \\ u = d_\Gamma & \text{on } \Gamma \end{cases}$$

We want to solve this problem using FDM.

We need to approximate Δu using finite differences. We now that

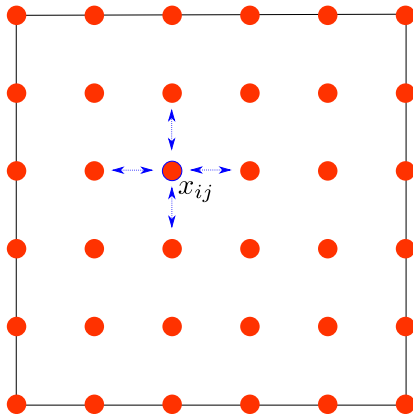
$$\frac{d^2}{dx^2}u|_{(x_0,y_0)} \approx \frac{u(x_0 + h, y_0) + u(x_0 - h, y_0) - 2u(x_0, y_0)}{h^2}$$

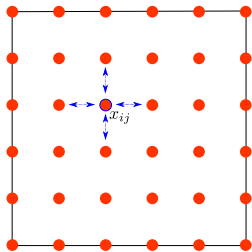
$$\frac{d^2}{dy^2}u|_{(x_0,y_0)} \approx \frac{u(x_0, y_0 + h) + u(x_0, y_0 - h) - 2u(x_0, y_0)}{h^2}$$

Therefore, we have that

$$\begin{aligned}\Delta u(x_0, y_0) &= \frac{d^2}{dx^2}u|_{(x_0,y_0)} + \frac{d^2}{dy^2}u|_{(x_0,y_0)} \\ &= \frac{1}{h^2} \left(u(x_0 + h, y_0) + u(x_0 - h, y_0) + u(x_0, y_0 + h) \right. \\ &\quad \left. + u(x_0, y_0 - h) - 4u(x_0, y_0) \right)\end{aligned}$$

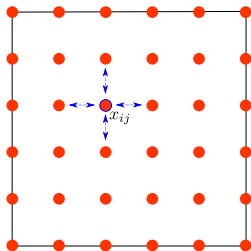
This is the so called *five-point stencil*





We want to try to understand the structure of the matrix that we need to solve.

First of all, if we have n points per side (for example, in our picture n is 6) we will have n^2 unknowns. Therefore, we will have a $n^2 \times n^2$ matrix (again, in our example, the matrix will have 1296 entries).



We decide that u_0 is located at the top left corner and that u_1 is at its right (at the same level). We go on in this way and, therefore, the last dot of the top line will be u_5 . The first element of the second row is u_6 while u_7 is the second element of the same row.

The seventh equation, therefore, will involve u_7 , the points at its right and left on the same row (u_6 and u_8) and the points on the same column (u_1 and u_3).

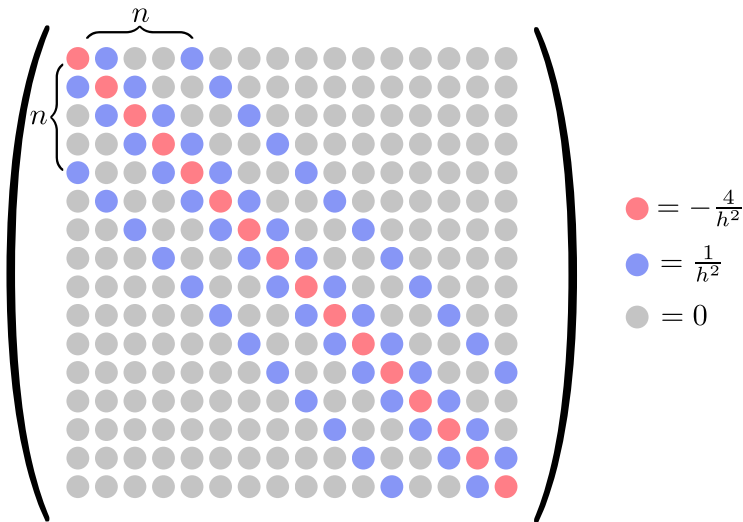
The seventh row of the matrix will then be

$$1/h^2 \cdot (0, 1, 0, 0, \dots, 0, 1, -4, 1, 0, 0, \dots, 1, 0, \dots, 0)$$

where -4 is at position 7.

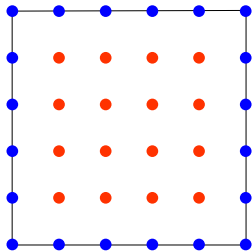
Our matrix has 5 diagonals. The main diagonal is filled with $-4/h^2$. One step above and one step below of this diagonal, all the elements are $1/h^2$. Finally, there are two more diagonals that have an offset that is exactly n (above and below) where n is the number of points per side.

The following picture shows an example for $n = 4$, i.e. there are 4 points per side and, therefore, 16 unknowns.



But what about the points, like u_2 for example, that do not have a neighbor up or down?

Luckily enough, they disappear because of the boundary conditions. Indeed, the only “real” unknowns that we have are the one that here are drawn in red. The blue ones are simply defined by the boundary conditions.



Let's write this part of the code!

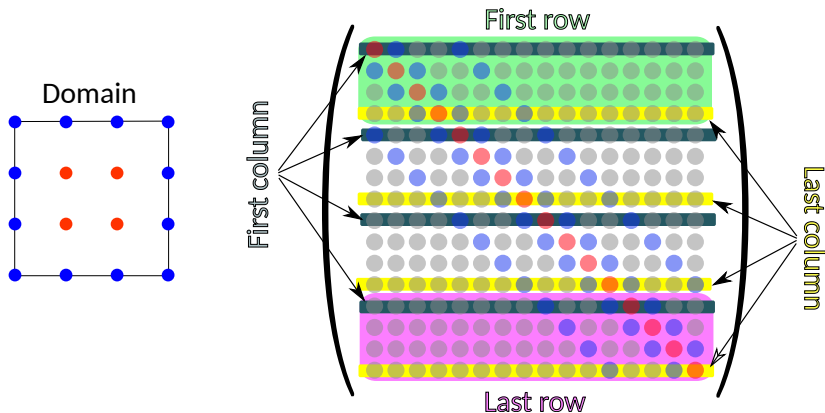
SYSTEM CONDENSATION

At some point, we realize that the variable u_i is a “blue” one, i.e. is constrained by the boundary conditions. This means that the equation that we have written in our matrix is wrong. We need therefore to fix it.

This process is called *condensation*

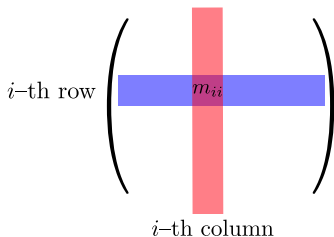
Condensation is much more common in FEM than in FDM, because usually in FDM it is possible to condense on the fly (i.e. to write directly the right matrix). Nevertheless, we need to condense our system because we have generated it without taking into account the boundary conditions.

For example, if we take a domain made of 4×4 nodes, then we have 16 unknowns, but 12 of them are fixed by the boundary conditions.



Therefore, we need only 4 equations. The other 12 equations (rows) of our matrix must be removed because they involve constrained nodes. How?

This is the matrix with the row and the column for the index i of u_i enlighten.



The diagram shows a matrix represented by large parentheses. A vertical red bar highlights the i -th column, and a horizontal blue bar highlights the i -th row. The intersection of these two bars is a purple square containing the label m_{ii} . The text " i -th row" is to the left of the blue bar, and " i -th column" is below the red bar.

The i -th row represents the equation

$$\frac{1}{h^2} (u_{i-l} + u_{i-1} - 4u_i + u_{i+1} + u_{i+l}) = k$$

If u_i is on the boundary, we do not want this equation inside our system!

We can simply delete this row! Therefore, we will remove the row i from our system (filling it with zeros).

Unfortunately, we are still in trouble because now we have a system with n variables and only $n - 1$ equations (because we have deleted a row). We need to completely remove the variable u_i from our system in order to have a system with $n - 1$ equations and $n - 1$ variables.

Therefore, we need to work on the i -th column, because it represents the equations in which u_i is present. So if on the row j we have that m_{ji} is not zero, this means that u_i appears as an unknown in the equation j . For example, the equation j could be

$$u_j + u_{j+i} + 2u_i = 3$$

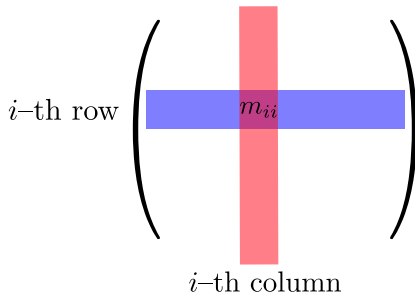
u_i is a constrained node and therefore we know in advance that the value of the unknown u_i must be a fixed value k (from the boundary conditions). Therefore, we would like to rewrite the previous equation as

$$u_j + u_{j+1} = 3 - 2k$$

in order to remove the variable u_i .

In this way, the unknown u_i completely disappear from our system.

To do that, we take the column i of the matrix and we multiply it by $-k$ (where k is the value that we know must be assumed by u_i). Now, we sum the result to the right hand side vector of our system and we remove (filling it with zeros) the column i from our system.



Now that the blue row and the red column are filled with zeros, the matrix is singular and therefore we can not solve our system! We need to put a value in the intersection (in the position m_{ii}) so that the matrix will be invertible again.

If 1 would be a perfect value from a mathematical point of view, it could be not numerically. It would be better to choose a number α that does not change the condition number of the matrix (for example, the average of the diagonal).

Finally, we can say that we have “disabled” the variable u_i from our system. Therefore, we can remove the right hand side of the equation i (that does not exists anymore) from the right hand side vector setting $r_i = 0$.

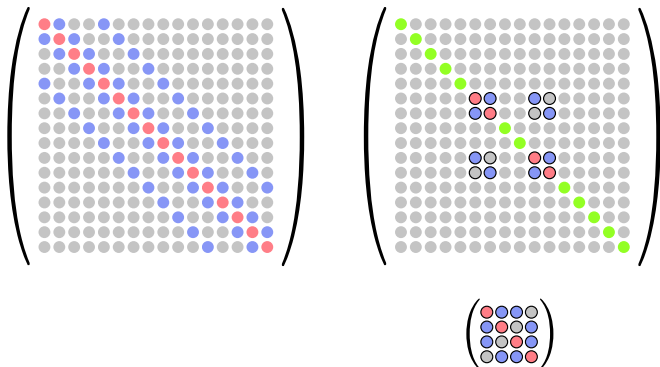
What we have obtained is a system of n equations; $n - 1$ of them involve only the $n - 1$ variables that are not u_i while the equation i is now just

$$\alpha u_i = 0$$

and is irrelevant for us.

Be aware that, when you will solve the matrix, you will not find a value for u_i inside the solution vector. Indeed, if s is the vector that you get solving the system, s_i will be just numerical noise. Therefore, after you solve the system, you need to set the constrained unknowns to the expected value by hand!

In this picture you can see the matrix for a system with 4×4 nodes before and after the condensation.



Because of the condensation, solving the condensed system is equivalent to solving a smaller system obtained copying all the elements m_{ij} whose indices i and j do not refer to a variable u_i (or u_j) that is constrained by the boundary conditions (in the picture, they are highlighted with a black circle).

But is this really convenient? Indeed, in our last example, we are solving a 16×16 system just to retrieve 4 numbers. Would it be more efficient to just solve a 4×4 system that just involves the non constrained variables?

It depends on what algorithm you are using, how it is implemented, how complex is your geometry, ...

For example, a possible different approach could be to copy in a new smaller matrix only the part of the original matrix that involves the unconstrained variables. We will see that, in general, this is a really bad idea!

In any case, there are a few observations that is worth to note.

- ▶ If your domain is a square and you are using 4 points per side, then you are assembling a system of 16 variables just for computing 4 of them. But if you increase the number of points, the overhead becomes not so evident. For example, if we take a reasonable real life scenario, let's say with 10^4 points per side, you are assembling a system with 10^8 variables to compute 99,960,004 of them, increasing the size of the system by only the 0.04%. This means that the overhead due to the fact that we have included the constrained variables inside our system becomes less and less relevant as soon as we increase the size of the problem.
- ▶ On the other side, copying such a big matrix just to decrease its size a little bit would be really expensive! In these cases, it is usually better to use condensation and keep the number of rows fixed.
- ▶ Usually, the time required to assemble the matrix and the time required to solve the system are comparable.

Therefore, this is the rule of thumb!

If your domain has a simple geometry, if it is easy to identify which variables are constrained and if building a matrix only for the unconstrained ones is as expensive as assembling the matrix for all the variables, then you can assemble the matrix only for the unconstrained nodes and forget about condensation. Lucky man!

In principle, we could have solved our problem in Python in this way; but, if we did, we would have lost the opportunity to learn how condensation works.

If, instead, taking care of the constrained variables slows down all the assembling procedure, then it is better to just build the system for all the variables and then apply condensation.

USING SPARSE MATRICES

How can a sparse matrix be implemented in a computer (for example, in Python)?

The easiest idea is to use a dictionary. Let us suppose you want to store the following matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -5 & 0 & 7 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{pmatrix}$$

then you can store the following dictionary:

$$(0,0) \rightarrow 1 \quad (1,2) \rightarrow -5 \quad (1,4) \rightarrow 7 \quad (2,1) \rightarrow 3 \quad \dots$$

This is really efficient if you want to insert or remove entries in the future.

But it is very inefficient for looping on columns and rows. Indeed, every operation that requires those loops will be very slow (therefore, matrix multiplications or solving linear systems)

Let us try with another structure

THE SPARSE ROW FORMAT

Another possibility is the following. We will use three one-dimensional arrays called `row_ptr`, `column_ptr` and `data`:

- ▶ Define a vector `row_ptr` of $n + 1$ elements (where n is the number of rows of the matrix) to store which elements of data are inside a particular row. In other words, if $k1 = \text{row_ptr}[i]$ and $k2 = \text{row_ptr}[i+1]$, then `data[k1:k2]` is the space reserved for the elements of the i -th row;
- ▶ `column_ptr` between $k1$ and $k2$ stores the position of the elements of the i -th row that are not zero (the index of their column);
- ▶ `data` between $k1$ and $k2$ stores the values of the elements of the i -th row that are not zero.

Let us go back to our example:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -5 & 0 & 7 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A possibility is

$$\text{row_ptr} = (0, 1, 3, 5, 7, 8)$$

$$\text{column_ptr} = (0, 2, 4, 1, \bullet, 3, \bullet, 0)$$

$$\text{data} = (1, -5, 7, 3, \bullet, 1, \bullet, 4)$$

The \bullet represents space that is allocated but it is not used. Therefore, adding another element on the third row will be easy (because there is a space ready), but adding another element on the first row requires to rewrite completely the matrix.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -5 & 0 & 7 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{pmatrix}$$

If there are no ●, the format is called *compressed*. In compressed row format, our matrix will become

$$\text{row_ptr} = (0, 1, 3, 4, 5, 6)$$

$$\text{column_ptr} = (0, 2, 4, 1, 3, 0)$$

$$\text{data} = (1, -5, 7, 3, 1, 4)$$

In CSR (Compressed Sparse Row) format is extremely easy to read a single row. It is instead very complicated to read a specific column.

They are very fast for summing, multiplying and solving linear systems. They are really inefficient if you want to add or remove entries. Moreover, slicing on rows is fast, but slicing on columns is slow!

If you need to slice on columns (but not on rows) you can use the CSC (Compressed Sparse Column) format. It is the same, but it works for columns instead of for rows (i.e. to save a matrix m in CSC format you must write the very same vectors that you would write if you had to save the transpose of m in CSR format).

Now we are in trouble! Indeed, CSR matrices seem to be ok for us, but we have no idea about how to construct them (if we add elements one by one, it will be extremely inefficient).

There are usually two options to solve this problem

- ▶ Create an object that is a *sparsity pattern*. This object knows where the matrix will store entries that are not zeros. Then you create your matrix starting from the sparsity pattern and eventually you start to fill the entries in the places that the sparsity pattern has made available.
- ▶ Rebuilding a matrix from a sparse format to another one is usually not so expensive. Therefore, we could create a matrix in a format that allows to add and remove entries efficiently and then rebuilt it in CSR format.

Of course, Python (and, in particular, Scipy) has a lot of tools ready to use for us!

Indeed, `scipy.sparse` module has a lot of sparse formats available.

The `dok_matrix` (Dictionary Of keys) is the sparse format that uses a dictionary to store the entries.

There are also the `csr_matrix` and the `csc_matrix`. In Python, all the matrices in sparse row format and sparse column format are always compressed!

There are no sparsity pattern objects in Scipy; therefore, use the `dok_matrix` and then build a `cs*_matrix` starting from it.

There are also other formats that are used less commonly, but that can be really useful in some specific cases.

For example, for our problem, there is the `dia_matrix` that remembers the entries as a list of diagonals!

Let see how it works... It's time to go back to Python!