Instructions to run the codes and ideas for the report

Joe & Víctor

Department of Aeronautics
Imperial College London

1 How to run the codes and contribute to the repository

First you need git clone the repository. Open a terminal and run

```
git clone https://github.com/victorballester7/sparseIMS.git
cd sparseIMS
```

If you have `uv` installed you should be able to run the codes as follows, for example:

```
uv run examples/dimension.py
```

If you don't have `uv` installed, then check the `README.md` file in the repository for instructions on how to set up a virtual environment and install the required packages.

To contribute to the repository, we need to follow these steps:

1. Type `git pull` to make sure you have the latest version of the repository.
2. First edit some of the files, add new codes, etc.
3. Type `git add .` to stage all the changes you have made.
4. Type `git commit -m "Your message here"` to commit the changes. Make sure to write a meaningful message that describes what changes you have made. For example: `git commit -m "Added new tridiagonal sparse matrix generator"`.
5. Type `git push` to push the changes to the remote repositoryso that others can see them.

The first time you do this, maybe you can try by just writing a comment in a any file and doing the `git add`, `git commit` and `git push` and then go to the repository in GitHub and check that your comment appears there. If you have any issues with git, please let us know and we will help you.

Organizing the code files

You should write your codes into the `src/ims/` folder. You can create several files and call them from a main file, as you wish.

Since the computational time required to run some of the tests can vary significantly depending on the machine used, it would be a good idea to at the end, only one of you run the final simulations that will contain the data for the report. This way we avoid discrepancies in the results.

Examples of how to generate the matrices, call the solvers and plot the results can be found in the `examples/` folder. The *decorator* called `@characterise` basically adds some code surrounding the main function to measure the computational time. For example this code:

```
@characterise(
    methods=[MatrixInverse, LU, Cholesky, QR],
    measures=[ExecutionTime],
    realisations=10,
)
def generate_symnetric(dimension: int) -> Tuple[np.ndarray, np.ndarray]:
    ...

    return A, b
```

will run the function `generate_symmetric` 10 times for each of the methods specified in the `methods` list, measuring the execution time for each method. That's why when you observe the plot in the examples provided you see 10 points for each method. We do this to assess the variability on how ill-posed the random matrices are. For each point, the linear system is solved `repetitions` · `iterations` times (default to 5 and 100, respectively) to get an average execution time. Let me explain how. We do `iterations` runs of the same linear system to get a good average execution time. Then we repeat this process `realisations` times. And among those sets of averages times we take the minimum one as the final execution time for that method and matrix size. If you want to change the values (maybe for large size matrices you want to reduce them, specially the `iterations`) you can modify the default values in `src/utils/measures/_execution_time.py`. Of course any changing here needs to later be reported in the report.

Available methods are:

- `MatrixInverse`: Solves the system by computing the matrix inverse.
- `LU`: Solves the system using LU decomposition of the matrix $\mathbf{A}$.
- `Cholesky`: Solves the system using Cholesky decomposition of the matrix $\mathbf{A}$. Note that this method only works for symmetric positive definite matrices.
- `QR`: Solves the system using QR decomposition of the matrix $\mathbf{A}$.
- `SparseLU`: Solves the system using Sparse LU decomposition of the matrix $\mathbf{A}$. It works for any time of matrix, but is specially powerful for sparse matrices whose inverse is also sparse.

There is only one measure available for now:

- `ExecutionTime`: Measures the execution time required to solve the linear system.

Thus, when calling `generate_symmetric(d)` in the example above, it won't return the matrix $\mathbf{A}$ and vector $\mathbf{b}$ but a dataframe with `realisations` (in the example above, 10) execution times for each method specified in the `methods` list, for a matrix of size `d`.

2 Ideas for the report

It would be very nice if we could write the report using LaTeX. LaTeX is a typesetting system that is extremely popular in academia, especially in mathematics and physics, due to its powerful handling of formulas and bibliographies. It also produces very professional-looking documents. I have created a template on Overleaf, which is an online LaTeX editor that allows for easy collaboration. In principle you should have received an invitation to join the project via email. If not, please open this link and you will be able to access and edit the project. Trust me, it's worth it is you could learn the basics of LaTeXat that young age. I learned it as well during high school and then in the university when all your collegues are learning it as well it's much easier. If you don't know how to write in math mode, I would recommend you to check this tutorial. In general, Overleaf docs are very good and you can find a lot of information there. Otherwise, google everything and it works as well in the forums, e.g. *integral sign in latex*, *how to pdf images in latex*, etc.

What should be in the report?

- Introduction to numerical linear algebra. Why is it so important to be able to solve linear systems efficiently? Some areas of applications? Same for sparse matrices (maybe as a subsection of this).
- Description of the goal of the project. E.g. *In this report we explore different methods to solve linear systems...*

- Description of the methods used: matrix inverse, LU, Cholesky, QR and SparseLU. When to use each one, when should we get and improvement in performance, etc. Particularly for the matrix inverse, research in the `scipy` docs how do they compute the inverse. **Important:** by description I don't mean going through the algorithm step by step but rather a high level description of how it works and for which matrices it is suitable. Also how do solve the linear system using each method.
- **Most important:** How and why we generate the random matrices used for testing.
- Description of the tests performed. What machine has been used to run the codes (e.g. CPU Architecture and/or Laptop Model), statistics done in the tests (explained in the section before)... Regarding the comparisons, it's up to you but at least I would include:

  A comparison of $\mathbf{Ax} = \mathbf{b}$ when

  1. $\mathbf{A}$ is a general dense matrix and $\mathbf{b}$ as well. We compare computational time of different methods vs size of the matrix.
  2. Same as 1. but now $\mathbf{A}$ is symmetric positive definite (positive definite means all the eigenvalues are positive).
  3. $\mathbf{A}$ is sparse but $\mathbf{b}$ is dense. We compare computational time of different methods vs density of the matrix.
  4. Same as 3. but now both $\mathbf{A}$ and $\mathbf{b}$ are sparse.
  5. $\mathbf{A}$ is sparse of fixed density and $\mathbf{b}$ is dense. Is there a difference between a sparse generated using the Greshgorin algorithm and a tridiagonal matrix of this form?

     $$\mathbf{B} = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

     Note that this matrix by definition has the density fixed to $\frac{3n-2}{n^2}$. So in order to compare with other sparse matrices we should generate them with the same density. We compare computational time of different methods vs size of the matrix.
  6. Same as 5. but now both $\mathbf{A}$ and $\mathbf{b}$ are sparse.
- Comparison of the results. Important, try to generate high quality plots that clearly show the differences between methods. Plots are very important in a report, because it's what people will look at first. We should export them in .pdf from Python, to then include them in high vector-graphics quality in the report.
- Conclusions

Most important, be concise and clear. Don't digress too much in unnecessary details.

As always any questions please let us know. Don't waste too much time in git issues or LaTeX issues. Write us as soon as you have any problem that doesn't let you move forward. For the math/technical problems, it's always better to discuss first between yourselves and then if you can't solve it, write us.