```
begin
    import Pkg
    Pkg.add("MAT")
end
```

```
begin
    using PlutoUI
    using MAT
    using LinearAlgebra, Statistics
    using Plots
end
```

Project : Reduced-order modeling of the two-dimensional cylinder flow

The two-dimensional cylinder flow is a canonical of bluff body flows in fluid dynamics. Despite its simplicity, this two-dimensional flow captures some fundamental features of larger scale flows as shown in the image below.

This flow pattern is known as the **Bénard-von Kàrmàn vortex street** and corresponds to the periodic shedding of vortices from the cylinder (or the island in this image above).

From a mathematical point of view, the dynamics of this flow can be modeled by the incompressible Navier-Stokes equations

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) = -\nabla p + \frac{1}{Re} \nabla^2 \mathbf{u}$$
$$\nabla \cdot \mathbf{u} = 0$$

where $\mathbf{u}$ is the two-dimensional velocity field, p the pressure field and Re the Reynolds number (i.e. the ratio of the inertial and viscous forces). In order to simulate these equations on a computer, they are typically discretized on a mesh with n points such that $\mathbf{u} \in \mathbb{R}^{2n}$ and $p \in \mathbb{R}^n$ where n can be of the order of several hundred thousands. From a dynamical point of view however, the dynamics are rather simple and correspond to simple periodic dynamics. As such, dynamically, only two degrees of freedom (e.g. the amplitude and the phase of the oscillation) are needed to characterize the state of the system rather than n degrees. In this notebook, you will have to analyze these dynamics using the different tools discussed during the class (i.e. dimensionality reduction using PCA or DMD, sparse sensor placement, least-squares and compressive sensing). For that purpose, you'll be given a dataset consisting in snapshots of the vorticity field (i.e. $\omega = \nabla \times \mathbf{u}$) over time obtained by direct numerical simulation of the Navier-Stokes equations. A random selection of these snapshots is shown below.

```
md"
# Project : Reduced-order modeling of the two-dimensional cylinder flow

The two-dimensional cylinder flow is a canonical of bluff body flows in fluid dynamics.
Despite its simplicity, this two-dimensional flow captures some fundamental features of larger scale flows as shown in the image below.

![](https://cdn.iflscience.com/images/091d5090-e168-5128-ba1f-8a06113f54c7/large-1464354324-40-von-karman-vortex-streets.jpg)

![](https://giphy.com/gifs/lU9KoIIYUoE088ilzI/html5)

This flow pattern is known as the [Bénard-von Kàrmàn vortex street](https://en.wikipedia.org/wiki/K%C3%A1rm%C3%A1n_vortex_street) and corresponds to the periodic shedding of vortices from the cylinder (or the island in this image above).

From a mathematical point of view, the dynamics of this flow can be modeled by the incompressible Navier-Stokes equations

```math
\begin{aligned}
    \dfrac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \left( \mathbf{u} \otimes \mathbf{u} \right) & = - \nabla p + \frac{1}{Re} \nabla^2 \mathbf{u} \\
\nabla \cdot \mathbf{u} & = 0
\end{aligned}
```

where $\mathbf{u}$ is the two-dimensional velocity field, p the pressure field and Re the Reynolds number (i.e. the ratio of the inertial and viscous forces).
In order to simulate these equations on a computer, they are typically discretized on a mesh with n points such that $\mathbf{u} \in \mathbb{R}^{2n}$ and $p \in \mathbb{R}^n$ where n can be of the order of several hundred thousands.
From a dynamical point of view however, the dynamics are rather simple and correspond to simple periodic dynamics.
As such, dynamically, only two degrees of freedom (e.g. the amplitude and the phase of the oscillation) are needed to characterize the state of the system rather than n degrees.
```

```
In this notebook, you will have to analyze these dynamics using the different tools discussed during the class (i.e. dimensionality reduction using PCA or DMD, sparse sensor placement, least-squares and compressive sensing).
For that purpose, you'll be given a dataset consisting in snapshots of the vorticity field (i.e. $\omega = \nabla \times \mathbf{u}$) over time obtained by direct numerical simulation of the Navier-Stokes equations.
A random selection of these snapshots is shown below.
"
```

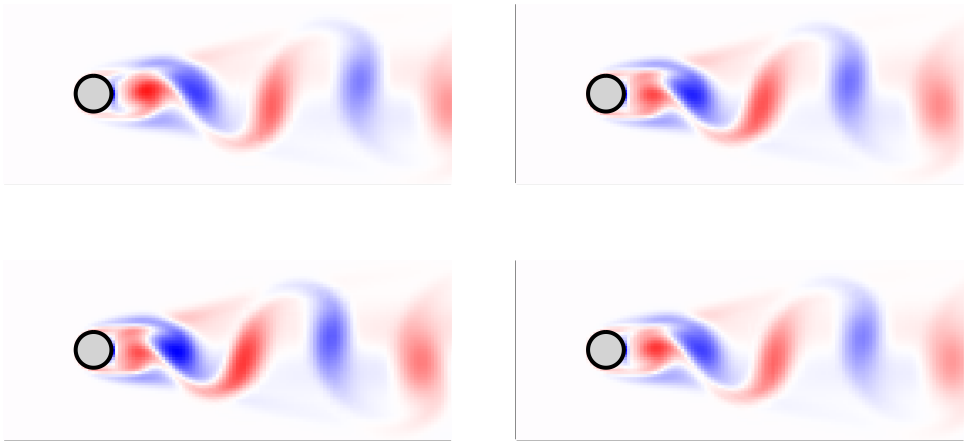
Dataset to be loaded : C:\Users\ASUS\Downloads\c

```
md"Dataset to be loaded : $@bind dset TextField()"
#C:\Users\ASUS\Downloads\cylinder_dataset.mat
```

```
begin
    # --> Load the dataset.
    data = matread(dset)

    # --> Extract the mesh and the snapshots collection.
    mesh, X = [data["x"][:], data["y"][:]], data["data"]
end;
```

Display random snapshots from the database : Click



```
begin
    go
    plot(
        plot_flow_field(mesh, X[:, rand(1:400)]),
        plot_flow_field(mesh, X[:, rand(1:400)]),
        plot_flow_field(mesh, X[:, rand(1:400)]),
        plot_flow_field(mesh, X[:, rand(1:400)]),
        layout=(2, 2),
        size=(512, 256),
        )
end

```

Dimensionality reduction

As a starting point, let us try to determine the intrinsic dimensionality of our dataset. For that purpose, we'll use *principal component analysis* (PCA). A brief recap' of PCA is given below.

Principal Component Analysis

PCA is closely related to the SVD matrix factorization. In this notebook, we will however use PCA through its correlation matrix formulation. Given a data matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ where each column corresponds to one snapshot of the cylinder flow simulation with m state variables and we have n such snapshots sampled uniformly in time, let us first compute the mean flow, i.e.

$$\overline{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i.$$

In a second step, the temporal correlation matrix

$$\mathbf{\Sigma_X} = \frac{1}{n-1}\sum_{i=1}^{n} (\mathbf{x}_i - \overline{\mathbf{x}})^T(\mathbf{x}_i - \overline{\mathbf{x}})$$

is constructed. Assuming that the mean flow has already been subtracted from the columns of $\mathbf{X}$ (`X .-= x̄` in Julia), this correlation matrix can easily be computed in one go as follows

$$\mathbf{\Sigma_X} = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}.$$

In a third step, the eigendecomposition of $\mathbf{\Sigma_X}$ is computed, i.e.

$$\mathbf{\Sigma_X V} = \mathbf{V \Lambda}$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ is the corresponding matrix of eigenvectors normalized such that $\mathbf{V}^T\mathbf{V} = \mathbf{I}$. The PCA modes can then be computed as

$$\mathbf{U} = \frac{1}{\sqrt{N-1}}\mathbf{X V \Lambda}^{-\frac{1}{2}}$$

while the projection of the snapshots into this particular basis is given by

$$\mathbf{a} = \mathbf{U}^T\mathbf{X}.$$

Application to the cylinder flow

It is now up to you. Given the dataset already uploaded in the notebook as `X`, use the cells below to :

1. Compute the mean flow `x̄` and plot it using the `plot_flow_field(mesh, x̄)` command.
2. Compute the covariance matrix `Σ` and plot it using `heatmap(Σ)`. How do you interpret this plot?
3. Compute the eigendecomposition of `Σ` using `eigen(Σ)` and plot the distribution of the eigenvalues. Given that each eigenvalue is directly related to the amount of turbulent kinetic energy captured by the corresponding PCA mode, how many modes do you need to keep in your analysis to capture 99% of the kinetic energy ?
4. Compute and plot the first four PCA modes using `plot_flow_field(mesh, U[:, i])`. Eventhough you may not be familiar with fluid dynamics, try to explain the patterns you observe.

```
Float64[4.60786e-21, -4.74338e-21, -6.09864e-22, 3.18484e-21, 3.79471e-21, -4.40457e
```

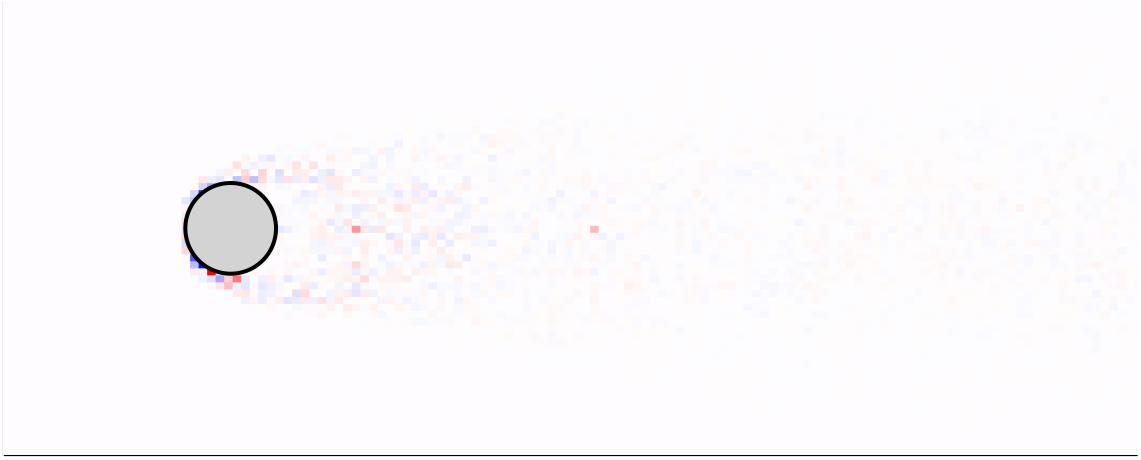
```
# --> Compute the mean flow.
begin
    m, n = size(X)
    sum = zeros(m)

    for i = 1:n
        sum .+= X[:,i]
    end

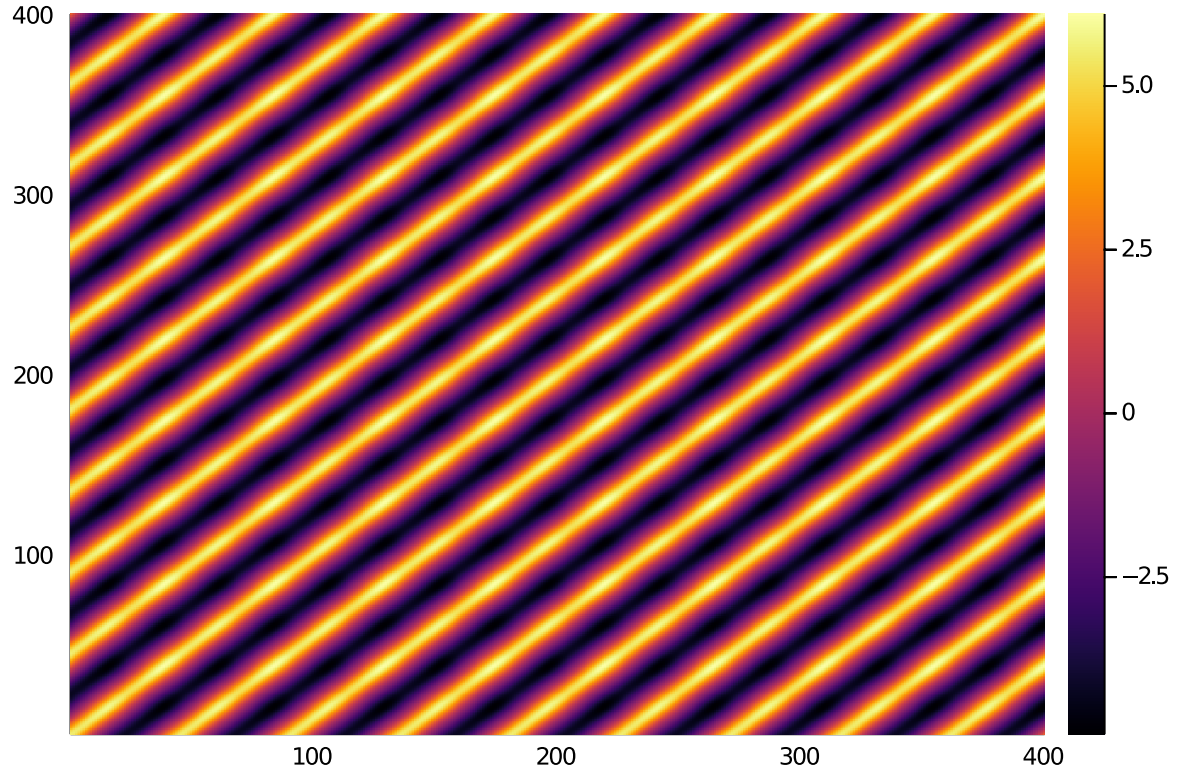
    x̄ = sum./n
end
```

```
(20769)
```

```
size(x̄)
```



```
plot_flow_field(mesh, x̄)
```




```
begin
    # --> Subtract mean from the data matrix.

    for i=1:n
        X[:,i] .-= x̄
    end

    # --> Compute the covariance matrix.

    Σ = X'*X./(n-1)

    # --> Plot the covariance matrix.

    heatmap(Σ)

end
```

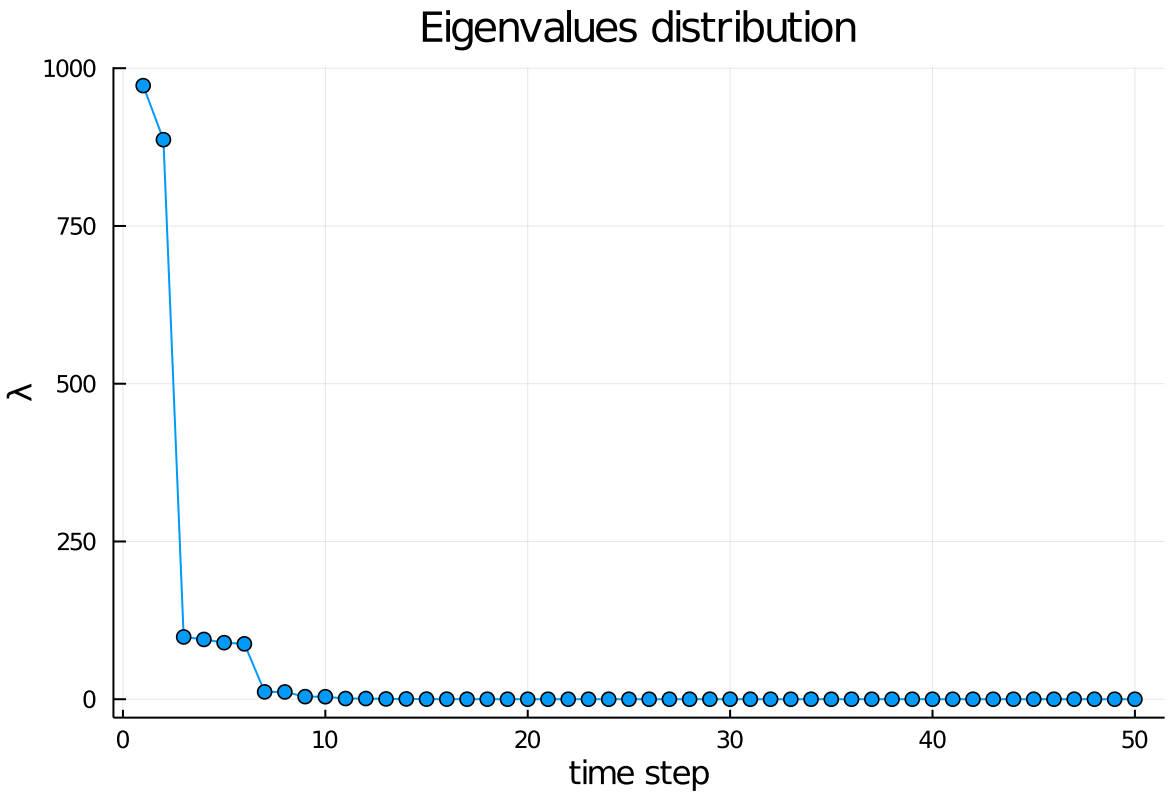
```
400×400 Array{Float64,2}:
  5.5835     5.43504    5.07754    4.55325   …   0.902087   1.46887    2.07674
  5.43504    5.52331    5.38615    5.03936       0.337835   0.854387   1.41015
  5.07754    5.38615    5.48579    5.35991      -0.184692   0.291688   0.798987
  4.55325    5.03936    5.35991    5.47153      -0.679681  -0.229677   0.239273
  3.9156     4.52291    5.02168    5.35492      -1.16266   -0.724569  -0.27999
  3.21765    3.88957    4.51005    5.02208   …  -1.64646   -1.20872   -0.773957
  2.50507    3.19348    3.87871    4.51283      -2.13866   -1.69586   -1.25954
  ⋮                                          ⋱
 -0.610965  -1.0888    -1.5639    -2.04323       4.90975    4.30653    3.64134
 -0.128455  -0.622525  -1.10247   -1.58052   …   5.47833    4.99256    4.38472
  0.372607  -0.150258  -0.644983  -1.12637       5.86344    5.55367    5.06133
  0.902087   0.337835  -0.184692  -0.679681      6.01925    5.929      5.6098
  1.46887    0.854387   0.291688  -0.229677      5.929      6.07324    5.9702
  2.07674    1.41015    0.798987   0.239273      5.6098     5.9702     6.09864
```

```
Σ
```

Interpretation :

The heatmap looks like an inversed tridiagonal matrix denoting the periodic nature of the data. We can notice that the direction shows that the covarience is negative since one variable decreases

while the other increases.




```
begin
    # --> Compute the eigendecomposition of Σ.

    λp, Vp = eigen(Σ)

    λ = sort(λp, rev=true)
    V = Vp[:,n:-1:1]

    Λ = Diagonal(λ)

    # --> Plot the distribution of eigenvalues.

    plot(λ[1:50], marker=".",
    xlabel="time step",
    ylabel="λ",
    leg=:false,
    title="Eigenvalues distribution",
    )

end
```

```
Λs = 100×100 SparseArrays.SparseMatrixCSC{Float64,Int64} with 100 stored entries:
       [1  ,   1]  =  972.397
       [2  ,   2]  =  886.729
       [3  ,   3]  =  98.6818
       [4  ,   4]  =  94.749
       [5  ,   5]  =  89.6219
       [6  ,   6]  =  87.7482
       ⋮
       [94 ,  94]  =  1.04214e-11
       [95 ,  95]  =  1.02543e-11
       [96 ,  96]  =  9.72302e-12
       [97 ,  97]  =  8.93746e-12
       [98 ,  98]  =  8.8084e-12
       [99 ,  99]  =  8.6241e-12
       [100, 100]  =  8.02154e-12
```

```
    Λs = Λ[1:100, 1:100]
```

```
Λsq = 100×100 Array{Float64,2}:
 0.0320685  0.0        0.0       0.0       …  0.0        0.0        0.0
 0.0        0.0335818  0.0       0.0          0.0        0.0        0.0
 0.0        0.0        0.100666  0.0          0.0        0.0        0.0
 0.0        0.0        0.0       0.102734     0.0        0.0        0.0
 0.0        0.0        0.0       0.0          0.0        0.0        0.0
 0.0        0.0        0.0       0.0       …  0.0        0.0        0.0
 0.0        0.0        0.0       0.0          0.0        0.0        0.0
 ⋮                                          ⋱
 0.0        0.0        0.0       0.0          0.0        0.0        0.0
 0.0        0.0        0.0       0.0       …  0.0        0.0        0.0
 0.0        0.0        0.0       0.0          0.0        0.0        0.0
 0.0        0.0        0.0       0.0          3.36939e5  0.0        0.0
 0.0        0.0        0.0       0.0          0.0        3.4052e5   0.0
 0.0        0.0        0.0       0.0          0.0        0.0        3.53078e5
```

```
    Λsq = Matrix(I,100,100)/Λs^(1/2)
```

7

```
begin
    ss = 0
    elmnt = 0
    for i=1:n
        ss += λ[i]
        if ss < 2241.986528676953
            elmnt += 1
        end
    end
    elmnt
end
```

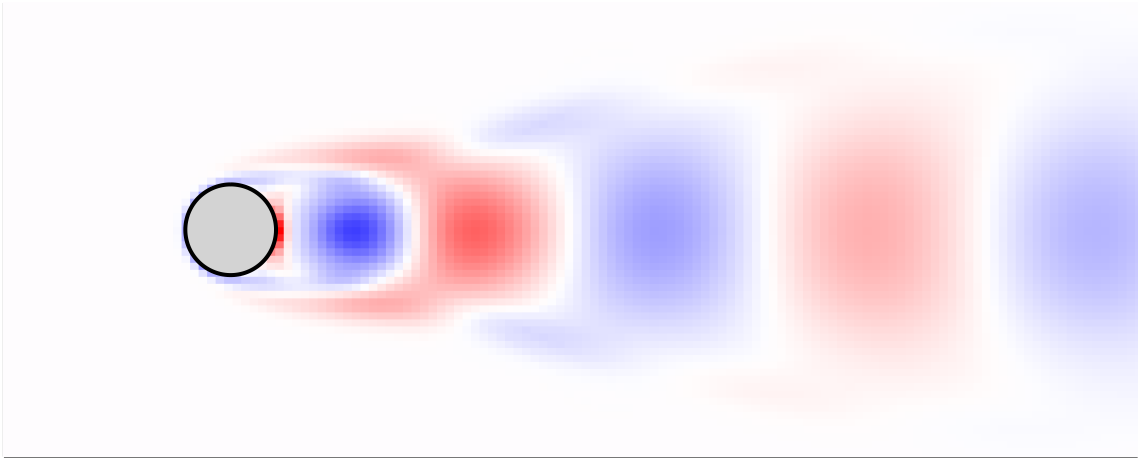
Each eigenvalue being related to the amount of turbulent kinetic energy within its respective PCA mode, we would need 7 eigen modes to keep in the analysis in order to capture 99% of the kinetic energy. Here 99% represents approximately 2241.98 which is 99% of the sum of the eigenvalues.

```
U = 20769×100 Array{Float64,2}:
 -3.46018e-6   1.37692e-6  -4.09182e-8   2.16315e-8  …  -6.90088e-5    2.67329e-5
 -3.47626e-6   1.3893e-6   -4.03592e-8   2.06432e-8     -2.20138e-5    3.03292e-5
 -3.49132e-6   1.40255e-6  -3.97073e-8   1.97293e-8     -4.87695e-6    1.86411e-5
 -3.50761e-6   1.4138e-6   -3.9224e-8    1.87091e-8     -4.36629e-5    5.00433e-5
 -3.52456e-6   1.424e-6    -3.88307e-8   1.7631e-8      -0.00010639    0.000104599
 -3.53885e-6   1.43719e-6  -3.81906e-8   1.67658e-8  …  -0.000122392   9.5931e-5
 -3.55466e-6   1.44816e-6  -3.77237e-8   1.57648e-8     -5.50721e-5    4.8315e-5
  ⋮                                                   ⋱
  1.56522e-6  -3.0167e-6   -1.78447e-7  -1.4641e-7        2.62093e-5   -1.49019e-5
  1.55773e-6  -3.00776e-6  -1.58547e-7  -1.25877e-7       4.42701e-5   -8.42917e-6
  1.54782e-6  -2.99865e-6  -1.4002e-7   -1.12861e-7  …    6.36775e-5    3.83412e-5
  1.53776e-6  -2.98957e-6  -1.21434e-7  -1.00357e-7       8.34041e-5    8.70112e-5
  1.52996e-6  -2.97817e-6  -1.09257e-7  -8.27941e-8       0.000101716   0.000130764
  1.52208e-6  -2.96498e-6  -9.69583e-8  -7.47447e-8       0.000118835   0.000148246
```

```
# --> Compute the PCA modes.

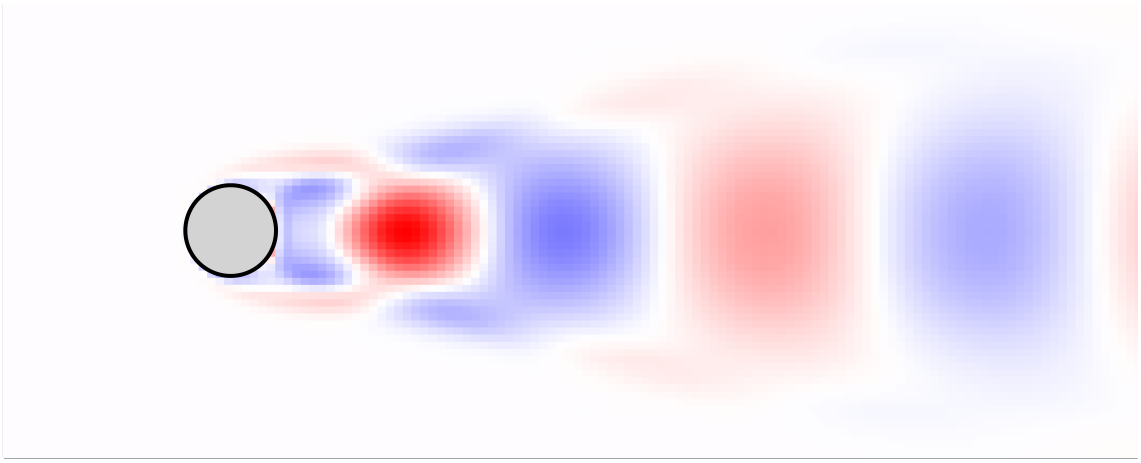
    U = (1/√(n-1))*X*V[:, 1:100]*Λsq

```

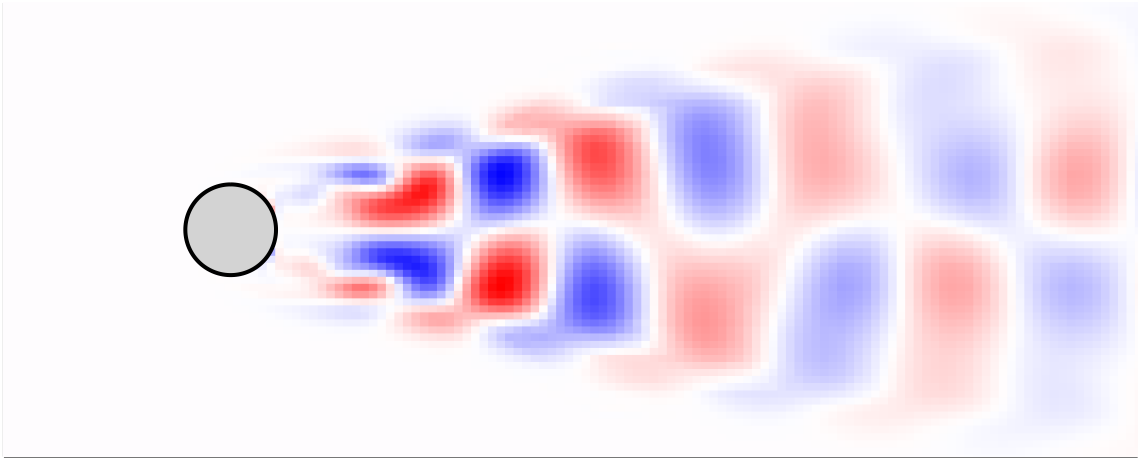


```
# --> Plot PCA 1.

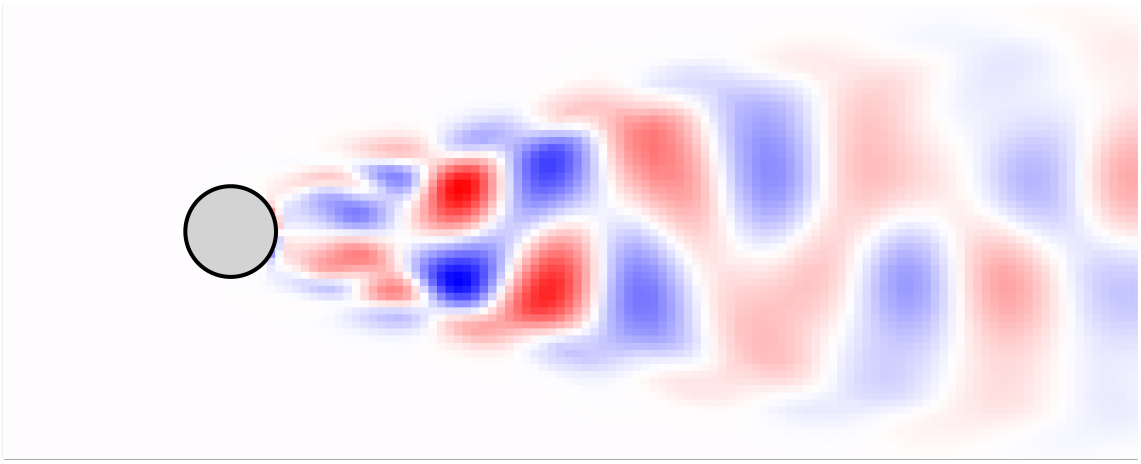
    plot_flow_field(mesh, U[:, 1])
```



```
# --> Plot PCA 2.
    plot_flow_field(mesh, U[:, 2])
```



```
# --> Plot PCA 3.
    plot_flow_field(mesh, U[:, 3])
```



```
# --> Plot PCA 4.
    plot_flow_field(mesh, U[:, 4])
```

Pattern :

We can notice that for the first PCA mode, the flow is regular along the bluff body which can be approximated to ideal conditions flow whereas in the second one, we observe a small perturbation. In the third PCA mode, smaller vortexes start appearing thus creating a cascade effect which

intensifies in the fourth PCA mode. This phenomenon keeps on going until the injection scale (scale of the vortex of the first PCA mode) becomes a small scale which is the last step before the destruction of the vortex by dissipation.

Sparse sensor placement

Measuring the velocity in the whole domain is technically impossible outside of a simulation. In any experiment, information is gathered from limited sensor measurements. Having gained some insight about the coherent structures existing in the flow, let us leverage these to guide us in placing actual sensors :

1. Choose a desired rank `r` for the PCA truncation (i.e. `Ψ = U[:, 1:r]`)
2. Compute the QR decomposition with pivot of `transpose(Ψ)` using `qr(transpose(Ψ), Val(true))` and return the `r` first pivots `p`.
3. Use the command `plot_flow_field(mesh, U[:, 1], p)` to superimpose the leading PCA mode and the location of the selected sensors.
4. Using your physical intuition, do these sensor locations make sense ? If so, why ? If not, why ?

Rank `r` of the PCA basis : 24

```
md"Rank `r` of the PCA basis : $@bind r Slider(1:50, default=2, show_value=true)"
```

```
Ψ = 20769×24 Array{Float64,2}:
     -3.46018e-6   1.37692e-6  -4.09182e-8  …   1.11064e-8   6.76295e-9  -3.38475e-9
     -3.47626e-6   1.3893e-6   -4.03592e-8      1.18533e-8   2.99711e-9  -4.21211e-9
     -3.49132e-6   1.40255e-6  -3.97073e-8      1.34765e-8   3.8207e-9   -6.59342e-9
     -3.50761e-6   1.4138e-6   -3.9224e-8       1.5549e-8   -2.26235e-9  -1.04982e-8
     -3.52456e-6   1.424e-6    -3.88307e-8      1.77692e-8  -1.219e-8    -1.52376e-8
     -3.53885e-6   1.43719e-6  -3.81906e-8  …   1.42452e-8  -9.61692e-9  -1.0672e-8
     -3.55466e-6   1.44816e-6  -3.77237e-8      1.24039e-8   2.41385e-9  -5.91534e-9
      ⋮                                    ⋱
      1.56522e-6  -3.0167e-6   -1.78447e-7     -3.80197e-8   1.48343e-8   1.59219e-8
      1.55773e-6  -3.00776e-6  -1.58547e-7     -3.90676e-8   4.94569e-9   1.53482e-8
      1.54782e-6  -2.99865e-6  -1.4002e-7   …  -3.53299e-8  -4.65763e-9   6.58837e-9
      1.53776e-6  -2.98957e-6  -1.21434e-7     -3.14648e-8  -1.3991e-8   -2.36343e-9
      1.52996e-6  -2.97817e-6  -1.09257e-7     -2.95326e-8  -9.19567e-9  -3.51542e-9
      1.52208e-6  -2.96498e-6  -9.69583e-8     -2.8444e-8    3.36754e-9  -2.44308e-9
```

```
# --> Truncated PCA basis Ψ (\Psi <TAB> to have Ψ displayed as variable name).

    Ψ = U[:, 1:r]
```

```
QRPivoted{Float64,Array{Float64,2}}
Q factor:
24×24 LinearAlgebra.QRPackedQ{Float64,Array{Float64,2}}:
 -0.157974    -0.123757    0.145301    …   0.104849    0.0969735  -0.251002
 -0.161736    -0.188651    0.120373       -0.271184    0.0984122   0.156935
 -0.00631298   0.0499783  -0.0242032      -0.480721   -0.632898   -0.166846
 -0.0940938    0.0160937   0.00505174      0.243151   -0.256349    0.696403
  0.0021027    0.0688357   0.0125476      -0.19642    -0.0555457   0.123822
  0.140408     0.127424   -0.144104    …  -0.246308    0.0919763   0.48806
  0.0961624   -0.137052    0.0332101      -0.44915     0.108341   -0.0786003
  ⋮                                   ⋱
  0.218741     0.244726    0.315703        0.104066    0.121683    0.0568702
  0.374957    -0.355204    0.196275       -0.0476809   0.0220707   0.0737097
 -0.25967      0.0905587   0.374416    …   0.0259956  -0.0715297   0.0811652
```

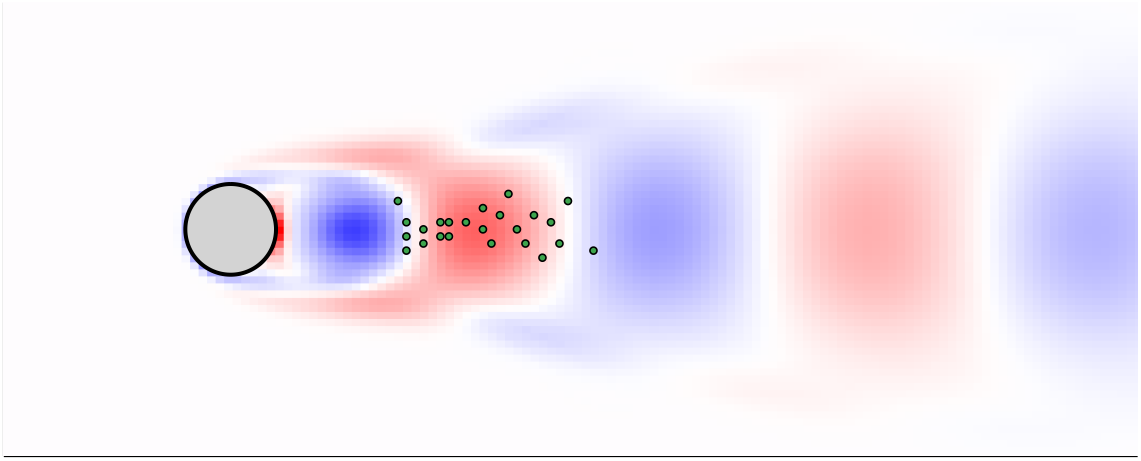
```
  0.107759     0.232051   -0.0328034        0.118066    0.0151423  -0.0369397
  0.438691    -0.323908    0.266604         0.0857239  -0.119536    0.0281983
  0.250194     0.442198    0.436909        -0.0192494  -0.0659429  -0.000390867
R factor:
24×20769 Array{Float64,2}:
 -0.254973  -0.0169709   0.0361198  …   2.38874e-7   2.39105e-7   2.45732e-7
  0.0       -0.252992   -0.10222        3.33005e-7   3.32243e-7   3.29975e-7
  0.0        0.0         0.230678      -1.37515e-7  -1.38066e-7  -1.39942e-7
  0.0        0.0         0.0            3.25525e-7   3.21003e-7   3.16691e-7
  0.0        0.0         0.0           -2.21443e-7  -2.16985e-7  -2.09209e-7
  0.0        0.0         0.0        …  -4.97326e-7  -4.94971e-7  -4.88673e-7
  0.0        0.0         0.0           -5.44073e-7  -5.41694e-7  -5.44303e-7
```

```
# --> Compute the QR decomposition with pivot.

    Q, R, pp = qr(transpose(Ψ), Val(true))
```

```
p =
  Int64[10257, 10126, 10255, 10772, 10899, 10128, 11032, 11415, 10775, 11548, 9612
```

```
    p = pp[1:r]
```



```
# --> Plot the first PCA mode superimposed with the sensor locations.

    plot_flow_field(mesh, U[:, 1], p)
```

Question 4

At first, the location of the sensors seem pretty random, but then by increasing the number of sensors, it seems that they are located in a way that they capture information at each position in the vortex scale. After that we can assume that the information is similar thanks to the periodic pattern observed.

The gif, joined to this document, shows the change in the position of the sensors with the increase of r.

Denoting by `a` the projection `tranpose(Ψ) * X` (i.e. the projection of the data into the leading PCA subspace), `y = X[p, :]` the measurements and `Θ = Ψ[p, :]` the measurement matrix, use the least-squares technique to solve

$$\mathbf{y} = \mathbf{\Theta \hat{a}}$$

and compare the the time-series of the estimated `â` coefficients with the true ones in `a` for varying values of `r`. What do you observe ?

```
24×400 Array{Float64,2}:
 -17.8876      -12.2365      -6.41565      …  -44.6201      -44.9831     -44.4366
  38.5281       40.4957      41.7586           -4.10189       1.72867      7.48945
  13.0925       13.107       12.1319           -8.96529      -6.58142     -3.87504
  -0.641533      1.7782       4.29254          -9.70365     -10.7446     -11.0104
  14.1443       14.1965      11.9624          -10.5853      -13.8357     -14.8696
  -4.11077       2.15048      7.9414       …    8.48555       3.21493     -2.78598
  -2.58558      -0.0794166    2.35091           2.96857       4.68157      4.95491
   ⋮                                        ⋱
   0.164378      0.017473    -0.156926          0.160095      0.0535061   -0.144136
  -0.00454855   -0.158512    -0.0449777         0.0270315    -0.144225    -0.0755555
   0.0891021     0.0152      -0.0888382    …   -0.00965241    0.086894     0.0127106
  -0.0164791     0.0868941    0.0189151         -0.0825977    -0.0134747    0.0822219
  -0.0472023    -0.00406354   0.0486981          0.0464248    -0.0109896   -0.0436212
   0.0106142    -0.0451617    0.000849374        0.005554      0.0432665   -0.0143476
```

```
begin
    # --> Takes the measurements y.
    y = X[p, :]

    # --> Build the measurement matrix Θ (\Theta <TAB>)
    Θ = Ψ[p, :]

    # --> Obtain the true low-dimensional projection (a = transpose(Ψ) * X).
    #a = transpose(Ψ) * X
    a = Ψ' * X

end
```

```
â = 24×400 Array{Float64,2}:
    -17.8522      -12.2459      -6.44443    …  -44.6197      -45.0276     -44.4066
     38.5274       40.4904      41.7608          -4.12008       1.73757      7.50114
     13.1581       13.0707      12.0931          -8.96848      -6.64152     -3.82895
     -0.720381      1.79847      4.36148         -9.68552     -10.704      -11.0518
     14.138        14.1714      11.9847         -10.6053      -13.8149     -14.8608
     -4.10281       2.14579      7.93943    …    8.48793       3.20858     -2.78809
     -2.61419      -0.035135     2.35171          3.01111       4.69436      4.90223
      ⋮                                       ⋱
      0.154124      0.0277573   -0.154779         0.156439      0.0609387   -0.146521
     -0.00861525   -0.153375    -0.0437318        0.0372302    -0.140843    -0.0870401
      0.112516     -0.014857    -0.0936171   …   -0.038567      0.0870157    0.0420435
     -0.0033923     0.0710498    0.0155753        -0.0858874    -0.0222204    0.0948427
     -0.040326     -0.0176568    0.0508223         0.0315053    -0.0106523   -0.0287866
     -0.00687243   -0.0400348    0.0141349        -0.00348822    0.0633474   -0.0200665
```

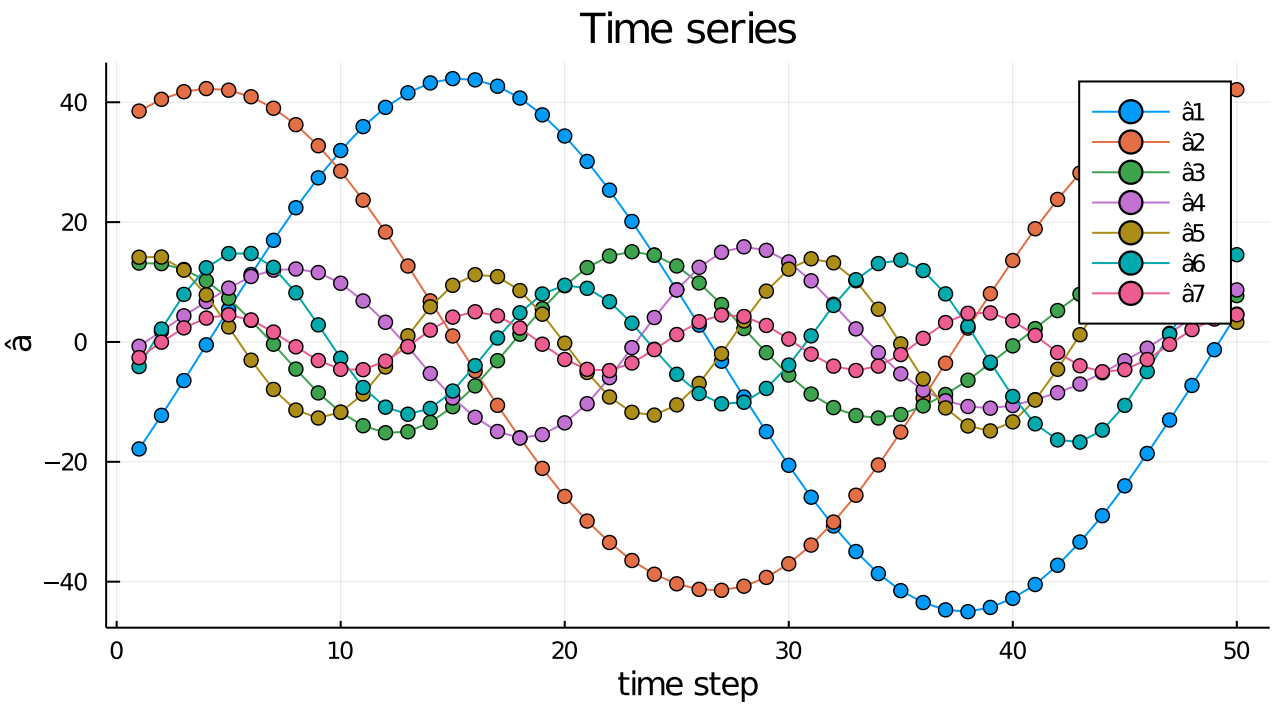
```
# --> Solve Θ * â = y.
    â = qr(Θ) \ y

```

```
24×400 Array{Float64,2}:
  0.903663   1.00307    1.1711      1.39589    …  -2.56497    -2.3881     -2.05145
  0.838023   1.04509    1.30572     1.60257       -1.9347     -1.32358    -0.54324
  0.544596   0.670826   0.859507    1.10734       -2.33896    -1.94606    -1.34615
  0.238315   0.362797   0.314455    0.268774      -2.41013    -2.5265     -2.58658
  0.060952   0.155938   0.0880709   0.0169629     -2.14467    -2.3101     -2.401
```

```
 1.19031    1.36757    1.59765     1.86013    …  -2.36714    -2.02845    -1.48408
-1.5949    -0.898531  -0.210489    0.227834       -1.47774    -1.80132    -2.0582
 ⋮                                              ⋱
-1.92522   -1.80361   -1.53255    -1.10025        -0.433085   -0.544671   -0.703157
 2.00748    2.12751    2.17212     2.15764         0.656699    0.608573    0.619792
 1.48374    1.76482    2.02507     2.23271    …    0.0134867   0.510637    0.680439
 2.77392    2.79088    2.75068     2.63958        -0.59638    -0.0704846   0.544301
-2.27936   -2.09136   -1.66656    -1.02531        -0.382741   -0.678168   -1.02652
-0.995151  -1.36729   -1.7021     -1.96929         0.123401    0.12189     0.113724
```

```
y
```




```
begin
    # --> Plot the time series.

    plot(â[1,1:50], marker=".",
    xlabel="time step",
    ylabel="â",
    label="â1",
    leg=:true,
    title="Time series",
    size=(650, 350)
    )
    plot!(â[2,1:50], marker=".", label="â2")
    plot!(â[3,1:50], marker=".", label="â3")
    plot!(â[4,1:50], marker=".", label="â4")
    plot!(â[5,1:50], marker=".", label="â5")
    plot!(â[6,1:50], marker=".", label="â6")
    plot!(â[7,1:50], marker=".", label="â7")

end
```

What do you observe as `r` increases ?`

From the time series plot, we can see that the amplitude decreases when r increases which makes sense since it is the first PCA modes that let us capture the largest information and with the increase of r, it fades away.

Inferring the vorticity field from sparse measurements

Now that we know where to place our sensors and can reconstruct the low-dimensional state vector $\mathbf{a}$, let us recontruct the whole vorticity field. Denoting by $\hat{\mathbf{a}}$ the estimated low-dimensional state vector, the whole flow field can be reconstructed as

$$\hat{\omega}(\mathbf{x}, t_k) = \mathbf{\Psi}\hat{\mathbf{a}}.$$

The reconstruction error is defined as

$$\text{Err} = \frac{1}{N}\sum_{i=1}^{N} \|\hat{\omega}_i - \omega_i\|_2^2$$

Write a function that computes the reconstruction error as a function of the number r of sensors place in the flow.

```
ω̂ = 20769×400 Array{Float64,2}:
      0.000116361   9.94415e-5    8.06147e-5   …   0.000156439   0.000162225
      0.00011715    0.000100157   8.12408e-5       0.00015718    0.000163032
      0.000117952   0.000100896   8.19003e-5       0.000157875   0.000163799
      0.000118697   0.000101565   8.24793e-5       0.000158625   0.000164609
      0.000119413   0.000102198   8.30157e-5       0.000159405   0.000165442
      0.000120192   0.000102916   8.36633e-5   …   0.000160066   0.000166176
      0.00012091    0.000103562   8.42288e-5       0.000160795   0.000166963
      ⋮                                        ⋱
     -0.000148977  -0.000146242  -0.000140567     -7.06089e-5   -8.75735e-5
     -0.000148205  -0.000145431  -0.000139752     -7.06553e-5   -8.75492e-5
     -0.000147392  -0.000144631  -0.000138999  …  -7.04967e-5   -8.73046e-5
     -0.000146576  -0.000143832  -0.00013825      -7.03264e-5   -8.70471e-5
     -0.000145808  -0.000143028  -0.000137441     -7.02642e-5   -8.6915e-5
     -0.000144973  -0.000142184  -0.000136612     -7.0098e-5    -8.66534e-5
```

```
ω̂ = Ψ * â
```

```
ω = 20769×400 Array{Float64,2}:
      0.000116488   9.94195e-5    8.05059e-5   …   0.000156268   0.000162313
      0.000117278   0.000100135   8.11317e-5       0.000157008   0.00016312
      0.00011808    0.000100874   8.17907e-5       0.000157702   0.000163888
      0.000118826   0.000101544   8.23693e-5       0.000158451   0.000164698
      0.000119542   0.000102176   8.29055e-5       0.00015923    0.000165531
      0.000120322   0.000102894   8.35526e-5   …   0.000159891   0.000166265
      0.00012104    0.00010354    8.41177e-5       0.000160619   0.000167052
      ⋮                                        ⋱
     -0.000149033  -0.000146253  -0.000140508     -7.05152e-5   -8.75785e-5
     -0.000148261  -0.000145442  -0.000139695     -7.05617e-5   -8.75542e-5
     -0.000147448  -0.000144642  -0.000138942  …  -7.04029e-5   -8.73099e-5
     -0.000146632  -0.000143842  -0.000138193     -7.02325e-5   -8.70527e-5
     -0.000145864  -0.000143038  -0.000137386     -7.01706e-5   -8.69204e-5
     -0.000145029  -0.000142193  -0.000136557     -7.00045e-5   -8.66593e-5
```

```
ω = Ψ * a
```

```
0.007757187003592994
```

```
begin
    som = 0
    for i=1:n
        norm = 0
```

```
        for j=1:m
            norm += (ω̂[j,i] - ω[j,i])^2
        end
        som += norm
    end
    Err = som / n
end
```

Based on your results, how would choose r such that it best balances the reconstruction accuracy and the number of required sensors ? Justify.

t **Answer :**

The choice of r will strongly depend on the error percentage we want to get. In our case, we were able to see that 7 sensors were enough to capture 99% of the data but we still get a huge error (~2100%) which is probably due to the turbulence and the sensors locations that seem quite random.

A good trade off would be r=24 which gives less than 1% error.

```
md"t
**Answer :**

The choice of r will strongly depend on the error percentage we want to get. In our case, we were able to see that 7 sensors were enough to capture 99% of the data but we still get a huge error (~2100%) which is probably due to the turbulence and the sensors locations that seem quite random.

A good trade off would be r=24 which gives less than 1% error.

"
```

The methodology presented here is quite general and can be applicable to large variety of different engineering fields. Suggest three applications of this methodology to other problems and explain why you think if would be a good way to tackle the problem.

Answer :

- Medical use : for example showing the correlation of Cholesterol and low density lipo-protein.
- Finance use : financial time series, dynamic trading strategies, financial risk computations and even estimation of future stock price values.
- Face recognition.

```
md"
**Answer :**

- Medical use : for example showing the correlation of Cholesterol and low density lipo-protein.

-  Finance use : financial time series, dynamic trading strategies, financial risk computations and even estimation of future stock price values.

- Face recognition.

"
```

```
plot_flow_field (generic function with 1 method)
```

```
function plot_flow_field(mesh, x)

    scale = maximum(abs.(x))

    p = heatmap(
        mesh[1], mesh[2], reshape(x, length(mesh[2]), length(mesh[1])),
        axis=[],
        ylims=(-2.5, 2.5),
        xlims=(-2.5, 10),
        aspect_ratio=1,
        legend=false,
        c=cgrad(:bwr),
        clims=(-scale, scale),
    )

    plot!(circle(0, 0, 0.5), seriestype=[:shape], lw=2, c=:lightgray)

    return p
end
```

```
circle (generic function with 1 method)
```

```
function circle(xc, yc, r)
    θ = LinRange(0, 2π, 128)
    return xc .+ r*sin.(θ), yc .+ r*cos.(θ)
end
```

```
plot_flow_field (generic function with 2 methods)
```

```
function plot_flow_field(mesh, x, pxl)

    # --> Plot the flow field.
    p = plot_flow_field(mesh, x)

    # --> Get the cartesian indices.
    idx = CartesianIndices((1:length(mesh[2]), 1:length(mesh[1])))

    ix = [mesh[1][idx[i][2]] for i in pxl]
    iy = [mesh[2][idx[i][1]] for i in pxl]

    scatter!(ix, iy, ms=2)

    return p
end
```