```julia
1  begin
2      using LinearAlgebra
3      using PlutoUI
4      using PlutoTeachingTools
5      using Plots
6      using LaTeXStrings
7      using HypertextLiteral: @htl, @htl_str
8      using Printf
9  end
```

☰ Table of Contents

⚠ TODO ⚠

If we do this chapter *after* boundary value problems, we can actually motivate eigenvalue problems from solving an equation like $-\Delta u = f$ in a bounded domain (discretised e.g. using sine functions). This is nice because (a) it is related to the Resonance phaenomena equations, that lead to the resonance catastrophe in bridges (if f hits an eigenvector of the laplacian) and (b) it makes the whole eigenvalue problems better embedded into the rest of the course and as an "application".

```
1  TODO(md"""If we do this chapter *after* boundary value problems, we can actually
   motivate eigenvalue problems from solving an equation like $-\Delta u = f$ in a
   bounded domain (discretised e.g. using sine functions). This is nice because (a) it
   is related to the Resonance phaenomena equations, that lead to the resonance
   catastrophe in bridges (if $f$ hits an eigenvector of the laplacian) and (b) it
   makes the whole eigenvalue problems better embedded into the rest of the course and
   as an "application". """)
```

Eigenvalue problems 🔗

Recall that the eigenpairs of a matrix $\mathbf{A}$ are the pairs $(\lambda_i, \mathbf{v}_i)$ of eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ such that

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i.$$

Geometrically speaking the eigenvectors provide special directions in space along which forming the matrix-vector-product $\mathbf{A}\mathbf{v}_i$ is particularly simple, namely it just *scales* the vector $\mathbf{v}_i$ by a number.

But more generally if $\mathbf{x}$ is an arbitrary vector and if for simplicity we assume $\mathbf{A}$ to be symmetric, then we find

$$\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\| \, \|\mathbf{x}\| = \sqrt{\lambda_{\max}^{\mathrm{abs}}(\mathbf{A}^T\mathbf{A})} \, \|\mathbf{x}\| = \lambda_{\max}^{\mathrm{abs}}(\mathbf{A}) \, \|\mathbf{x}\|$$

where we used the inequalities introduced at the end of <u>Direct methods for linear systems</u>. We note that the **largest eigenvalue of $\mathbf{A}$** provides a **bound to the action of $\mathbf{A}$**.

This may sound technical, but as **matrices are common in physics and engineering** and since their **eigenpairs characterise the action of these matrices**, the computation of eigenpairs often carries a **physical interpretation**.

For example, in the classical mechanics of rotating objects, the eigenvectors of the **<u>Moment of inertia</u>** tensor are the **principle axes** along which an object spins without coupling to other rotational degrees of freedom.

In engineering the **eigenvalues of the Hessian matrix (the Laplacian)** of an object's total energy describe the **resonance frequences**. That is the frequencies at which the object best absorbs energy from an outside excitation. When these frequencies coincide with the motion of humans or cars this can lead to a <u>Resonance disaster</u>, e.g. the famous <u>Tacoma Narrows Bridge Collapse</u> or the <u>Millenium Bridge Lateral Excitation Problem</u>). Analysing the eigenfrequencies of bridges is nowadays required as part of the procedure to obtain the permissons for construction.

In this notebook we will discuss some simple iterative methods for actually computing eigenpairs. However, the topic is vast and we will only scratch the surface. Readers interested in a more in-depth treatment of eigenvalue problems are encouraged to attend the master class <u>MATH-500: Error control in scientific modelling</u>. Some recommended further reading can also be found in the

book [Numerical Methods for Large Eigenvalue Problems](#) by Youssef Saad as well as the [Lecture notes on Large Scale Eigenvalue Problems](#) by Peter Arbenz.

Power iteration 🔗

We start with a simple question: What happens if we apply a matrix multiple times ? Here, we choose the matrix

```
A = 2×2 Matrix{Float64}:
    0.5  0.333333
    0.5  0.666667
1 A = [ 0.5  1/3;
2        0.5  2/3]
```

and a random 2-element vector:

```
▶[-1.64403, 0.314048]
1 begin
2     dummy   # For rerun to work
3     x = randn(2)
4 end
```

```
▶[-1.64403, 0.314048]
1 x
```

Applying the matrix once seems to be very innocent:

```
▶[-0.71733, -0.612647]
1 A * x
```

But if we appy many times we start to see something ...
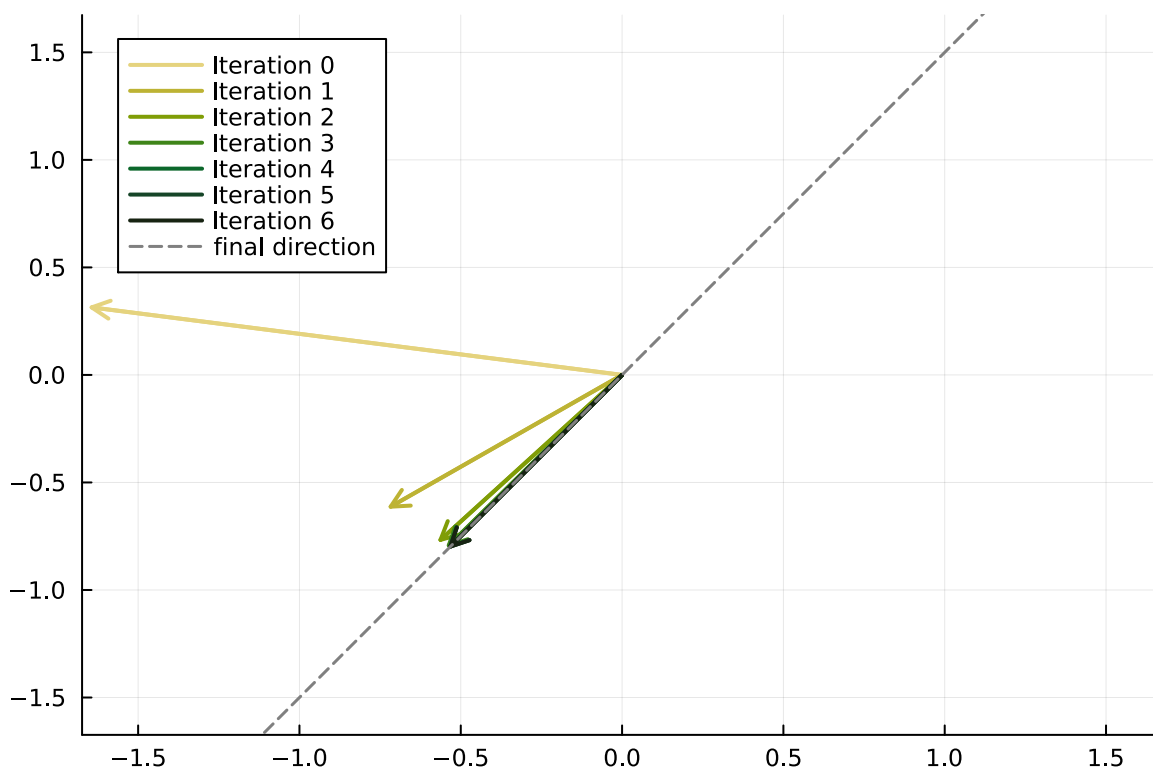
```
1  begin
2      history = [x]
3      for j in 1:6
4          x = A * x
5          push!(history, x)
6          @printf "Iteration %i:  x = [%12.4f, %12.4f]\n" j x[1] x[2]
7      end
8      maxerror = maximum(abs, (x - A * x))
9      @show maxerror
10  end;
```

```
Iteration 1:  x = [      -0.7173,       -0.6126]
Iteration 2:  x = [      -0.5629,       -0.7671]
Iteration 3:  x = [      -0.5371,       -0.7928]
Iteration 4:  x = [      -0.5328,       -0.7971]
Iteration 5:  x = [      -0.5321,       -0.7978]
Iteration 6:  x = [      -0.5320,       -0.7980]
maxerror = 1.986229314476784e-5
```


Regenerate random vector and rerun experiment

Note how the iterations stabilise, i.e. that $\mathbf{x}$ and $\mathbf{Ax}$ start to be alike. In other words we seem to achieve $\mathbf{Ax} = \mathbf{x}$, which is nothing else than saying that $\mathbf{x}$ is an eigenvector of $\mathbf{A}$ with eigenvalue 1 .

Let us understand what happened in this example in detail. We consider the case $\mathbf{A} \in \mathbb{R}^{n \times n}$ diagonalisable and let further

$$|\lambda_1| \leq |\lambda_2| \leq |\lambda_3| \leq \cdots \leq |\lambda_{n-1}| < |\lambda_n| \tag{1}$$

be its eigenvalues with corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, which we collect column-wise in a unitary matrix $\mathbf{V}$, i.e.

$$\mathbf{V} = \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \\ \downarrow & \downarrow & \cdots & \downarrow \end{pmatrix}$$

Note that $|\lambda_{n-1}| < |\lambda_n|$, such that λ_n is non-degenerate and the absolutely largest eigenvalue. We call it the **dominant eigenvalue** of $\mathbf{A}$. If $k > 0$ is a positive integer, then we have

$$\mathbf{A}^k = \left(\mathbf{V} \mathbf{D} \mathbf{V}^{-1}\right)^k = \mathbf{V} \mathbf{D}^k \mathbf{V}^{-1}$$

and we can expand any random starting vector $\mathbf{x}^{(1)}$ in terms of the eigenbasis of $\mathbf{A}$, i.e.

$$\mathbf{x}^{(1)} = \sum_{i=1}^{n} z_i \mathbf{v}_i = \mathbf{V} \mathbf{z},$$

where $\mathbf{z}$ is the vector collecting the coefficients z_i. Notably this implies $\mathbf{z} = \mathbf{V}^{-1} \mathbf{x}^{(1)}$.

Consider applying $\mathbf{A}$ a k number of times to $\mathbf{x}^{(1)}$. This yields

$$\mathbf{A}^k \mathbf{x}^{(1)} = \mathbf{V} \mathbf{D}^k \underbrace{\mathbf{V}^{-1} \mathbf{x}^{(1)}}_{=\mathbf{z}} = \mathbf{V} \begin{pmatrix} \lambda_1^k z_1 \\ \lambda_2^k z_2 \\ \vdots \\ \lambda_n^k z_n \end{pmatrix} \tag{2}$$

$$= \lambda_n^k \left(\left(\frac{\lambda_1}{\lambda_n}\right)^k z_1 \mathbf{v}_1 + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n}\right)^k z_{n-1} \mathbf{v}_{n-1} + z_n \mathbf{v}_n \right).$$

If $z_n \neq 0$ then

$$\left\| \frac{\mathbf{A}^k \mathbf{x}^{(1)}}{\lambda_n^k} - z_n \mathbf{v}_n \right\| \leq \left|\frac{\lambda_1}{\lambda_n}\right|^k |z_1| \|\mathbf{v}_1\| + \cdots + \left|\frac{\lambda_{n-1}}{\lambda_n}\right|^k |z_{n-1}| \|\mathbf{v}_{n-1}\| \tag{3}$$

A consequence of the ascending eigenvalue ordering of equation (1) is that

$$\left|\frac{\lambda_j}{\lambda_n}\right| < 1 \qquad \text{for all } j = 1, \ldots, n-1$$

and therefore that each of the terms $\left| \frac{\lambda_j}{\lambda_n} \right|^k$ for $j = 1, \ldots, n-1$ goes to zero for $k \to \infty$. Therefore overall

$$\left\| \frac{\mathbf{A}^k \mathbf{x}^{(1)}}{\lambda_n^k} - z_n \mathbf{v}_n \right\| \to 0 \qquad \text{as } k \to \infty.$$

In other words $\frac{1}{\lambda_n^k} \mathbf{A}^k \mathbf{x}^{(1)}$ **eventually becomes a multiple of the eigenvector** associated with the dominant eigenvalue.

> ### Conventions of eigenpair ordering
>
> In the literature as well as across linear algebra codes there are multiple conventions regarding the ordering of eigenvalues. E.g. whether eigenvalues are ordered from smallest to largest, from largest to smallest, whether one considers the absolute value of the eigenvalues or the signed values etc. Wherever this is possible and makes sense we will employ the ordering
>
> $$|\lambda_1| \leq |\lambda_2| \leq |\lambda_3| \leq \cdots \leq |\lambda_n|$$
>
> i.e. that eigenvalues increase in magnitude, but their signs may differ.

Power iteration algorithm 🔗

Let's try applying our idea from the earlier section to the matrix

```
B = 2×2 Matrix{Float64}:
    0.1  5.0
    0.0  5.0
```

```
1  B = [0.1 5.0;
2        0.0 λn]
```

which has eigenvalues 0.1 and λ_n = 5.0. The latter is the dominant one, which can further be changed by this slider:

- λ_n = ▬▬▬▬●▭ 5.0

We again run 6 subsequent applications of $\mathbf{B}$ and hope the iterations to stabilise:

```
1  let
2      x = randn(2)
3      for j in 1:6
4          x = B * x
5          @printf "Iteration %i:  x = [%12.4f, %12.4f]\n" j x[1] x[2]
6      end
7  end;
```

```
Iteration 1:  x = [      -0.9014,       -1.0783]
Iteration 2:  x = [      -5.4817,       -5.3915]
Iteration 3:  x = [     -27.5058,      -26.9577]
Iteration 4:  x = [    -137.5390,     -134.7884]
Iteration 5:  x = [    -687.6958,     -673.9419]
Iteration 6:  x = [   -3438.4792,    -3369.7096]
```

But this time this does not work ... unless λ_n happens to be 1.0.

- This can be understood looking at equation (2): if $\lambda_n > 1.0$, then λ_n^k becomes extremely large, such that $\mathbf{A}^k \mathbf{x}^{(1)} \approx \lambda_n^k z_n \mathbf{v}^k$, which grows significantly from one iteration to the next, such that no stabilisation is achieved. Similarly if $\lambda_n < 1.0$ then as $k \to \infty$ the λ_n^k becomes smaller and smaller.
- Apart from not converging, this also poses difficulties from a numerical point of view, since accurately representing the vector entries of $\mathbf{A}^k \mathbf{x}$ and performing the associated matrix-vector products becomes more and more difficult the larger the range of numbers that have to be represented.

Naively one could take a look at (3) and just **normalise in each iteration** by dividing by λ_n. And indeed this works:

```
1  let
2      x = randn(2)
3      for j in 1:6
4          x = x / λn    # Normalise
5          x = B * x
6          @printf "Iteration %i:  x = [%12.4f, %12.4f]\n" j x[1] x[2]
7      end
8  end;
```

```
Iteration 1:  x = [       0.8641,        0.8765]
Iteration 2:  x = [       0.8938,        0.8765]
Iteration 3:  x = [       0.8944,        0.8765]
Iteration 4:  x = [       0.8944,        0.8765]
Iteration 5:  x = [       0.8944,        0.8765]
Iteration 6:  x = [       0.8944,        0.8765]
```

The problem here is that for general problems **we don't know λ_n**, so we cannot use it for normalisation. Fortunately it turns out, however, that pretty much any normalisation of **x** works. For example we can use the infinity norm:

$$\|x\|_\infty = \max_{i=1,\ldots n} |x_i|,$$

which simply selects the largest absolute entry of the vector. Again this works as expected:

```
1  let
2      x = randn(2)
3      for j in 1:6
4          x = x / maximum(abs.(x))   # Normalise
5          x = B * x
6          @printf "Iteration %i:  x = [%12.4f, %12.4f]\n" j x[1] x[2]
7      end
8      @printf "Estimate for eigenvalue: %12.6f" maximum(abs.(x))
9  end;
```

```
Iteration 1:   x = [       0.8856,       0.7856]
Iteration 2:   x = [       4.5354,       4.4354]
Iteration 3:   x = [       4.9898,       4.8898]
Iteration 4:   x = [       4.9998,       4.8998]
Iteration 5:   x = [       5.0000,       4.9000]
Iteration 6:   x = [       5.0000,       4.9000]
Estimate for eigenvalue:      5.000000
```

We also note in passing that $\|x\|_\infty$ seems to converge to the dominant eigenvalue.

Note that $\|x\|_\infty = \max_{i=1,\ldots n} |x_i|$ implies that there exists an index $m \in \{1,\ldots n\}$, such that $\|x\|_\infty = |x_m|$.

Keeping this in mind we formulate the algorithm

> ## Algorithm 1: Power iterations
>
> Given a diagonalisable matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and an initial guess $\mathbf{x}^{(1)} \in \mathbb{R}^n$ we iterate for $k = 1, 2, \ldots$:
>
> 1. Set $\mathbf{y}^{(k)} = \mathbf{A}\mathbf{x}^{(k)}$
> 2. Find the index m such that $\left|y_m^{(k)}\right| = \left\|y^{(k)}\right\|_\infty$
> 3. Compute $\alpha^{(k)} = \frac{1}{y_m^{(k)}}$ and set $\beta^{(k)} = \frac{y_m^{(k)}}{x_m^{(k)}}$ *(see below why)*
> 4. Set $\mathbf{x}^{(k+1)} = \alpha^{(k)}\mathbf{y}^{(k)}$ *(Normalisation)*

> We obtain $\beta^{(k)}$ as the estimate to the dominant eigenvalue and $\mathbf{x}^{(k)}$ as the estimate of the corresponding eigenvector.

In this algorithm $\alpha^{(k)} = \frac{1}{y_m^{(k)}} = \frac{1}{\|y^{(k)}\|_\infty}$. Step 4 is thus performing the normalisation we developed above.

Furthermore $\beta^{(k)}$ is now computed as the eigenvalue estimate instead of $\|x\|_\infty$. The idea is that if $\mathbf{x}^{(k)}$ is already close to the eigenvector $\mathbf{v}_n$ associated to the dominant eigenvalue λ_n, then

$$\mathbf{y}^{(k)} = \mathbf{A}\,\mathbf{x}^{(k)} \approx \mathbf{A}\,\mathbf{v}_n = \lambda_n \mathbf{x}^{(k)} \quad \Rightarrow \quad y_m^{(k)} \approx \lambda_n \mathbf{x}_m^{(k)} \quad \Rightarrow \quad \lambda_n \approx \frac{\mathbf{y}_m^{(k)}}{\mathbf{x}_m^{(k)}} = \beta^{(k)}$$

An implementation of this power method algorithm is:

```
power_method (generic function with 1 method)
 1  function power_method(A, x; maxiter=100)
 2      n = size(A, 1)
 3      x = normalize(x, Inf)  # Normalise initial guess
 4
 5      history = Float64[]  # Record a history of all βs (estimates of eigenvalue)
 6      for k in 1:maxiter
 7          y = A * x
 8          m = argmax(abs.(y))
 9          α = 1 / y[m]
10          β = y[m] / x[m]
11          push!(history, β)
12          x = α * y
13      end
14
15      (; x, λ=last(history), history)
16  end
```

Let's try this on a lower-triangular matrix as a test. Note that the eigenvalues of a lower-triangular matrix are exactly the diagonal entries. We set

```
5×5 Matrix{Float64}:
 1.0    1.0    1.0    1.0   1.0
 0.0   -0.75   1.0    1.0   1.0
 0.0    0.0    0.6    1.0   1.0
 0.0    0.0    0.0   -0.4   1.0
 0.0    0.0    0.0    0.0   0.0
```

```
1  begin
2      λref = [1, -0.75, 0.6, -0.4, 0]
3      # Make a triangular matrix with eigenvalues on the diagonal.
4      M = triu(ones(5,5),1) + diagm(λref)
5  end
```

By construction the largest eigenvalue of this matrix is 1. So let's track convergence:

Convergence of power iteration
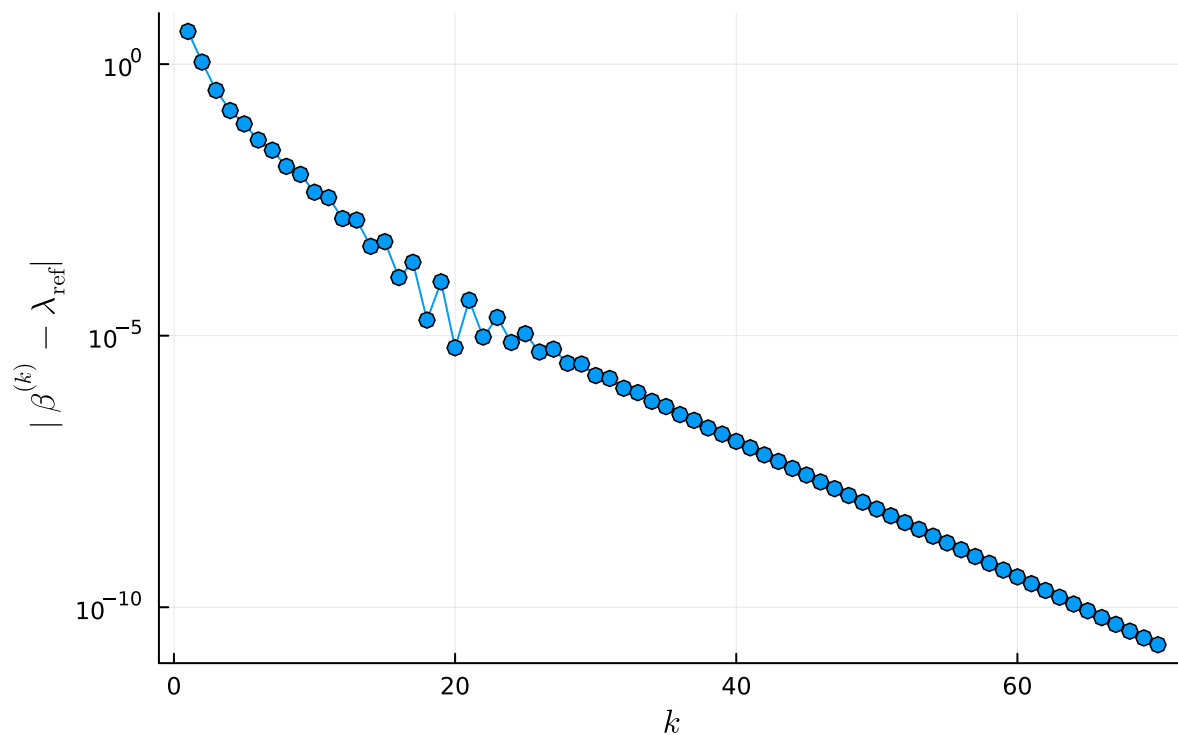
```
 1  begin
 2      λlargest = 1.0   # Largest eigenvalue of M (by construction)
 3
 4      x1 = ones(size(M, 2))
 5      results = power_method(M, x1; maxiter=70)
 6      error = abs.(results.history .- λlargest)
 7      plot(error; mark=:o, label="", yaxis=:log,
 8          title="Convergence of power iteration",
 9          ylabel=L"|β^{(k)} - λ_{\textrm{ref}}|", xlabel=L"k")
10  end
```

We want to demonstrate why $\beta^{(k)} \to \lambda_n$ as $k \to \infty$ more rigorously.

Let us first note that for our algorithm

$$\mathbf{x}^{(k)} = \alpha^{(1)} \cdot \alpha^{(2)} \cdot \cdots \cdot \alpha^{(k)} \cdot \mathbf{A}^k \mathbf{x}^{(1)} = \prod_{i=1}^{k} \alpha^{(i)} \cdot \mathbf{A}^k \mathbf{x}^{(1)} \qquad (4)$$

and by construction we always have $\left\| \mathbf{x}^{(k)} \right\|_\infty = 1$. We further note

$$
\begin{aligned}
\beta^{(k)} &= \frac{y_m^{(k)}}{x_m^{(k)}} = \frac{\left(\mathbf{A}\,\mathbf{x}^{(k)} \right)_m}{x_m^{(k)}} \\
&\stackrel{(4)}{=} \frac{\left(\prod_{i=1}^{k} \alpha^{(i)} \cdot \mathbf{A}^{k+1} \mathbf{x}^{(1)} \right)_m}{\left(\prod_{i=1}^{k} \alpha^{(i)} \cdot \mathbf{A}^{k} \mathbf{x}^{(1)} \right)_m} \\
&\stackrel{(2)}{=} \frac{\lambda_n^{k+1} \left(\left(\frac{\lambda_1}{\lambda_n} \right)^{k+1} z_1 \mathbf{v}_1 + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n} \right)^{k+1} z_{n-1} \mathbf{v}_{n-1} + z_n \mathbf{v}_n \right)_m}{\lambda_n^{k} \left(\left(\frac{\lambda_1}{\lambda_n} \right)^{k} z_1 \mathbf{v}_1 + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n} \right)^{k} z_{n-1} \mathbf{v}_{n-1} + z_n \mathbf{v}_n \right)_m} \\
&= \frac{\lambda_n \left(\left(\frac{\lambda_1}{\lambda_n} \right)^{k+1} b_1 + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n} \right)^{k+1} b_{n-1} + b_n \right)}{\left(\left(\frac{\lambda_1}{\lambda_n} \right)^{k} b_1 + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n} \right)^{k} b_{n-1} + b_n \right)}
\end{aligned}
$$

where m denotes that we take the m-th element of the vector in the brackets and we further defined $b_i = z_i (\mathbf{v}_i)_m$, that is z_i multiplied by the m-th element of the eigenvector $\mathbf{v}_i$. Again we assume $z_n \neq 0$, which implies $b_n \neq 0$. Under this assumption

$$\beta^{(k)} = \frac{y_m^{(k)}}{x_m^{(k)}} = \lambda_n \frac{\left(\frac{\lambda_1}{\lambda_n} \right)^{k+1} \frac{b_1}{b_n} + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n} \right)^{k+1} \frac{b_{n-1}}{b_n} + 1}{\left(\frac{\lambda_1}{\lambda_n} \right)^{k} \frac{b_1}{b_n} + \cdots + \left(\frac{\lambda_{n-1}}{\lambda_n} \right)^{k} \frac{b_{n-1}}{b_n} + 1} \to \lambda_n \qquad \text{as } k \to \infty. \quad (5)$$

Keeping in mind that $\frac{\lambda_j}{\lambda_n} < 1$ for all $1 \leq i \leq n-1$ we indeed observe $\beta^{(k)}$ to converge to λ_n.

▶ **A remark on $z_n \neq 0$**

Convergence of power method 🔗

The above plot already suggests a **linear convergence** towards the exact eigenvalue. Indeed a detailed analysis shows that the convergence rate can be computed as

$$r_{\text{power}} = \lim_{k \to \infty} \frac{|\beta^{(k+1)} - \lambda_n|}{|\beta^{(k)} - \lambda_n|} = \left| \frac{\lambda_{n-1}}{\lambda_n} \right|. \tag{6}$$

▶ **Optional: Detailed derivation**

So the smaller the ratio between $|\lambda_{n-1}|$ and $|\lambda_n|$, the faster the convergence.

Let us take a case where $\lambda_n = 1.0$. Therefore the size of λ_{n-1} determines the rate of convergence. Let's take λ_{n-1} = -0.9.

▶ `[0.0, -0.4, 0.6, -0.9, 1.0]`

```
1  begin
2      λ_δ = [0.0, -0.4, 0.6, λn-1, 1.0]    # The reference eigenvalues we will use
3  end
```

and we will put these on the diagonal of a triangular matrix, such that the eigenvalues of the resulting matrix M are exactly the λ_δ:
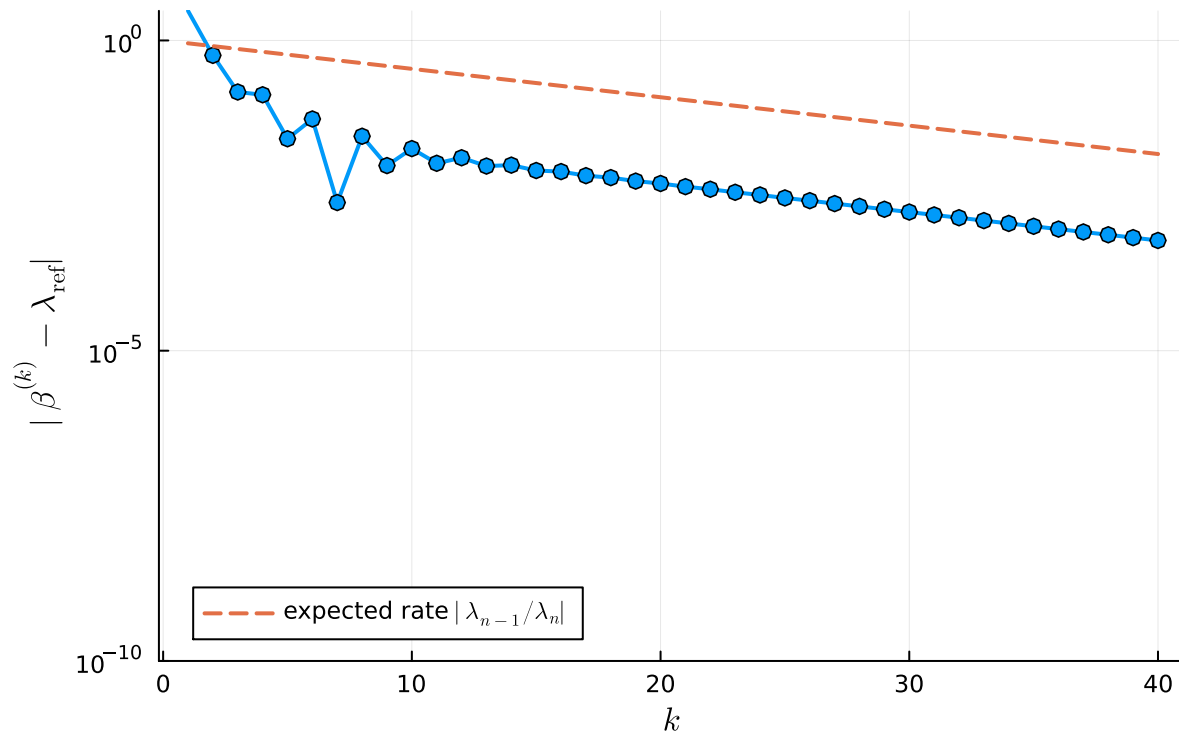
```
M_δ = 5×5 Matrix{Float64}:
      0.0   1.0  1.0   1.0  1.0
      0.0  -0.4  1.0   1.0  1.0
      0.0   0.0  0.6   1.0  1.0
      0.0   0.0  0.0  -0.9  1.0
      0.0   0.0  0.0   0.0  1.0
```

```
1  M_δ = triu(ones(5, 5), 1) + diagm(λ_δ)
```

We introduce a slider to tune the λ_{n-1} of equation (6):

- λ_{n-1} = ▬▬▬▬●▭ -0.9

Convergence of power iteration

```
1  let
2      λlargest = maximum(abs.(λref))  # Largest eigenvalue of M_δ (by construction)
3
4      x1 = ones(size(M_δ, 2))
5      results = power_method(M_δ, x1; maxiter=40)
6      error = abs.(results.history .- λlargest)
7      p = plot(error; mark=:o, label="", yaxis=:log, ylims=(1e-10, 3),
8          title="Convergence of power iteration",
9          ylabel=L"|β^{(k)} - λ_{\textrm{ref}}|",
10         xlabel=L"k", legend=:bottomleft, lw=2)
11
12     λₙ = 1.0
13     r_power = abs(λₙ₋₁ / λₙ)
14     plot!(p, k -> r_power^k; ls=:dash, label=L"expected rate $| λ_{n-1} / λ_n|$",
       lw=2)
15 end
```

Spectral transformations 🔗

The power method provides us with a simple algorithm to compute the *largest* eigenvalue of a matrix and its associated eigenvector. But what if one actually wanted to compute the smallest eigenvalue or an eigenvalue somewhere in the middle ?

In this section we discuss an extension to power iteration, which makes this feasible. We only need a small ingredient from linear algebra the **spectral transformations**.

We explore based on a few examples. Consider

⚠ TODO ⚠

Maybe pick a matrix with simpler eigenvalues

```
1  TODO("Maybe pick a matrix with simpler eigenvalues")
```

```
Ashift = 3×3 Matrix{Float64}:
        0.4  -0.6   0.2
       -0.3   0.7  -0.4
       -0.1  -0.4   0.5
```

```
1  Ashift = [ 0.4 -0.6 0.2;
2            -0.3 0.7 -0.4;
3            -0.1 -0.4 0.5]
```

Its eigenvalues and eigenvectors are:

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 2.220446049250313e-16
 0.43944487245360087
 1.160555127546399
vectors:
3×3 Matrix{Float64}:
 0.57735   0.676008   0.636852
 0.57735  -0.272642  -0.698428
 0.57735  -0.684602   0.326522
```

```
1  eigen(Ashift)
```

Now we add a multiple of the identity matrix, e.g.:

```
σ = 2
```

```
1  σ = 2  # Shift
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 1.9999999999999991
 2.4394448724536018
 3.1605551275463983
vectors:
3×3 Matrix{Float64}:
 0.57735   0.676008   0.636852
 0.57735  -0.272642  -0.698428
 0.57735  -0.684602   0.326522
```
```
1  eigen(Ashift + σ * I)   # Add 2 * identity matrix
```

Notice, how the eigenvectors are the same and only the eigenvalues have been shifted by σ.

Similarly:

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 0.3164001131587031
 0.40992932912404656
 0.5000000000000002
vectors:
3×3 Matrix{Float64}:
  0.636852  -0.676008  -0.57735
 -0.698428   0.272642  -0.57735
  0.326522   0.684602  -0.57735
```
```
1  let
2      A⁻¹ = inv(Ashift + σ * I)
3      eigen(A⁻¹)
4  end
```

Notice how this matrix still has the same eigenvectors (albeit in a different order) and the *inverted* eigenvalues of $\mathbf{A} + \sigma\mathbf{I}$.

```
▶ [0.5, 0.409929, 0.3164]
```
```
1  1 ./ eigvals(Ashift + σ * I)
```

We want to formalise our observation more rigourously. Recall that λ_i is an eigenvalue of $\mathbf{A}$ with (non-zero) eigenvector $\mathbf{x}_i$ if and only if

$$\mathbf{A}\mathbf{x}_i = \lambda_i \mathbf{x}_i. \tag{7}$$

We usually refer to the pair $(\lambda_i, \mathbf{x}_i)$ as an **eigenpair** of $\mathbf{A}$.

The statement of the **spectral transformations** is:

Theorem 1: Spectral transformations

Assume $\mathbf{A} \in \mathbb{R}^{n \times n}$ is diagonalisable. Let $(\lambda_i, \mathbf{x}_i)$ be an eigenpair of $\mathbf{A}$, then

1. If $\mathbf{A}$ is invertible, then $\left(\frac{1}{\lambda_i}, x_i\right)$ is an eigenpair of $\mathbf{A}^{-1}$.
2. For every $\sigma \in \mathbb{R}$ we have that $(\lambda_i - \sigma, x_i)$ is an eigenpair of $\mathbf{A} - \sigma\mathbf{I}$.
3. If $\mathbf{A} - \sigma\mathbf{I}$ is invertible, then $\left(\frac{1}{\lambda_i - \sigma}, x_i\right)$ is an eigenpair of $(\mathbf{A} - \sigma\mathbf{I})^{-1}$.

Proof: The result follows from a few calculations on top of (7). We proceed in order of the statements above.

1. Let $(\lambda_i, \mathbf{x}_i)$ be an arbitrary eigenpair of $\mathbf{A}$. Since $\mathbf{A}$ is invertible by assumption, $\lambda_i \neq 0$. Therefore for all eigenvalues λ_i of $\mathbf{A}$ the fraction $\frac{1}{\lambda_i}$ is meaningful and we can show:

$$\frac{1}{\lambda_i}\mathbf{x}_i = \frac{1}{\lambda_i}\mathbf{I}\mathbf{x}_i = \frac{1}{\lambda_i}\left(\mathbf{A}^{-1}\mathbf{A}\right)\mathbf{x}_i = \frac{1}{\lambda_i}\mathbf{A}^{-1}\left(\mathbf{A}\mathbf{x}_i\right) \overset{(6)}{=} \mathbf{A}^{-1}\mathbf{x}_i,$$

which indeed is the statement that $\left(\frac{1}{\lambda_i}, x_i\right)$ is an eigenpair of $\mathbf{A}^{-1}$.
2. The argument is similar to 1 and we leave it as an exercise.
3. This statement follows by combining statements 1. and 2.

Exercise 1

Assume $\mathbf{A} \in \mathbb{R}^{n \times n}$ is diagonalisable. Prove statement 2. of Theorem 1, that is for all $\sigma \in \mathbb{R}$: If $(\lambda_i, \mathbf{x}_i)$ is an eigenpair of $\mathbf{A}$, then $\mathbf{x}_i$ is also an eigenvector of $\mathbf{A} - \sigma\mathbf{I}$ with eigenvalue $\lambda_i - \sigma$.

Consider point 1. of Theorem 1 and assume that $\mathbf{A}$ has a *smallest* eigenvalue

$$0 < |\lambda_1| < |\lambda_2| \leq |\lambda_3| \leq \cdots \leq |\lambda_n|$$

then $\mathbf{A}^{-1}$ has the eigenvalues

$$|\lambda_1^{-1}| > |\lambda_2^{-1}| \geq |\lambda_3^{-1}| \geq \cdots \geq |\lambda_n^{-1}|,$$

thus a dominating (i.e. largest) eigenvalue $|\lambda_1^{-1}|$. **Applying** the `power_method` function (Algorithm 1) **to $\mathbf{A}^{-1}$** we thus converge to λ_1^{-1} from which we can deduce λ_1, the **eigenvalue of $\mathbf{A}$ closest to zero**.

Now consider point 3. and assume σ has been chosen such that

$$|\lambda_n - \sigma| \geq |\lambda_{n-1} - \sigma| \geq \cdots |\lambda_2 - \sigma| > |\lambda_1 - \sigma| > 0, \tag{8}$$

i.e. such that σ is closest to the eigenvalue λ_1. It follows

$$|(\lambda_n - \sigma)^{-1}| \leq |(\lambda_{n-1} - \sigma)^{-1}| \leq \cdots |(\lambda_2 - \sigma)^{-1}| < |(\lambda_1 - \sigma)^{-1}|,$$

such that the `power_method` function converges to $(\lambda_1 - \sigma)^{-1}$. From this value we can deduce λ_1, since we know σ. In other words by **applying Algorithm 1** to the **shift-and-invert matrix** $(\mathbf{A} - \sigma \mathbf{I})^{-1}$ enables us to find the **eigenvalue of $\mathbf{A}$ closest to σ**.

A naive application of Algorithm 1 would first compute $\mathbf{P} = (\mathbf{A} - \sigma \mathbf{I})^{-1}$ and then apply

$$\mathbf{y}^{(k)} = \mathbf{P} \mathbf{x}^{(k)}$$

in each step of the power iteration. However, for many problems the explicit computation of the inverse $\mathbf{P}$ is numerically unstable. Instead of computing $\mathbf{P}$ explicitly one instead obtains $\mathbf{y}^{(k)}$ by **solving a linear system**

$$(\mathbf{A} - \sigma \mathbf{I}) \, \mathbf{y}^{(k)} = \mathbf{x}^{(k)}$$

for $\mathbf{y}^{(k)}$, which is done **using LU factorisation**. We arrive at the following algorithm, where the changes compared to Algorithm 1 are marked in red.

Algorithm 2: Inverse iterations

Given

- a diagonalisable matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$,
- a shift $\sigma \in \mathbb{R}$, such that $\mathbf{A} - \sigma \mathbf{I}$ is invertible
- an initial guess $\mathbf{x}^{(1)} \in \mathbb{R}^n$

we iterate for $k = 1, 2, \ldots$:

1. Solve $(\mathbf{A} - \sigma \mathbf{I}) \mathbf{y}^{(k)} = \mathbf{x}^{(k)}$ for $\mathbf{y}^{(k)}$.
2. Find the index m such that $|y_m^{(k)}| = \|\mathbf{y}^{(k)}\|_\infty$
3. Set $\alpha^{(k)} = \frac{1}{y_m^{(k)}}$ and $\beta^{(k)} = \sigma + \frac{x_m^{(k)}}{y_m^{(k)}}$.

4. Set $\mathbf{x}^{(k+1)} = \alpha^{(k)}\mathbf{y}^{(k)}$

Note the additional change in step 3: In Algorithm 1 we obtained the estimate of the dominant eigenvalue as $y_m^{(k)}/x_m^{(k)}$. Here this estimate approximates $\frac{1}{\lambda_1-\sigma}$, the dominant eigenvalue of $(\mathbf{A} - \sigma\mathbf{I})^{-1}$. Therefore an estimate of λ_1 itself is obtained by solving

$$\frac{y_m^{(k)}}{x_m^{(k)}} = \frac{1}{\lambda_1 - \sigma} \quad \Longrightarrow \quad \lambda_1 = \sigma + \frac{x_m^{(k)}}{y_m^{(k)}}$$

for λ_1, which yields exactly the expression shown in step 3 of Algorithm 2.

An implementation of this algorithm is given in:

inverse_iterations (generic function with 1 method)
```julia
 1  function inverse_iterations(A, σ, x; maxiter=100)
 2      # A:  Matrix
 3      # σ:  shift
 4      # x:  initial guess
 5
 6      n = size(A, 1)
 7      x = normalize(x, Inf)
 8
 9      fact = lu(A - σ*I)   # Compute LU factorisation of A-σI
10
11      history = Float64[]
12      for k in 1:maxiter
13          y = fact \ x
14          m = argmax(abs.(y))
15          α = 1 / y[m]
16          β = σ + x[m] / y[m]
17          push!(history, β)
18          x = α * y
19      end
20
21      (; x, λ=last(history), history)
22  end
```

Notice that in this implementation we make use of the fact that once an **LU factorisation is computed** it can be **re-used for solving multiple linear systems** with changing right-hand sides: in our case we can expect to solve many linear systems involving the matrix $\mathbf{A} - \sigma\mathbf{I}$. Therefore we **compute the LU factorisation** only once, namely at the beginning of the algorithm and before entering the iterative loop.

Since for dense matrices computing the factorisation scales $O(n^3)$, but solving linear systems based on the factorisation only scales $O(n^2)$ ([recall chapter 5](#)), this reduces the cost per iteration.

To investigate the possibilities enabled by inverse iterations, we consider a few examples using the following triangular matrix
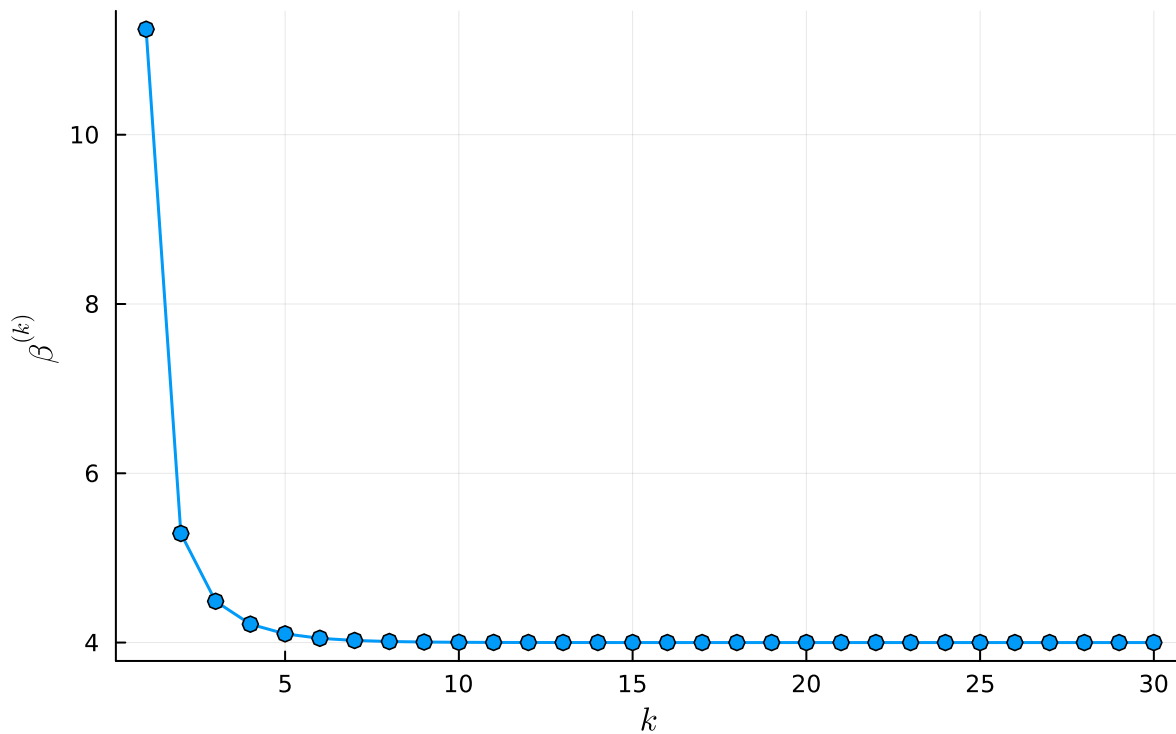
```
C = 3×3 Matrix{Float64}:
    -4.0   3.0   4.0
     0.0   4.0   5.0
     0.0   0.0   2.0
```

```
1  C = [-4.0  3.0  4.0;
2        0.0  4.0  5.0;
3        0.0  0.0  2.0]
```

which has eigenvalues $\lambda_1 = -4$, $\lambda_2 = 4$ and $\lambda_3 = 2$.

Since it has no unique dominant eigenvalue ($|\lambda_1| = |\lambda_2| = 4$) **plain power iterations** do not converge, but rather oscillate:

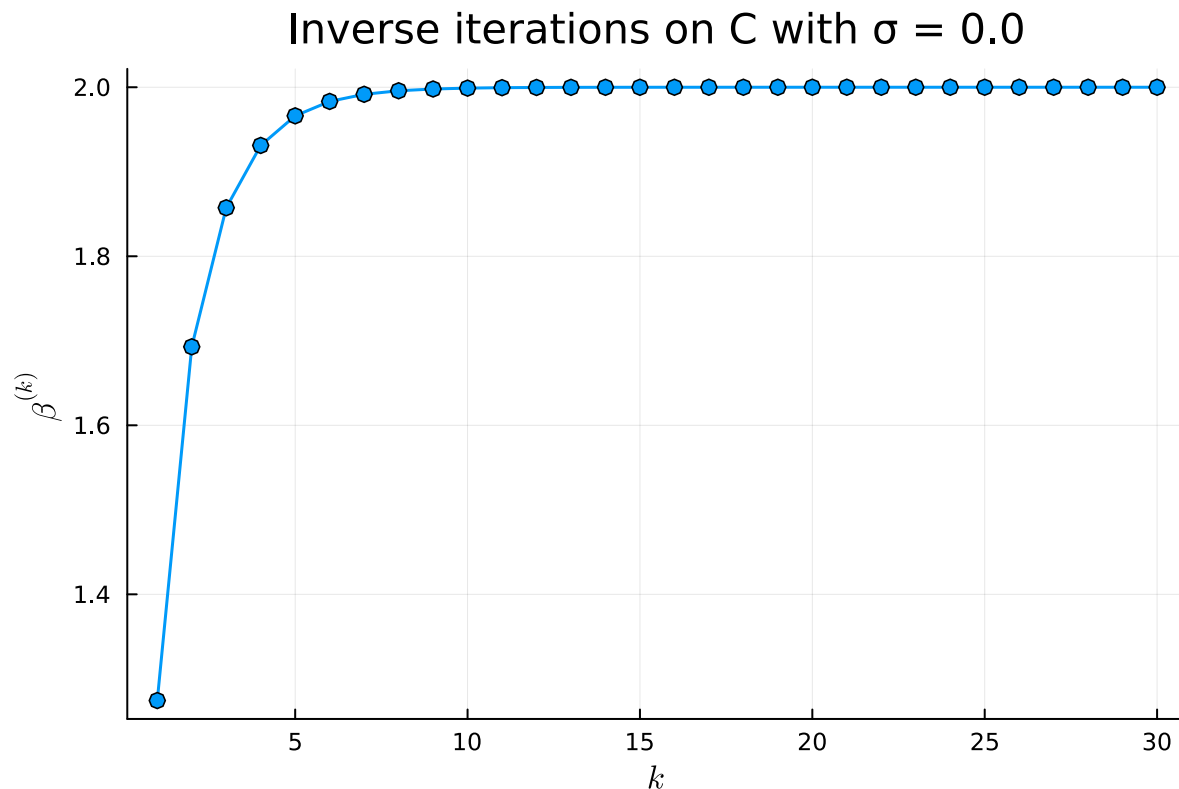# Power iterations on C


```
1  let
2      xinit = randn(size(C, 2))
3      res = power_method(C, xinit; maxiter=30)
4
5      plot(res.history; mark=:o, c=1, label="", lw=1.5, ylabel=L"β^{(k)}",
       xlabel=L"k",
6          title="Power iterations on C")
7  end
```

In contrast for **inverse iterations** (with $\sigma = 0$) what matters are the eigenvalues of $\mathbf{C}^{-1}$, which are $1/\lambda_i$ or

$$\frac{1}{\lambda_1} = -\frac{1}{4} \qquad \frac{1}{\lambda_2} = \frac{1}{4} \qquad \frac{1}{\lambda_3} = \frac{1}{2}$$

Therefore there is a single dominant eigenvalue ($\frac{1}{2}$) and inverse iterations will converge to the eigenvalue 2:
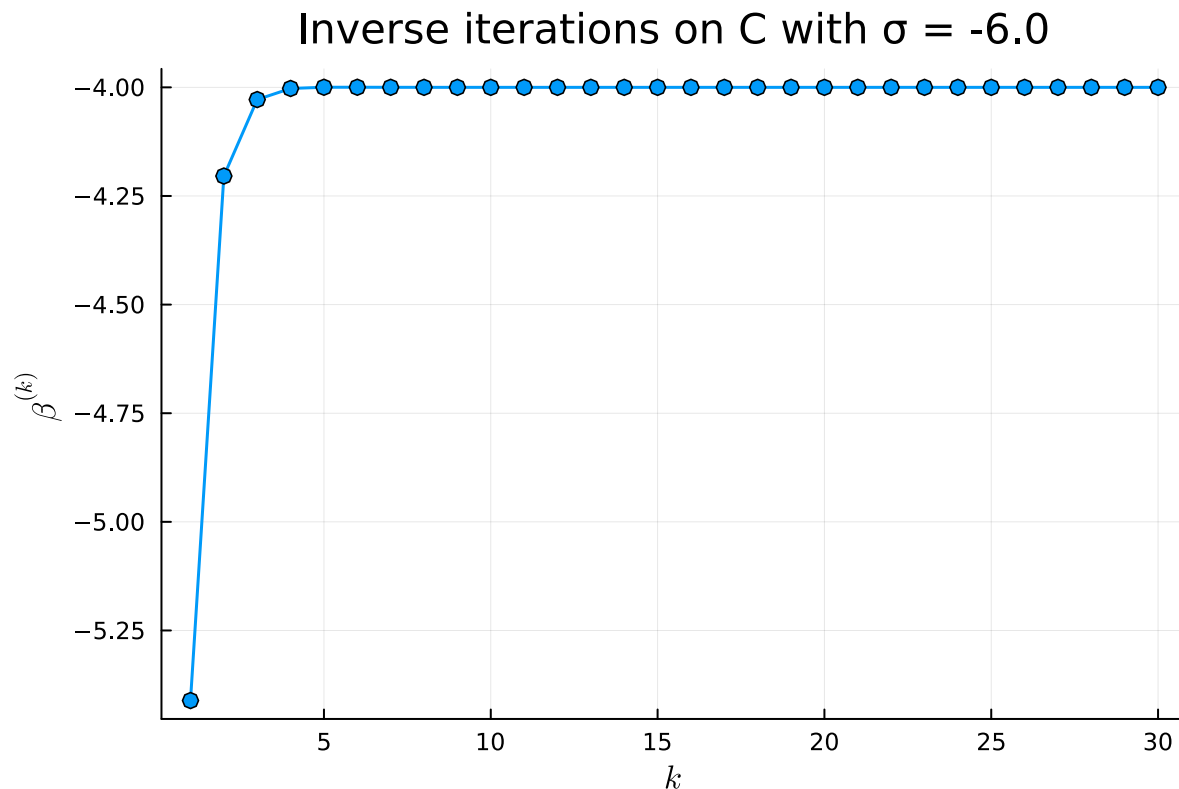
Inverse iterations on C with σ = 0.0

```
1  let
2      σ = 0.0   # Just perform plain inverse iterations
3
4      xinit = randn(size(C, 2))
5      res = inverse_iterations(C, σ, xinit; maxiter=30)
6
7      plot(res.history; mark=:o, c=1, label="", lw=1.5, ylabel=L"β^{(k)}",
       xlabel=L"k",
8          title="Inverse iterations on C with σ = $σ")
9  end
```

Now suppose we knew that $\mathbf{C}$ has one eigenvalue near -6, such that we take $\sigma = -6$. Then what matters for convergence in the inverse iterations are the eigenvalues of $(\mathbf{C} - (-6)\mathbf{I})^{-1}$, which are $\mu_i = \frac{1}{\lambda_i - \sigma}$ or

$$\mu_1 = \frac{1}{-4 + 6} = \frac{1}{2} \qquad \mu_2 = \frac{1}{4 + 6} = \frac{1}{10} \qquad \mu_3 = \frac{1}{2 + 6} = \frac{1}{8},$$

such that $\mu_1 = \frac{1}{2}$ is the dominating eigenvalue. Therefore with $\sigma = -6$ inverse iterations converge to $-6 + 1/\mu_1 = -6 + 1/(\frac{1}{2}) = -4$:

Inverse iterations on C with σ = -6.0

```
1  let
2      σ = -6.0
3
4      xinit = randn(size(C, 2))
5      res = inverse_iterations(C, σ, xinit; maxiter=30)
6
7      plot(res.history; mark=:o, c=1, label="", lw=1.5, ylabel=L"β^{(k)}",
       xlabel=L"k",
8          title="Inverse iterations on C with σ = $σ")
9  end
```

Convergence of inverse iterations 🔗

Inserting $\mathbf{A} - \sigma\mathbf{I}$ in the place of $\mathbf{A}$ in (6) and recalling the eigenvalue ordering (8), i.e.
$|\lambda_n - \sigma| \geq |\lambda_{n-1} - \sigma| \geq \cdots |\lambda_2 - \sigma| > |\lambda_1 - \sigma| > 0$ one can show similarly that **inverse iterations converge linearly** with rate

$$r_{\text{inviter}} = \lim_{k \to \infty} \frac{|\beta^{(k+1)} - \lambda_1|}{|\beta^{(k)} - \lambda_1|} = \left| \frac{\lambda_1 - \sigma}{\lambda_2 - \sigma} \right| \qquad \text{as } k \to \infty. \qquad (9)$$

This implies that convergence is fastest if σ is closest to the targeted eigenvalue λ_1 than to all other eigenvalues of $\mathbf{A}$.

Finally let us consider a case where we use a slider to be able to change the applied shift:

```
5×5 Matrix{Float64}:
 1.0  1.0   1.0   1.0  1.0
 0.0  0.75  1.0   1.0  1.0
 0.0  0.0   0.6   1.0  1.0
 0.0  0.0   0.0  -0.4  1.0
 0.0  0.0   0.0   0.0  0.0
```

```julia
1  begin
2      λT = [1, 0.75, 0.6, -0.4, 0]   # The reference eigenvalues we will use
3      T  = triu(ones(5, 5), 1) + diagm(λT)
4  end
```
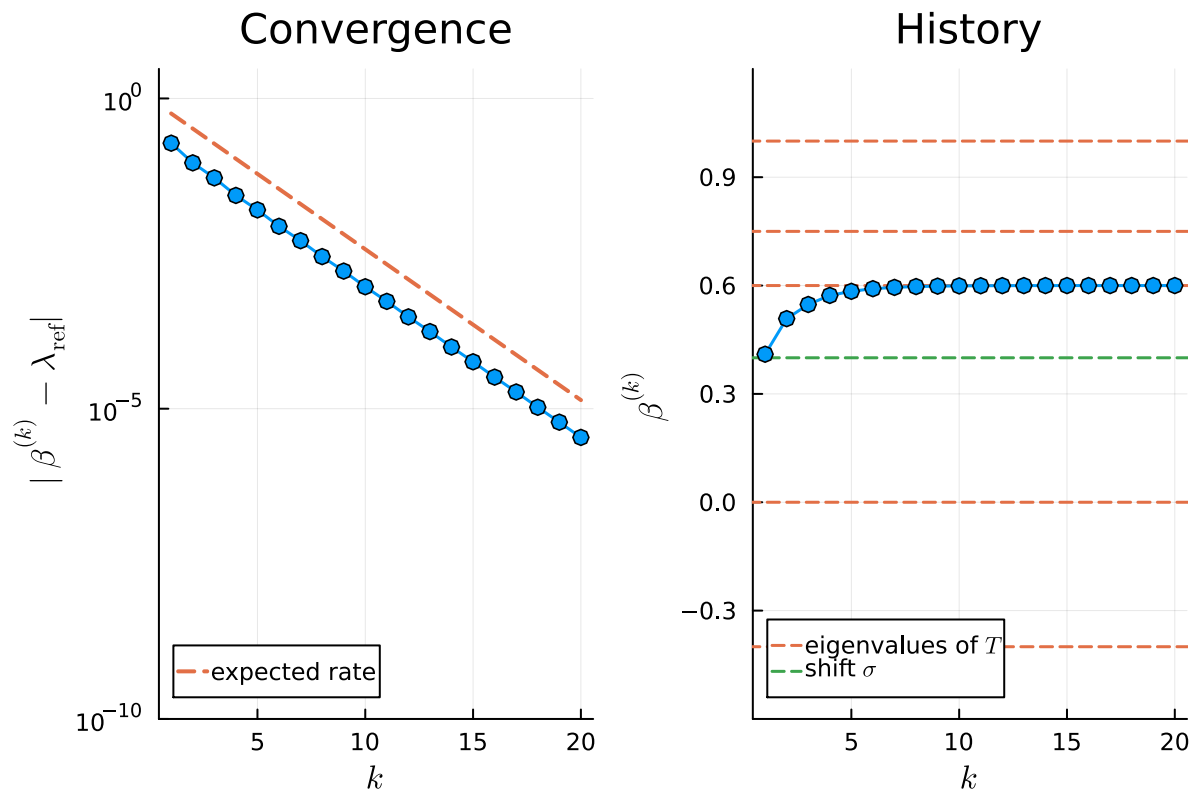
This time the slider allows to tune the value of σ (which we store in the variable σT):

- σT = ▬▬▬▬●▭▭▭ 0.4

This value of σ results in a rate:

```
0.5714285714285713
```

```julia
1  begin
2      # compute |λ_5 - σ|  >  |λ_4 - σ|  >  |λ_3 - σ|  >  |λ_2 - σ|  >  |λ_1 - σ|
3      # We need sort to get the data in that order
4      diff_sorted = sort(abs.(λT .- σT); rev=true)
5
6      # take last element (|λ_1 - σ|) and last-but-one (|λ_2 - σ|)
7      r_inviter = diff_sorted[end] / diff_sorted[end-1]
8  end
```

```
1  let
2      # The eigenvalue the iteration targets
3      i = argmin(abs.(λT .- σT))
4      λtarget = λT[i]
5
6      p = plot(title="Convergence",
7          ylabel=L"|β^{(k)} - λ_{\textrm{ref}}|", xlabel=L"k", legend=:bottomleft,
8          yaxis=:log, ylims=(1e-10, 3))
9
10     q = plot(title="History", xlabel=L"k", ylabel=L"β^{(k)}",
11         legend=:bottomleft, ylims=(-0.6, 1.2))
12     hline!(q, λT, ls=:dash, label=L"eigenvalues of $T$", c=2, lw=1.5)
13     hline!(q, [σT], ls=:dash, label=L"shift $σ$", c=3, lw=1.5)
14
15     if abs(σT - λtarget) > 1e-10  # Guard to prevent numerical issues
16         results = inverse_iterations(T, σT, xstart; maxiter=20)
17         error = abs.(results.history .- λtarget)
18
19         plot!(p, error; mark=:o, label="", lw=1.5)
20         plot!(p, k -> r_inviter^k; ls=:dash, label="expected rate", lw=2)
21
22         plot!(q, results.history, mark=:o, c=1, label="", lw=1.5)
23     end
24     plot(p, q; layout=(1,2))
25  end
```

Optional: Dynamic shifting 🔗

In the discussion in the previous section we noted that the convergence of inverse iterations is best if σ is chosen close to the eigenvalue of the matrix $\mathbf{A}$. Since **inverse iterations** (Algorithm 2) actually **produce an estimate for the eigenvalue** in each iteration (the $\beta^{(k)}$), a natural idea is to **update the shift**: instead of using the same σ in each iteration, we select a different $\sigma = \beta^{(k)}$ *dynamically* based on the best eigenvalue estimate currently available to us.

This also implies that for solving the linear system in step 1, i.e. $(\mathbf{A} - \sigma\mathbf{I})\,\mathbf{y}^{(k)} = \mathbf{x}^{(k)}$ the system matrix is different for each k. Therefore pre-computing the LU factorisation before entering the iteration loop is no longer possible.

We arrive at the following implementation for an **inverse iteration algorithm with dynamic shifting**:

dynamic_shifting (generic function with 1 method)

```julia
 1  function dynamic_shifting(A, σ, x; maxiter=100, tol=1e-8)
 2      # A:  Matrix
 3      # σ:  shift
 4      # x:  initial guess
 5  
 6      n = size(A, 1)
 7      x = normalize(x, Inf)
 8  
 9      history = Float64[]
10      for k in 1:maxiter
11          y = (A - σ*I) \ x    # LU-factorise (A - σ*I) and solve system
12          m = argmax(abs.(y))
13          α = 1 / y[m]
14          β = σ + x[m] / y[m]
15          push!(history, β)
16          x = α * y
17  
18          if abs(σ - β) < tol  # We are converged, so exit the iterations.
19              break
20          end
21          σ = β
22      end
23  
24      (; x, λ=last(history), history)
25  end
```

Since we now need to compute a fresh LU factorisation in each iteration, the overall cost of `dynamic_shifting` is larger than the cost of `inverse_iterations`. On the upside, convergence is

improved: We go from linear to **quadratic convergence** (see chapter 8.3 of Driscoll, Brown: Fundamentals of Numerical Computation).
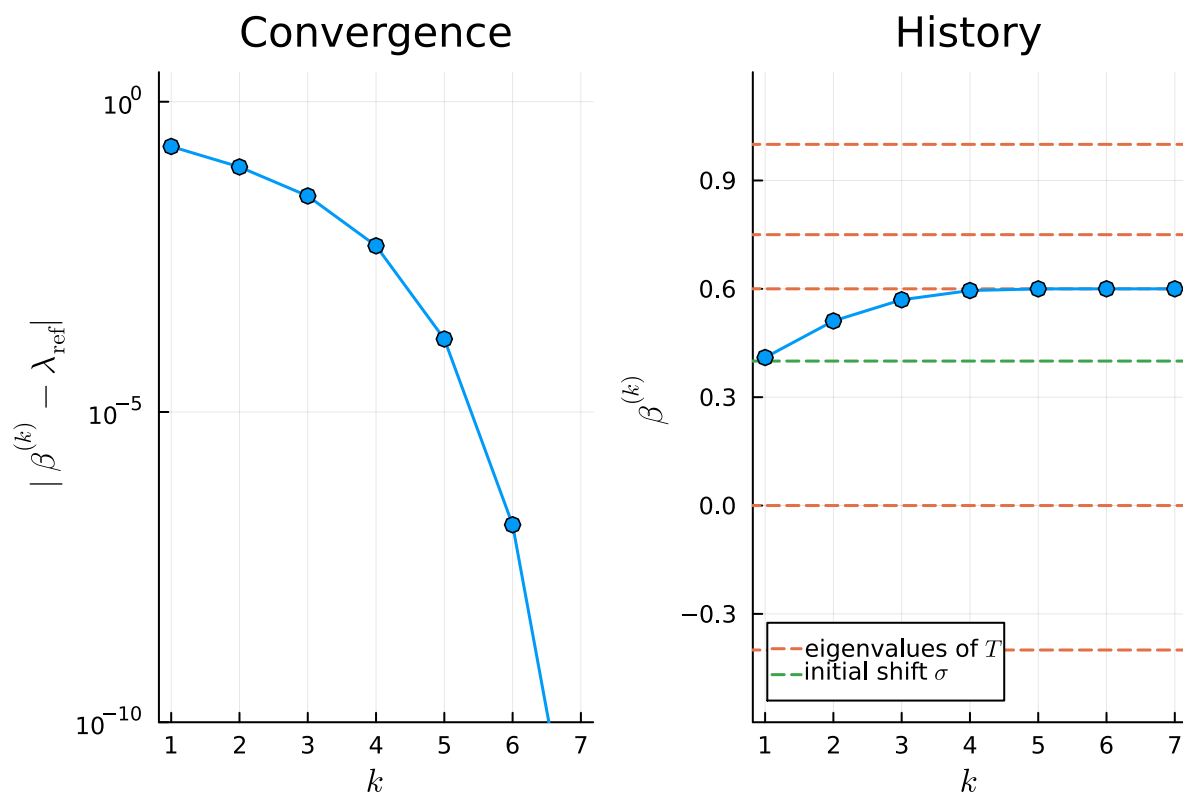
This is easily checked numerically. We use the same matrix

```
5×5 Matrix{Float64}:
 1.0  1.0   1.0   1.0  1.0
 0.0  0.75  1.0   1.0  1.0
 0.0  0.0   0.6   1.0  1.0
 0.0  0.0   0.0  -0.4  1.0
 0.0  0.0   0.0   0.0  0.0
```
```
1  T
```

and introduce another slider to tune the value of the *initial* shift σ, which we store in the variable σD:

- σD = [====●====] 0.4

The much faster quadratic convergence is clearly visible.

```
1  xstart = randn(size(T, 2));
```

Numerical analysis