Click here to view the PDF version.

```julia
begin
    using LinearAlgebra
    using PlutoUI
    using PlutoTeachingTools
    using Plots
    using LaTeXStrings
    using HypertextLiteral: @htl, @htl_str
end
```

Table of Contents

Iterative methods for linear systems

In the previous notebook we looked at direct methods for solving linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$ based on LU factorisation of the system matrix $\mathbf{A}$. We saw that even for sparse matrices we may need $O(n^3)$ computational time to perform the factorisation and $O(n^2)$ memory and to store the $\mathbf{L}$ and $\mathbf{U}$ factors due to fill in.

Both can make the **solution of linear systems prohibitively expensive for large matrices**. In this notebook we will develop *iterative methods*. These build a sequence of solution vectors $\mathbf{x}^{(k)}$, which is designed to converge to the solution $\mathbf{x}$, that is $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}$. Typically these methods are based on performing matrix-vector products $\mathbf{A}\mathbf{v}$ in each iteration step. Exactly this difference to the direct methods — that is *not* employing the matrix $\mathbf{A}$, but *only the matrix-vector product* — is what often leads to an overall reduction of the computational cost.

Richardson iterations

A general framework for building iterative methods for solving linear systems is Richardson's method. The idea is to introduce a **matrix splitting** $\mathbf{A} = \mathbf{P} - \mathbf{R}$ where $\mathbf{P}$ is an invertible matrix and $\mathbf{R} = \mathbf{P} - \mathbf{A}$. With this idea we rewrite $\mathbf{A}\mathbf{x} = \mathbf{b}$ to

$$(\mathbf{P} - \mathbf{R})\,\mathbf{x} = \mathbf{b} \qquad \Leftrightarrow \qquad \mathbf{P}\mathbf{x} = (\mathbf{P} - \mathbf{A})\,\mathbf{x} + \mathbf{b} \qquad \Leftrightarrow \qquad \mathbf{x} = \mathbf{P}^{-1}\left[(\mathbf{P} - \mathbf{A})\mathbf{x} + \mathbf{b}\right]$$

If we define $g(\mathbf{x}) = \mathbf{P}^{-1}\left[(\mathbf{P} - \mathbf{A})\mathbf{x} + \mathbf{b}\right]$ we observe this to be exactly the setting of the multi-dimensional fixed-point methods we developed in chapter 4, i.e. our goal is to obtain a solution $\mathbf{x}_*$ with $\mathbf{x}_* = g(\mathbf{x}_*) = \mathbf{P}^{-1}\left[(\mathbf{P} - \mathbf{A})\mathbf{x}_* + \mathbf{b}\right]$.

Starting from an initial vector $\mathbf{x}^{(0)}$ we thus iterate

$$\mathbf{P}\mathbf{x}^{(k+1)} = (\mathbf{P} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b}, \qquad k = 0, 1, \ldots, \tag{2}$$

that is in each step we solve the linear system $\mathbf{P}\mathbf{x}^{(k+1)} = \widetilde{\mathbf{b}}^{(k)}$ where $\widetilde{\mathbf{b}}^{(k)} = (\mathbf{P} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b}$. Assuming this sequence converges, i.e. that $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}_*$, then

$$\mathbf{P}\mathbf{x}_* = (\mathbf{P} - \mathbf{A})\mathbf{x}_* + \mathbf{b} \qquad \Leftrightarrow \qquad \mathbf{0} = \mathbf{b} - \mathbf{A}\mathbf{x}_*,$$

i.e. the limit $\mathbf{x}_*$ is indeed the solution to the original equation.

The matrix $\mathbf{P}$ is usually referred to as the **preconditioner**. Since we need to solve a linear system $\mathbf{P}\mathbf{u}^{(k)} = \mathbf{r}^{(k)}$ in *each iteration* k, an important criterion for a good preconditioner is that solving such linear systems is cheap.

Let us rewrite equation (2) as

$$\mathbf{P}\underbrace{\left(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right)}_{=\mathbf{u}^{(k)}} = \underbrace{\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}}_{=\mathbf{r}^{(k)}}.$$

The quantity $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ appearing in this expression is the **residual** at iteration k. As usual it measures how far the iterate $\mathbf{x}^{(k)}$ is from being a solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$: if $\mathbf{r}^{(k)} = \mathbf{0}$, then indeed $\mathbf{A}\mathbf{x}^{(k)} = \mathbf{b}$, i.e. that $\mathbf{x}^{(k)}$ is the solution.

We formulate Richardson's iterations:

Algorithm 1: Richardson iteration

Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a system matrix, right-hand side $\mathbf{b} \in \mathbb{R}^n$, an invertible preconditioner $\mathbf{P} \in \mathbb{R}^{n\times n}$, an initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$ as well as the desired convergence tolerance ε. For $k = 0, 1, \ldots$ iterate:

1. Compute residual $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$
2. Solve linear system $\mathbf{P}\mathbf{u}^{(k)} = \mathbf{r}^{(k)}$ for the update $\mathbf{u}^{(k)}$.
3. Compute new $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{u}^{(k)}$.

The iteration is stopped as soon as $\frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} < \varepsilon$.

We will discuss the rationale behind the stopping condition $\|\mathbf{r}^{(k)}\| / \|\mathbf{b}\| < \varepsilon$ in the subsection on error control below.

Comparing with our discussion on the fixed point methods we studied in chapter 4 we notice that Algorithm 1 is essentially fixed-point iteration $\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)})$ with the map g given by

$$g(\mathbf{x}) = \mathbf{x} + \mathbf{P}^{-1}\underbrace{(\mathbf{b} - \mathbf{A}\mathbf{x})}_{=\mathbf{r}} \tag{3}$$

and as discussed above the fixed point $\mathbf{x}_* = g(\mathbf{x}_*)$ necessarily satisfies $\mathbf{0} = \mathbf{r}_* = \mathbf{b} - \mathbf{A}\mathbf{x}_*$ and therefore is a solution to the linear system.

An implementation of Algorithm 1 in Julia is given by

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

```julia
function richardson(A, b, P; x=zero(b), tol=1e-6, maxiter=100)
    history  = [float(x)]  # Keep history of xs
    relnorms = Float64[]   # Track relative residual norm
    for k in 1:maxiter
        r = b - A * x

        relnorm = norm(r) / norm(b)
        push!(relnorms, relnorm)
        if relnorm < tol
            break
        end

        u = P \ r
        x = x + u

        push!(history, x)
    end
    (; x, relnorms, history)  # Return current iterate and history
end
```

Let us consider the test problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ with

```
A = 100×100 Matrix{Float64}:
    34.8048      0.698064   0.0912903   0.110129  …  -0.409052    0.697422  -1.04543
     0.666056   41.1921     0.510527   -2.32168       1.10391    -0.526851  -0.08354
     0.474286   -1.39428   41.8852      0.150974     -0.688416    0.915631   0.46982
    -0.473043    1.05738   -1.05232    29.5658       -2.21622    -0.771756  -1.45717
    -1.24018     1.46078    0.36293    -0.129446     -0.164112   -0.428056  -1.24723
     0.0361345  -0.347109   0.465493    0.30022   …  -0.0134317  -0.766304   0.394693
     1.4133      0.545532   0.208997   -1.36675      -1.76489    -1.31785    0.870358
     ⋮                                            ⋱
     0.655297   -0.266465  -0.609266   -0.657441     -0.49885     0.592234   1.18366
    -0.398953   -0.468695   0.416494    1.38779   …  -1.38401     0.108533   1.41024
    -1.23063    -0.396662   1.63005    -0.927915      0.12108     1.16041   -0.392143
     0.369631   -1.42732   -0.243615    1.44618      34.2564     -0.452845  -0.0258712
     0.684164   -0.162484  -1.09723    -0.61984       0.131653   30.7371    -1.20176
     0.648161    0.492944  -0.440426   -0.193745      0.109915   -1.12334   33.8566
```

```julia
# Generate a random matrix, which has large entries on the diagonal
A = randn(100, 100) + Diagonal(15 .+ 50rand(100))
```

```julia
b = rand(100);
```

In this case the problem is sufficiently small that it's reference can still be computed by a direct method, e.g. the LU factorisation performed by default when employing Julia's `\` operator:

```
x_reference =
▸[0.0184302, 0.000181953, 0.00371573, 0.0200995, 0.0210902, 0.0189052, 0.0129691, 0.005205
```

```julia
x_reference = A \ b
```

To apply Richeardson iterations, we need a preconditioner $\mathbf{P}$. Due to the construction of $\mathbf{A}$ we chose it has its largest entries in each row on the diagonal. A particularly simple preconditioner, which typically works well for such matrices is to use the diagonal of $\mathbf{A}$ as $\mathbf{P}$, i.e.

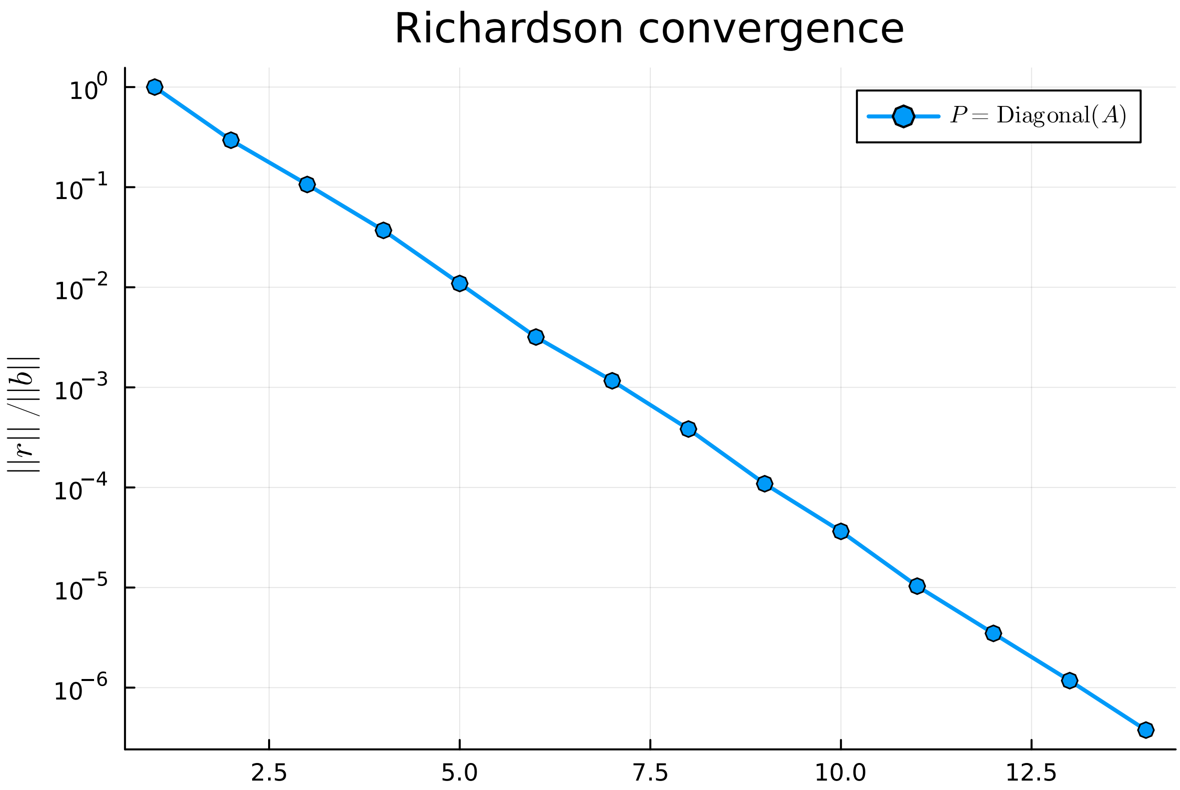
```
P = 100×100 Diagonal{Float64, Vector{Float64}}:
     34.8048    ⋅        ⋅        ⋅        ⋅       …     ⋅        ⋅        ⋅        ⋅
       ⋅     41.1921     ⋅        ⋅        ⋅             ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅     41.8852     ⋅        ⋅             ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅     29.5658     ⋅             ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅     42.0235          ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅       …     ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅             ⋅        ⋅        ⋅        ⋅
      ⋮                                                ⋱                   
       ⋅        ⋅        ⋅        ⋅        ⋅             ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅       …     ⋅        ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅          17.3058     ⋅        ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅             ⋅     34.2564     ⋅        ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅             ⋅        ⋅     30.7371     ⋅
       ⋅        ⋅        ⋅        ⋅        ⋅             ⋅        ⋅        ⋅     33.8566
```

```
P = Diagonal(A)
```

Applying this preconditioner within the Richardson iterations, yields the correct result to the chosen tolerance:

```
2.082247593446218e-8
```

```
begin
    richardson_result = richardson(A, b, P; tol=1e-6)
    maximum(abs, richardson_result.x - x_reference)
end
```



```julia
let
    plot(richardson_result.relnorms;
        yaxis=:log, mark=:o, lw=2, ylabel=L"||r|| / ||b||",
        title="Richardson convergence", label=L"$P = \textrm{Diagonal(}A\textrm{)}$")
end
```

Computational cost

Our main motivation for considering iterative methods was to overcome the $O(n^3)$ computational cost of Gaussian elimination / LU factorisation for solving linear systems.

Let us revist Richardson iteration (Algorithm 1) and discuss the computational cost for the case where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a full matrix. Recall that the cost of LU factorisation scaled as $O(n^3)$ for this setting.

1. In the **first step** the most costly operation is the **computation of the matrix-vector product** $\mathbf{A}\mathbf{x}^{(k)}$. For a full matrix this costs $O(n^2)$.
2. In the **second step** we solve $\mathbf{P}\mathbf{u}^{(k)} = \mathbf{r}^{(k)}$. If $\mathbf{P}$ is a **diagonal matrix** (like we just considered), this costs $O(n)$, whereas for a **triangular matrix** (where one would perform backward or forward substitution) the cost is $O(n^2)$.
3. The computation of the new $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{u}^{(k)}$ again costs only $O(n)$.

Overall **for full matrices** the **cost of Richardson iterations** is thus only $O(n^2)$.

For **sparse matrices** $\mathbf{A}\mathbf{x}^{(k)}$ can computed in $O(n)$ computational cost. Similarly using clever preconditioners step 2 can be done in $O(n)$ time. Examples would be a sparsity-adapted forward substitution algorithm or again a diagonal preconditioner. Overall Richardson iterations thus **only cost** $O(n)$ — in stark contrast to the $O(n^3)$ cost also required for LU factorisation in this setting.

> **Observation: Computational cost of iterative methods**
>
> When solving linear systems using iterative methods, the computational cost (per iteration) scales
>
> - for **full matrices** $\mathbf{A} \in \mathbb{R}^{n\times n}$ as $O(n^2)$
> - for **sparse matrices** as $O(n)$.
>
> A recurring theme is that the cost of the matrix-vector product $\mathbf{A}\mathbf{x}^{(k)}$ essentially determis the cost of the iterative scheme.

Convergence analysis

Having established above that Richardson iteration indeed leads to numerically the correct answer, we proceed to analyse its convergence.

As an iterative method Algorithm 1 can be brought into the setting of a fixed point method $\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)})$ by choosing

$$g(\mathbf{x}) = \mathbf{x} + \mathbf{P}^{-1}\left(\mathbf{b} - \mathbf{A}\mathbf{x}\right).$$

In line with our discussion in Root finding and fixed-point problems the convergence depend on the properties of the **Jacobian** (i.e the derivative) of this map at the fixed point. The term $\mathbf{P}^{-1}\mathbf{b}$ does not depend on $\mathbf{x}$ and thus drops, for the rest we compute:

$$g'(\mathbf{x}_*) = \underbrace{\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}}_{=\mathbf{B}}$$

where the matrix $\mathbf{B}$ is usually referred to as the **iteration matrix**. Notice that here we employed the short-hand g' to denote the Jacobian matrix $\mathbf{J}_g$ of the fixed-point map g.

Recall that our condition for convergence was $\|g'(\mathbf{x}_*)\| < 1$, which for the case of Richardson iterations is thus $\|\mathbf{B}\| < 1$. As a reminder, recall that the definition of the matrix norm for a matrix

$M \in \mathbb{R}^{n\times n}$ is

$$\|\mathbf{M}\| = \max_{\mathbf{0}\neq\mathbf{v}\in\mathbb{R}^n} \frac{\|\mathbf{M}\mathbf{v}\|}{\|\mathbf{v}\|}.$$

We summarise in a theorem:

Theorem 1

Given a system matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, RHS $\mathbf{b} \in \mathbb{R}^n$ and preconditioner $\mathbf{P} \in \mathbb{R}^{n\times n}$ invertible, the Richardson method with iteration matrix $\mathbf{B} = \mathbf{I} - \mathbf{P}^{-1}\mathbf{A}$ converges for *any initial guess* $\mathbf{x}^{(0)}$ if $\|\mathbf{B}\| < 1$. Moreover

$$\|\mathbf{x}_* - \mathbf{x}^{(k)}\| \leq \|\mathbf{B}\|^k \|\mathbf{x}_* - \mathbf{x}^{(0)}\|.$$

This is **linear convergence with rate** $\|\mathbf{B}\|$.

Notice that the theorem mentions that Richardson iterations converges for *any initial guess*, which for general fixed-point methods is not true.

This is in fact a consequence of the fact that g is a linear function, i.e. that its Taylor expansion

$$g(\mathbf{x}^{(k)}) = g(\mathbf{x}_*) + g'(\mathbf{x}_*)(\mathbf{x}^{(k)} - \mathbf{x}_*)$$

terminates *exactly* after the linear term *(see below for an explicit verification)*.

▶ **Explicit verification the Taylor expansion terminates**

Denoting the error in the k-th iteration as $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}_*$ and subtrtacting $\mathbf{x}_*$ from the above Taylor expansion one can thus show that

$$\mathbf{e}^{(k+1)} = \underbrace{\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)}_{=\mathbf{B}\,=\,g'(\mathbf{x}_*)} \mathbf{e}^{(k)}, \qquad (4)$$

such that if $\|\mathbf{B}\| < 1$ the error is *guaranteed to shrink* in an iteration, independent on the current iterate $\mathbf{x}^{(k)}$.

Exercise

Show that for the fixed-point map of Richardson iteration $g(\mathbf{x}) = \mathbf{x} + \mathbf{P}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x})$ we indeed have that $\mathbf{e}^{(k+1)} = \mathbf{B}\,\mathbf{e}^{(k)}$ where $\mathbf{B} = \mathbf{I} - \mathbf{P}^{-1}\mathbf{A}$. Making use of the inequality

$$\|\mathbf{M}\mathbf{v}\| \leq \|\mathbf{M}\|\,\|\mathbf{v}\|$$

and by adapting our arguments in the convergence analysis discussion of Root finding and fixed-point problems show that for Richardson iteration the error always tends to zero as the iteration progresses, i.e. that $\lim_{k\to\infty} \|\mathbf{e}^{(k)}\| = 0$.

Difference between linear and non-linear functions g

Notice that the result (4) and Theorem 1 of this notebook only applies for fixed-point problems with a *linear* function g. This is therefore a different setting to the Convergence analysis discussion in Root finding and fixed-point problems, where we discussed the more general case of a *non-linear* fixed point functions g.

For non-linear g we can only state $\mathbf{e}^{(k+1)} = g'(\mathbf{x}_*)\mathbf{e}^{(k)} + O(\text{small})$, where the small term has a size that is quadratic in $\|\mathbf{e}^{(k)}\|$. Therefore an exact equality like (4) does not hold, which in particular means that we need the higher-order terms hidden in $O(\text{small})$ to be sufficiently small in order for the error to actually shrink going from k to $k+1$ (implying convergence). This is only ensured if $\|\mathbf{e}^{(k)}\|$ is small enough, i.e. if one starts the iterations sufficiently close to the fixed point x_*. This is why in Theorem 1 and Theorem 3 of Root finding and fixed-point problems we have the condition that $x^{(0)}$ needs to be taken from an interval around x_*.

From Theorem 1 we take away that the norm of iteration matrix $\|\mathbf{B}\|$ is the crucial quantity to determine not only *if* Richardson iterations converge, but also *at which rate*. Recall in Lemma 4 of Direct methods for linear systems we had the result that for any matrix $\mathbf{B} \in \mathbb{R}^{m\times n}$

$$\|\mathbf{B}\| = \sqrt{\lambda_{\max}^{\text{abs}}(\mathbf{B}^T\mathbf{B})}. \tag{5}$$

With this and the condition $\|\mathbf{B}\| < 1$ for convergence in Theorem we can understand the **role of preconditioning**. We again consider the matrix $\mathbf{A}$ from before.

```
100×100 Matrix{Float64}:
 34.8048      0.698064   0.0912903   0.110129  …  -0.409052    0.697422  -1.04543
  0.666056   41.1921     0.510527   -2.32168       1.10391    -0.526851  -0.08354
  0.474286   -1.39428   41.8852      0.150974     -0.688416    0.915631   0.46982
 -0.473043    1.05738   -1.05232    29.5658       -2.21622    -0.771756  -1.45717
 -1.24018     1.46078    0.36293    -0.129446     -0.164112   -0.428056  -1.24723
  0.0361345  -0.347109   0.465493    0.30022   …  -0.0134317  -0.766304   0.394693
  1.4133      0.545532   0.208997   -1.36675      -1.76489    -1.31785    0.870358
  ⋮                                            ⋱
  0.655297   -0.266465  -0.609266   -0.657441     -0.49885     0.592234   1.18366
 -0.398953   -0.468695   0.416494    1.38779   …  -1.38401     0.108533   1.41024
 -1.23063    -0.396662   1.63005    -0.927915      0.12108     1.16041   -0.392143
  0.369631   -1.42732   -0.243615    1.44618      34.2564     -0.452845  -0.0258712
  0.684164   -0.162484  -1.09723    -0.61984       0.131653   30.7371    -1.20176
  0.648161    0.492944  -0.440426   -0.193745      0.109915   -1.12334   33.8566
```

```
A
```

If we were not to perform any preconditioning, i.e. $\mathbf{P} = \mathbf{I}$, then $\|\mathbf{B}\| = \|\mathbf{I} - \mathbf{A}\|$ becomes

```
69.46077995854577
```

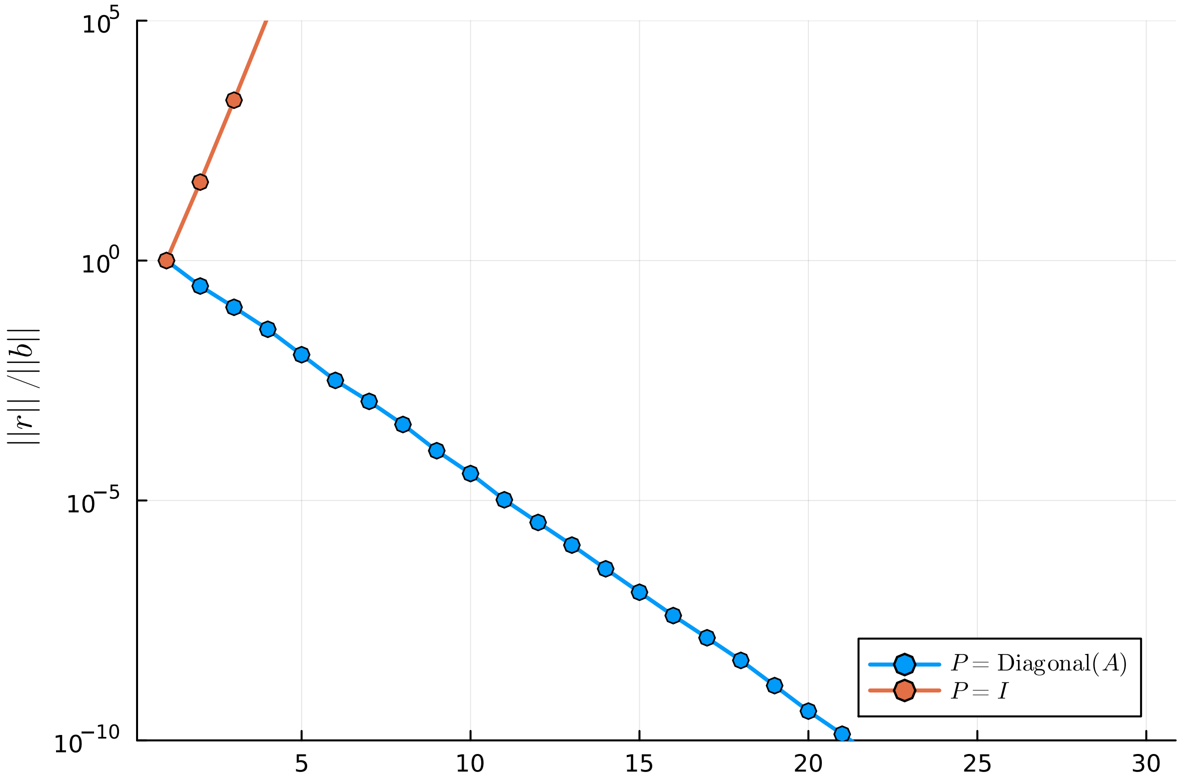
```
let
    B = I - A     # Iteration matrix for P = I, i.e. no preconditioner
    norm_b = sqrt(maximum(eigvals( B' * B )))
end
```

However, when using a diagonal preconditioner `P = Diagonal(A)`, i.e. $\|\mathbf{B}\| = \|\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\|$, then

```
0.7236406064900974
```

```
let
    P = Diagonal(A)
    B = I - inv(P) * A  # Iteration matrix for P = Diagonal(A)
    norm_b = sqrt(maximum(eigvals( B' * B )))
end
```

Which is much smaller, in particular less than 1. Therefore only the preconditioned iterations converge:



```julia
let
    P = Diagonal(A)
    richardson_diagonal = richardson(A, b, P; tol=1e-10, maxiter=30)
    richardson_no_preconditioner = richardson(A, b, I; tol=1e-10, maxiter=30)

    plot(richardson_diagonal.relnorms;
         yaxis=:log, mark=:o, lw=2, ylabel=L"||r|| / ||b||",
         label=L"$P = \textrm{Diagonal(}A\textrm{)}$", legend=:bottomright,
         ylims=(1e-10, 1e5))
    plot!(richardson_no_preconditioner.relnorms;
          label=L"$P = I$", lw=2, mark=:o)
end
```

Choosing a good preconditioner

From the above experiments we notice:

Observation: Preconditioners

A good preconditioner P in the Richardson iterations, satisfies the following properties:

1. It is *cheap to invert*, that is linear systems $\mathbf{P}\mathbf{u} = \mathbf{r}$ are cheap to solve.
2. The iteration matrix norm $\|\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\|$ is as small as possible,

definitely smaller than one (to ensure convergence).

Clearly condition 2. suggests that the *perfect preconditioner* is $\mathbf{P} = \mathbf{A}$, such that $\mathbf{P}^{-1} = \mathbf{A}^{-1}$. In this setting $\|\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\| = \|\mathbf{I} - \mathbf{I}\| = 0$, i.e. we converge in a single Richardson step ! The trouble of this choice is that step 2 of Algorithm 1 (Richardson iteration) now effectively requires to solve the system $\mathbf{A}\mathbf{u}^{(k)} = \mathbf{r}^{(k)}$ for $\mathbf{u}^{(k)}$, i.e. we are back to solving the original problem.

On the other hand condition 1. suggests to use $\mathbf{P}^{-1} = \mathbf{I}^{-1}$ (since the identity is cheapest to invert –- nothing needs to be done). However, this does not really do anything and does thus not reduce the value of $\|\mathbf{B}\|$. It will just be $\|\mathbf{I} - \mathbf{A}\|$.

In practice we thus need to find a compromise between the two. As mentioned above standard choices for the preconditioner are:

- the diagonal of $\mathbf{A}$
- the lower triangle of $\mathbf{A}$
- another common choice is the *incomplete LU factorisation*, i.e. where one seeks a factorisation $\tilde{\mathbf{L}}\tilde{\mathbf{U}} \approx \mathbf{A}$ with a lower triangular matrix $\tilde{\mathbf{L}}$ and an upper triangular $\tilde{\mathbf{U}}$, which only *approximately* factorises $\mathbf{A}$. This gives additional freedom in designing the factorisations, in particular to avoid fill-in for sparse matrices.
- in many physics or engineering applications $\mathbf{A}$ results from a physical model. A preconditioner $\mathbf{P}$ can thus be resulting from an approximation to the employed physical model (e.g. by dropping the modelling of some physical effects).

We will not discuss the art of designing good preconditioners any further at this stage. Interested readers are referred to the excellent book Youssef Saad *Iterative methods for Sparse Linear Systems*, SIAM (2003).

Error control and stopping criterion

Let us return to clarifying the choice of the stopping criterion in the Richardson iterations (Algorithm 1).

As usual our goal is to control the error $\|\mathbf{x}_* - \mathbf{x}^{(k)}\|$ based on our stopping criterion, but without having access to the exact solution x_*. However, we know that $\mathbf{A}\mathbf{x}_* = \mathbf{b}$, since $\mathbf{x}_*$ is just the solution to the linear system we want to solve.

Similarly $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ directly implies

$$\mathbf{A}\mathbf{x}^{(k)} = \underbrace{\mathbf{b} - \mathbf{r}^{(k)}}_{=\widetilde{\mathbf{b}}}$$

where we defined the "perturbed" right-hand side $\widetilde{\mathbf{b}}$. Notably $\mathbf{x}^{(k)}$ is thus *exact* solution of a linear system involving $\mathbf{A}$ and this perturbed RHS, while we actually care to find the true solution $\mathbf{x}_*$ obtained by solving the system $\mathbf{A}\mathbf{x}_* = \mathbf{b}$ employing an "unperturbed" right-hand side $\mathbf{b}$.

We are thus in exactly the same setting as our final section on *Numerical stability* in our discussion on Direct methods for linear systems where instead of solving $\mathbf{A}\mathbf{x}_* = \mathbf{b}$ we are only able to solve the perturbed system $\mathbf{A}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}}$.

We can thus directly apply Theorem 2 from Direct methods for linear systems, which states that

$$\frac{\|\mathbf{x}_* - \widetilde{\mathbf{x}}\|}{\|\mathbf{x}_*\|} \leq \kappa(\mathbf{A}) \frac{\left\|\mathbf{b} - \widetilde{\mathbf{b}}\right\|}{\|\mathbf{b}\|}$$

Keeping in mind that here $\widetilde{\mathbf{x}} = \mathbf{x}^{(k)}$ and $\mathbf{b} - \widetilde{\mathbf{b}} = \mathbf{r}^{(k)}$ we thus obtain

$$\frac{\|\mathbf{x}_* - \mathbf{x}^{(k)}\|}{\|\mathbf{x}_*\|} \leq \kappa(\mathbf{A}) \frac{\left\|\mathbf{r}^{(k)}\right\|}{\|\mathbf{b}\|}$$

Combining this with our stopping criterion from **Algorithm 1**, that is

$$\frac{\left\|\mathbf{r}^{(k)}\right\|}{\|\mathbf{b}\|} < \varepsilon,$$

we finally obtain

$$\frac{\|\mathbf{x}_* - \mathbf{x}^{(k)}\|}{\|\mathbf{x}_*\|} < \kappa(\mathbf{A})\, \varepsilon.$$

In other words **our stopping criterion ensures** that the **relative error of the returned solution** is smaller than $\kappa(\mathbf{A})$ times the chosen tolerance.

If the matrix is well-conditioned, i.e. $\kappa(\mathbf{A})$ is close to 1, then the relative residual $\frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|}$ provides a good estimate of the true error and our stopping criterion is appropriate. However, if $\kappa(\mathbf{A})$ is large, even a small residual may imply a large error in the returned solution.

▶ **Alternative derivation without applying the previous Theorem 2**

Jacobi and Gauss-Seidel method

Note: We will only discuss the high-level ideas of this part in the lecture. You can expect that there will not be any detailed exam questions on Jacobi and Gauss-Seidel without providing you with the formulas and algorithms.

We will now briefly discuss the Jacobi and Gauss-Seidel methods, which can be seen as particular cases of Richardson iterations.

Recall equation (2)

$$\mathbf{P}\mathbf{x}^{(k+1)} = (\mathbf{P} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b}, \qquad k = 0, 1, \dots.$$

In the **Jacobi method** the key idea is to use the *diagonal* of the matrix $\mathbf{A}$ as the preconditioner

$$\mathbf{P} = \begin{pmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{nn} \end{pmatrix}.$$

and to explicitly insert this expression into equation (2). Rewriting we obtain in the $(k+1)$-st iteration of Jacobi's method

$$\begin{pmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{nn} \end{pmatrix} \begin{pmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{pmatrix} = - \begin{pmatrix} 0 & A_{12} & \cdots & A_{1n} \\ A_{21} & 0 & & \vdots \\ \vdots & & \ddots & \\ A_{n1} & \cdots & A_{n,n-1} & 0 \end{pmatrix} \begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

The diagonal structure of $\mathbf{P}$ allows the explicit computation of $\mathbf{x}^{(k+1)}$. Its i-th component can be obtained as

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij}\, x_j^{(k)} \right) \tag{7}$$

for all $i = 1, \dots, n$ and as long as $A_{ii} \neq 0$.

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

```julia
function jacobi(A, b; x=zero(b), tol=1e-6, maxiter=100)
    history  = [float(x)]  # History of iterates
    relnorms = Float64[]   # Track relative residual norm

    n  = length(x)
    xᵏ = x
    for k in 1:maxiter
        xᵏ⁺¹ = zeros(n)
        for i in 1:n
            Aᵢⱼxⱼ = 0.0
            for j in 1:n
                if i ≠ j
                    Aᵢⱼxⱼ += A[i, j] * xᵏ[j]
                end
            end  # Loop j
            xᵏ⁺¹[i] = (b[i] - Aᵢⱼxⱼ) / A[i, i]
        end  # Loop i

        relnorm_rᵏ = norm(b - A * xᵏ⁺¹) / norm(b)  # Relative residual norm
        push!(relnorms, relnorm_rᵏ)  # Push to history
        if relnorm_rᵏ < tol          # Check convergence
            break
        end

        xᵏ  = xᵏ⁺¹
        push!(history, xᵏ)
    end  # Loop k
    (; x=xᵏ, relnorms, history)
end
```

The **Gauss-Seidel method** employs the lower triangular part of A as the preconditioner $\mathbf{P}$ (in Julia `P = LowerTriangular(A)`):

$$\mathbf{P} = \begin{pmatrix} A_{11} \\ A_{21} & A_{22} \\ \vdots & & \ddots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix}.$$

By inserting the lower-triangular $\mathbf{P}$ into (2) we obtain in the $(k+1)$-st iteration of Gauss-Seidel:

$$\begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & & \ddots & \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{pmatrix} = - \begin{pmatrix} 0 & A_{12} & \cdots & A_{1n} \\ & 0 & \ddots & \vdots \\ & & \ddots & A_{n-1,n} \\ & & & 0 \end{pmatrix} \begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

Using forward substitution to solve this linear system leads to the following form for the i-th component of the vector $\mathbf{x}^{(k+1)}$.

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij}\, x_j^{(k)} \right) \tag{8}$$

for all $i = 1, \ldots, n$ and as long as $A_{ii} \neq 0$.

Notice that Gauss-Seidel is very similar to Jacobi's method, just with the small difference that for computing the new component $x_i^{(k+1)}$ we use the components $x_j^{(k+1)}$ for $j < i$, which have already been updated to their new values.

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

```julia
function gauss_seidel(A, b; x=zero(b), tol=1e-6, maxiter=100)
    history  = [float(x)]  # History of iterates
    relnorms = Float64[]   # Track relative residual norm

    n  = length(x)
    xᵏ = x
    for k in 1:maxiter
        xᵏ⁺¹ = zeros(n)
        for i in 1:n
            Aᵢⱼxⱼ = 0.0
            for j in 1:i-1
                Aᵢⱼxⱼ += A[i, j] * xᵏ⁺¹[j]
            end  # Loop j
            for j in i+1:n
                Aᵢⱼxⱼ += A[i, j] * xᵏ[j]
            end  # Loop j
            xᵏ⁺¹[i] = (b[i] - Aᵢⱼxⱼ) / A[i, i]
        end  # Loop i

        relnorm_rᵏ = norm(b - A * xᵏ⁺¹) / norm(b)  # Relative residual norm
        push!(relnorms, relnorm_rᵏ)  # Push to history
        if relnorm_rᵏ < tol          # Check convergence
            break
        end

        xᵏ = xᵏ⁺¹
        push!(history, xᵏ)
    end  # Loop k
    (; x=xᵏ, relnorms, history)
end
```

These two cases just provide two examples of the many flavours of Richardson iterations, which are used in practice. A good overview provides chapter 4 of Youssef Saad *Iterative methods for Sparse Linear Systems*, SIAM (2003).

We return to our example problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ with

```
100×100 Matrix{Float64}:
 34.8048      0.698064   0.0912903   0.110129  …  -0.409052    0.697422  -1.04543
  0.666056   41.1921     0.510527   -2.32168       1.10391    -0.526851  -0.08354
  0.474286   -1.39428   41.8852      0.150974     -0.688416    0.915631   0.46982
 -0.473043    1.05738   -1.05232    29.5658       -2.21622    -0.771756  -1.45717
 -1.24018     1.46078    0.36293    -0.129446     -0.164112   -0.428056  -1.24723
  0.0361345  -0.347109   0.465493    0.30022   …  -0.0134317  -0.766304   0.394693
  1.4133      0.545532   0.208997   -1.36675      -1.76489    -1.31785    0.870358
  ⋮                                            ⋱
  0.655297   -0.266465  -0.609266   -0.657441     -0.49885     0.592234   1.18366
 -0.398953   -0.468695   0.416494    1.38779   …  -1.38401     0.108533   1.41024
 -1.23063    -0.396662   1.63005    -0.927915      0.12108     1.16041   -0.392143
  0.369631   -1.42732   -0.243615    1.44618      34.2564     -0.452845  -0.0258712
  0.684164   -0.162484  -1.09723    -0.61984       0.131653   30.7371    -1.20176
  0.648161    0.492944  -0.440426   -0.193745      0.109915   -1.12334   33.8566
```

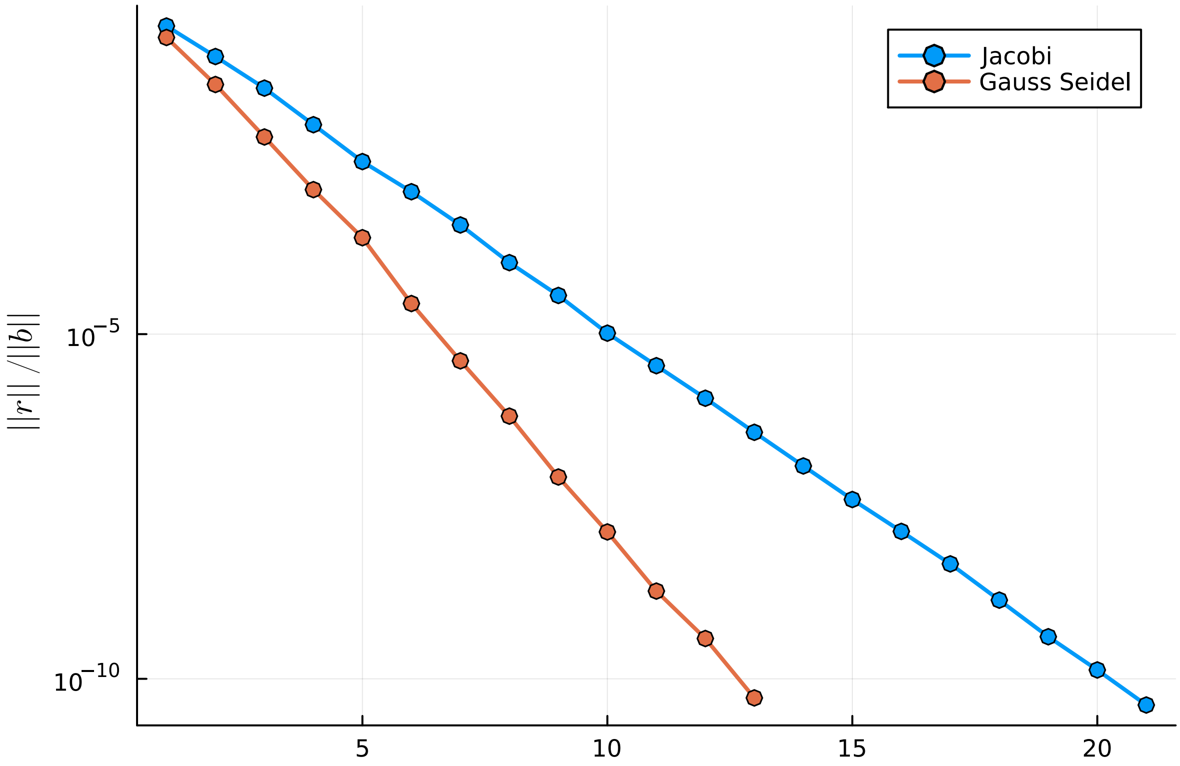
```
A
```

and

```
▸ [0.432575, 0.190922, 0.0125673, 0.294348, 0.770749, 0.678093, 0.906543, 0.140135, 0.15278
```

```
b
```

On this problem the convergence is as follows:



```
let
    result_jacobi       = jacobi(A, b; tol=1e-10, maxiter=30)
    result_gauss_seidel = gauss_seidel(A, b; tol=1e-10, maxiter=30)


    P = Diagonal(A)
    richardson_diagonal = richardson(A, b, P; tol=1e-6, maxiter=30)
    richardson_no_preconditioner = richardson(A, b, I; tol=1e-6, maxiter=30)

    plot(result_jacobi.relnorms;
         yaxis=:log, mark=:o, lw=2, ylabel=L"||r|| / ||b||",
         label="Jacobi", legend=:topright)
    plot!(result_gauss_seidel.relnorms;
          label="Gauss Seidel", lw=2, mark=:o)
end
```

Linear systems involving symmetric positive-definite matrices

In many applications in engineering and the sciences the arising system matrices $\mathbf{A}$ have additional properties, which can be exploited to obtain better-suited algorithms. An important case are the symmetric positive definite matrices.

This part of the notes only provides a condensed introduction. A more detailed, but very accessible introduction to the steepest descent and conjugate gradient methods discussed in this section provides Jonathan Shewchuk *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain* (1994).

Let us first recall the definition of symmetric positive definite matrices.

Definition: Positive definite

A matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is called **positive definite** if

$$\forall\, \mathbf{0} \neq \mathbf{v} \in \mathbb{R}^n \qquad \mathbf{v}^T \mathbf{A} \mathbf{v} > 0$$

For **symmetric matrices** we additionally have that $\mathbf{A}^T = \mathbf{A}$. Symmetric positive definite matrices (often abbreviated as s.p.d. matrices) arise naturally in physical systems when looking for configurations minimising their energy. A result underlining this construction is

Theorem 2

An matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is s.p.d. if and only if all its eigenvalues are real and positive.

To every linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ involving an s.p.d. system matrix $\mathbf{A}$ we can associate an **energy function** ϕ

$$\phi : \mathbb{R}^n \to \mathbb{R} : \phi(\mathbf{v}) = \frac{1}{2} \mathbf{v}^T \mathbf{A} \mathbf{v} - \mathbf{v}^T \mathbf{b}$$

Using basic vector calculus one shows that

$$\begin{array}{rll} \text{Gradient} & \nabla\phi(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = -\mathbf{r} & \textit{(negative residual)} \\ \text{Hessian} & H_\phi(\mathbf{x}) = \mathbf{A} & \textit{(system matrix)} \end{array} \tag{9}$$

Setting the gradient to zero, we obtain the stationary points. The corresponding equation

$$\mathbf{0} = \nabla\phi(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$$

only has a single solution, provided that $\mathbf{A}$ is non-singular. Since the Hessian $H_\phi(\mathbf{x}) = \mathbf{A}$ is positive definite, this stationary point is a minimum. As a result we obtain

Proposition 3

Given an s.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, the solution of the system $\mathbf{A}\mathbf{x}_* = \mathbf{b}$ is the unique minimum of the function ϕ, i.e.

$$\mathbf{x}_* = \operatorname{argmin}_{\mathbf{v}\in\mathbb{R}^n} \phi(\mathbf{v}). \qquad (10)$$

Importantly there is thus a **relation between optimisation problems** and **solving linear systems** if the system matrix $\mathbf{A}$ is s.p.d.

SPD matrices are not unusual. For example, recall that in polynomial regression problems (see least-squares problems in Interpolation), where we wanted to find the best polynomial through the points (x_i, y_i) for $i = 1, \ldots n$ by minimising the least-squares error, we had to solve the *normal equations*

$$\mathbf{V}^T\mathbf{V}\mathbf{c} = \mathbf{V}^T\mathbf{y}$$

where $\mathbf{c} \in \mathbf{R}^m$ are the unknown coefficients of the polynomial $\sum_{j=0}^m c_j x^j$, $\mathbf{y}$ is the vector collecting all y_i values and the $\mathbf{V}$ is the Vandermonde matrix

$$\mathbf{V} = \begin{pmatrix} 1 & x_1 & \ldots & x_1^m \\ 1 & x_2 & \ldots & x_2^m \\ \vdots \\ 1 & x_n & \ldots & x_n^m \end{pmatrix} \in \mathbb{R}^{n\times(m+1)}.$$

In this case the system matrix $\mathbf{A} = \mathbf{V}^T\mathbf{V}$ is *always* s.p.d.

Steepest descent method

If we view the problem of solving linear systems as an optimisation problem, a relatively simple idea is to construct an iterative method, where at every step k we try to decrease the energy function ϕ.

To guide our thoughts we consider a 2D problem which is easy to visualise. We take as an example

$$\mathbf{A}^{2\mathrm{D}} = \begin{pmatrix} 2 & 0 \\ 0 & 60 \end{pmatrix} \qquad \mathbf{b}^{2\mathrm{D}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

such that

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}^{2\mathrm{D}}\mathbf{x} - \mathbf{x}^T\mathbf{b}^{2\mathrm{D}} = x_1^2 + 30x_2^2$$

```
A2d = 2×2 Matrix{Float64}:
       1.0   0.0
       0.0  20.0
```

```
A2d = [1.0  0.0;
       0.0 20.0]
```

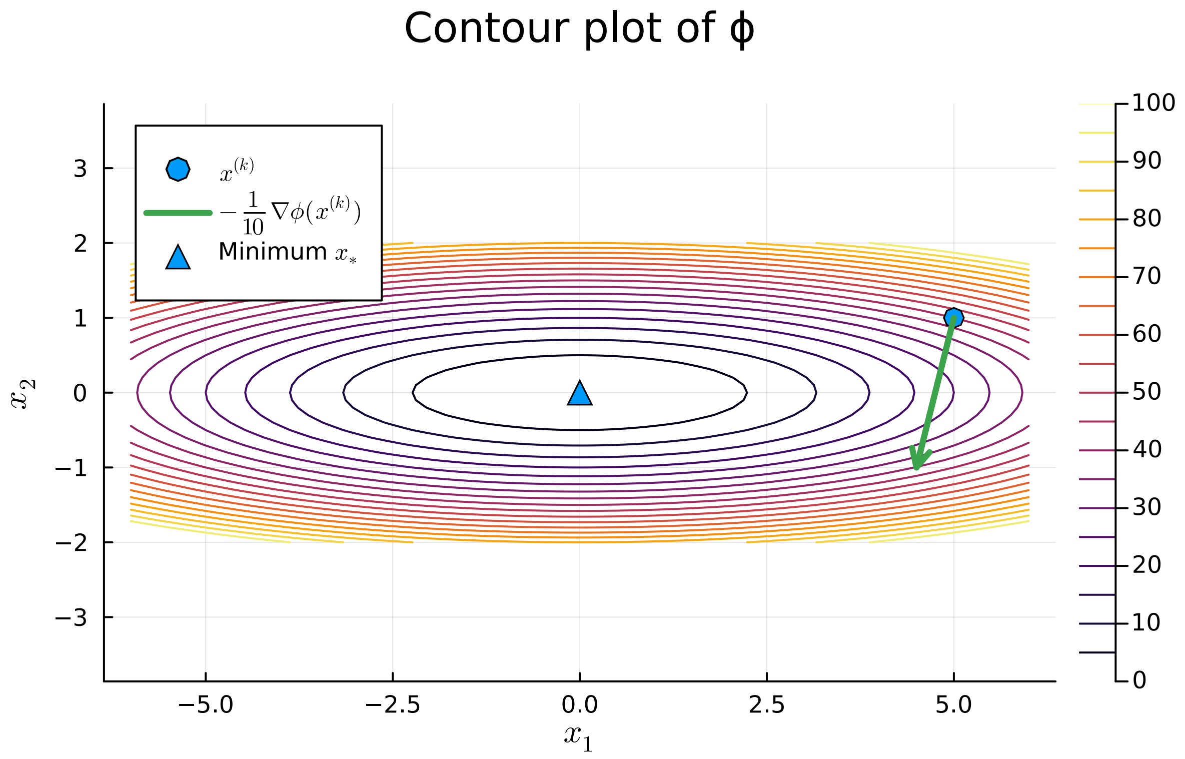
```
b2d = ▸[0.0, 0.0]
```

```
b2d = [0.0;
       0.0]
```

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

```
ϕ(x₁, x₂) = x₁^2 + 20 * x₂^2
```

This problem is visualised as a contour plot below:



Now suppose we have an estimate $\mathbf{x}^{(k)}$ of the solutin of $\mathbf{A}\mathbf{x} = \mathbf{b}$ shown above by the blue dot. This $\mathbf{x}^{(k)}$ is also an estimate of the minimum of ϕ. Our goal is thus to find a $\mathbf{x}^{(k+1)}$ satisfying $\phi(\mathbf{x}^{(k+1)}) < \phi(\mathbf{x}^{(k)})$, i.e. to get closer to the minimum (blue triangle).

The gradient $\nabla\phi(\mathbf{x}^{(k)})$ provides the slope of the function ϕ at $\mathbf{x}^{(k)}$ in the *upwards* direction. Therefore, taking a step along the direction of $-\nabla\phi$ takes us downhill (see green arrow). We thus propose an algorithm

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \alpha_k \nabla\phi(\mathbf{x}^{(k)}) \\ &= \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)},\end{aligned} \tag{11}$$

where we used that $\nabla\phi(\mathbf{x}^{(k)}) = -\mathbf{r}^{(k)}$, and where $\alpha_k > 0$ is a parameter determining how far to follow along the direction of the negative gradient.

Note, that this method is quite related to Richardson's iterations: for $\alpha_k = 1$ we actually recover **Algorithm 1** with $\mathbf{P} = \mathbf{I}$.

▶ **Rationale for introducing αₖ. Or: why not just take αₖ = 1 ?**

Since overall our goal is to find the minimum of ϕ, a natural idea to determine α_k exactly such that we make the value of $\phi(\mathbf{x}^{(k+1)})$ as small as possible. We compute

$$\begin{aligned}\phi(\mathbf{x}^{(k+1)}) &= \phi(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}) \\ &= \frac{1}{2}\left(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}\right)^T \mathbf{A} \left(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}\right) - \left(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}\right)^T \mathbf{b} \\ &= \underbrace{\frac{1}{2}(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)}}_{=a} \alpha_k^2 + \underbrace{\left[\frac{1}{2}(\mathbf{x}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)} + \frac{1}{2}(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{x}^{(k)} - (\mathbf{r}^{(k)})^T \mathbf{b}\right]}_{=b} \alpha_k^{(k)} \\ &\qquad + \underbrace{\left[\frac{1}{2}(\mathbf{x}^{(k)})^T \mathbf{A}\mathbf{x}^{(k)} - (\mathbf{x}^{(k)})^T \mathbf{b}\right]}_{=c}\end{aligned}$$

where

$$a = \frac{1}{2}(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)} \qquad c = \frac{1}{2}(\mathbf{x}^{(k)})^T \mathbf{A}\mathbf{x}^{(k)} - (\mathbf{x}^{(k)})^T \mathbf{b}$$

and

$$\begin{aligned}b &= \frac{1}{2}(\mathbf{x}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)} + \frac{1}{2}(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{x}^{(k)} - (\mathbf{r}^{(k)})^T \mathbf{b} \\ &= (\mathbf{x}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)} - (\mathbf{r}^{(k)})^T \mathbf{b} \\ &= -(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}\end{aligned}$$

We notice that $\phi(\mathbf{x}^{(k+1)}) = a\alpha_k^2 + b\alpha_k + c$ is a second-degree polynomial in α_k. Setting its gradient to zero gives us the α_k which minimises the ϕ as much as possible in this step. From

$$0 = \frac{d\phi}{d\alpha_k}(\mathbf{x}^{(k+1)}) = 2a\alpha_k + b$$

we obtain the optimal step size as

$$\alpha_k = \frac{-b}{2a} = \frac{(\mathbf{r}^{(k)})^T\,\mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T\mathbf{A}\,\mathbf{r}^{(k)}}. \tag{12}$$

In summary we obtain the algorithm:

Algorithm 2: Steepest descent method

Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be an s.p.d. system matrix, right-hand side $\mathbf{b} \in \mathbb{R}^n$, an initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and convergence threshold ε.

Compute the inital residual $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$. Then for $k = 0, 1, \dots$ iterate:

1. Compute $\mathbf{w}^{(k)} = \mathbf{A}\mathbf{r}^{(k)}$
2. Step size: $\alpha_k = \frac{(\mathbf{r}^{(k)})^T\,\mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T\mathbf{w}^{(k)}}$ *(Optimal because of (12))*
3. Take step: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{r}^{(k)}$.
4. Update residual $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k\mathbf{w}^{(k)}$

The iteration is stopped as soon as $\frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} < \varepsilon$.

Notice that in this algorithm we used the trick

$$\mathbf{r}^{(k+1)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} = \mathbf{b} - \mathbf{A}\left(\mathbf{x}^{(k)} + \alpha_k\mathbf{r}^{(k)}\right) = \mathbf{r}^{(k)} - \alpha_k\mathbf{w}^{(k)}$$

to compute the residual $\mathbf{r}^{(k+1)}$ from $\mathbf{r}^{(k)}$ and $\mathbf{w}^{(k)}$, that is without computing another matrix-vector product. As matrix-vector products scale as $O(n^2)$, but vector operations only as $O(n)$ this saves computational cost.

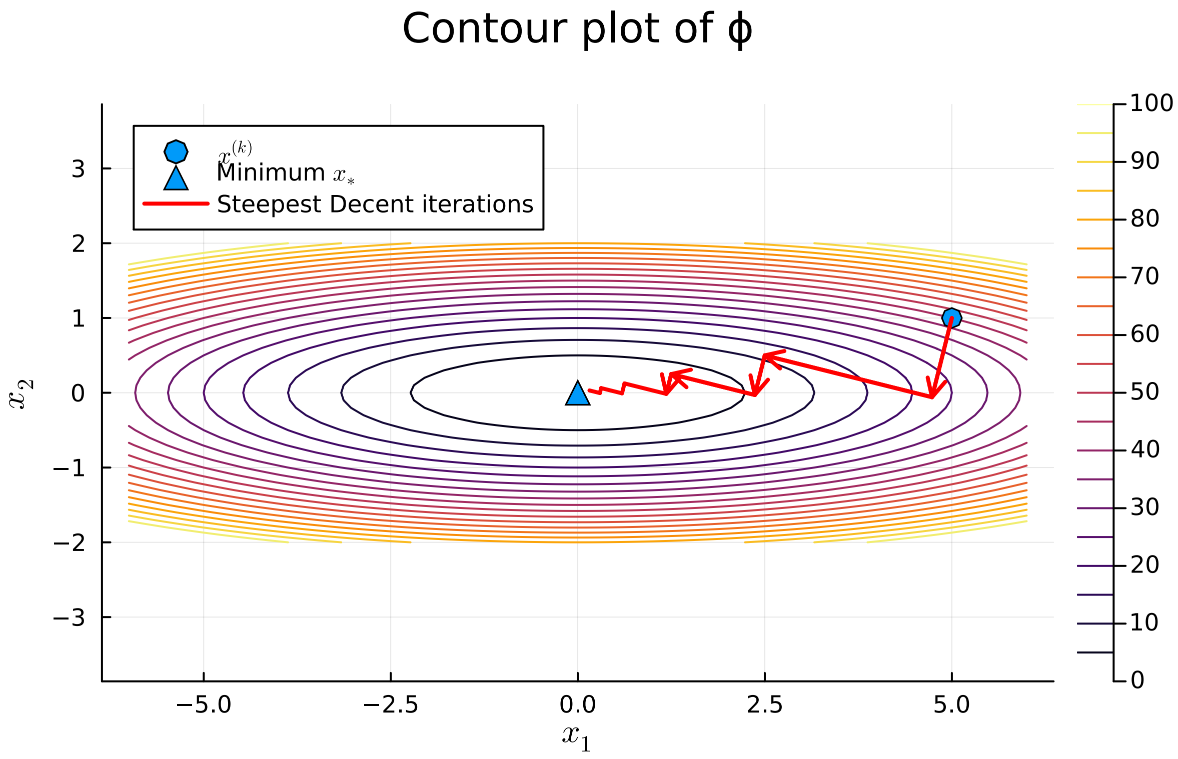
An implementation of this algorithm is given by the following function:

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

```julia
function steepest_descent_simple(A, b; x=zero(b), tol=1e-6, maxiter=100)
    history  = [float(x)]  # History of iterates
    relnorms = Float64[]   # Track relative residual norm

    x = x
    r = b - A * x
    for k in 1:maxiter
        relnorm = norm(r) / norm(b)
        push!(relnorms, relnorm)
        if relnorm < tol
            break
        end

        Ar = A * r
        α = dot(r, r) / dot(r, Ar)
        x = x + α * r
        r = r - α * Ar

        push!(history, x)
    end
    (; x, history, relnorms)
end
```

Running it on our 2x2 example problem plotted above it produces the following steps (first 6 steps shown).

We realise that convergence is steadily towards the minimum, but seems to oscillate around the "best possible path". We will see in a second why this is.



In line with our previous discussion we can again view **Algorithm 2** as a fixed-point method $\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)})$, this time with fixed-point function

$$g(\mathbf{x}) = \mathbf{x} + \alpha(\mathbf{x})\ (\mathbf{b} - \mathbf{A}\mathbf{x})$$

where $\alpha(\mathbf{x})$ is meant to indicate the step size determined according to equation (12). Clearly a fixed-point of this function satisfies $\mathbf{x} = \mathbf{x} + \alpha(\mathbf{x})\ (\mathbf{b} - \mathbf{A}\mathbf{x})$ and thus $\mathbf{b} = \mathbf{A}\mathbf{x}$.

While the details are more involved and beyond our scope, applying the usual convergence theory on fixed-point methods yields the following observation:

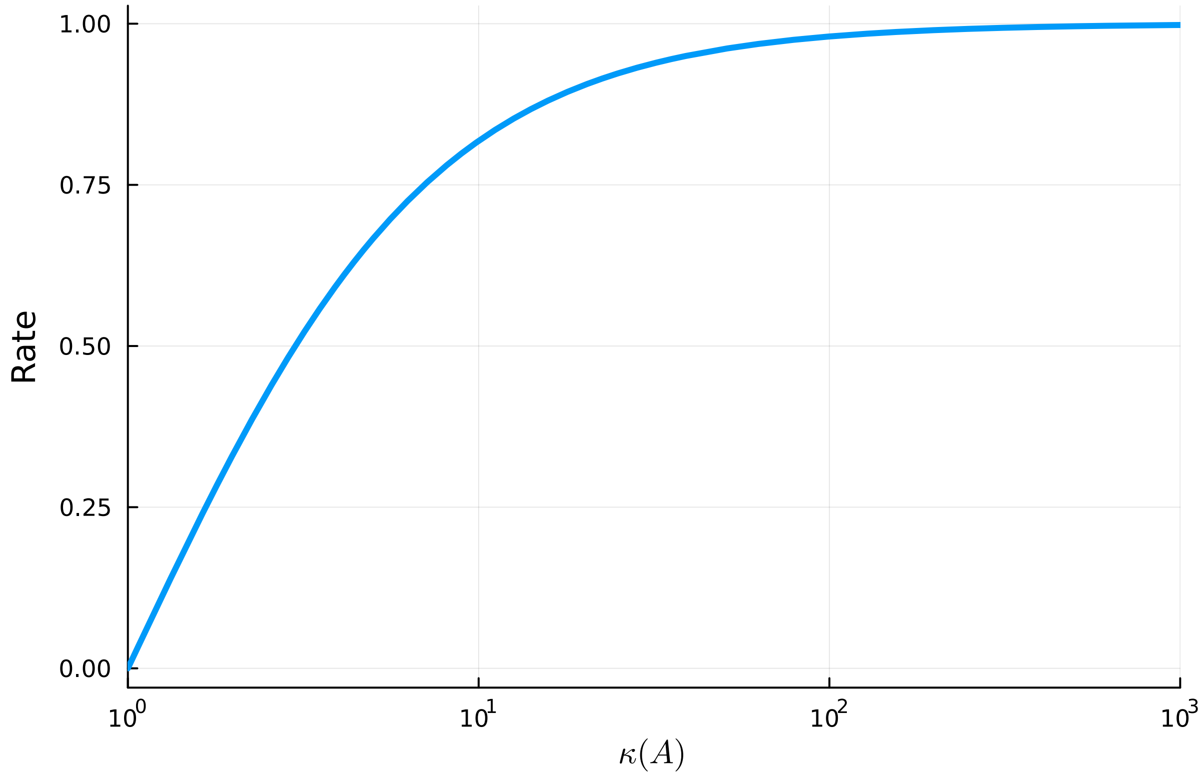
Theorem 4: Convergence of steepest descent

Given an s.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and a right-hand side $\mathbf{b}$ the solution of the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be obtained by applying steepest descent starting from *any* initial position $\mathbf{x}^{(0)}$. The following error estimate holds:

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C\left(\frac{\kappa(A) - 1}{\kappa(A) + 1}\right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\| \tag{13}$$

where C is a positive constant. This is **linear convergence** with rate $\frac{\kappa(A)-1}{\kappa(A)+1}$.

The main result is thus that steepest descent **always converges**. However, if $\kappa(A)$ is large the convergence can be slow. Here is a plot of the rate with increasing κ. Recall, that smaller rate means faster convergence and a rate of 1 is essentially stagnation.



We try steepest descent on the following random SPD matrix.

```
100×100 Matrix{Float64}:
  9.04051   -1.11409      0.230077   …  -0.312022    0.280083   -1.00696
 -1.11409    8.46927      0.128274      -0.512461    0.338626    0.732093
  0.230077   0.128274     9.54354       -0.476581    0.302192   -0.343011
  0.491476  -0.398399    -0.0609387     -0.849568   -0.253592   -0.767952
 -0.458634   0.58881     -0.115621      -0.0697998  -0.499045    0.601634
 -0.878459  -1.37659      0.095341   …   0.20345     0.292237   -0.0870487
  0.208997   0.326429    -0.467092      -0.293393    0.0677063  -0.177346
  ⋮                                  ⋱
 -0.156691  -0.382566    -0.894843       0.801477   -0.477816   -0.395234
 -0.292745   0.00408288   0.432793   …  -0.589568   -0.185808   -0.0565833
 -0.641162   0.81985      0.517527       1.30233    -0.398196   -0.110799
 -0.312022  -0.512461    -0.476581       8.76217    -0.0783765   0.0480122
  0.280083   0.338626     0.302192      -0.0783765   7.86149     0.0241475
 -1.00696    0.732093    -0.343011       0.0480122   0.0241475   8.3315
```

```
begin
    # Generate a random s.p.d. matrix
    U, Σ, V = svd(randn(100, 100))
    Aₛ = U * Diagonal(abs.(Σ)) * U'
end
```

```julia
bₛ = ones(100);
```

The system matrix has a condition number larger well larger than 1:

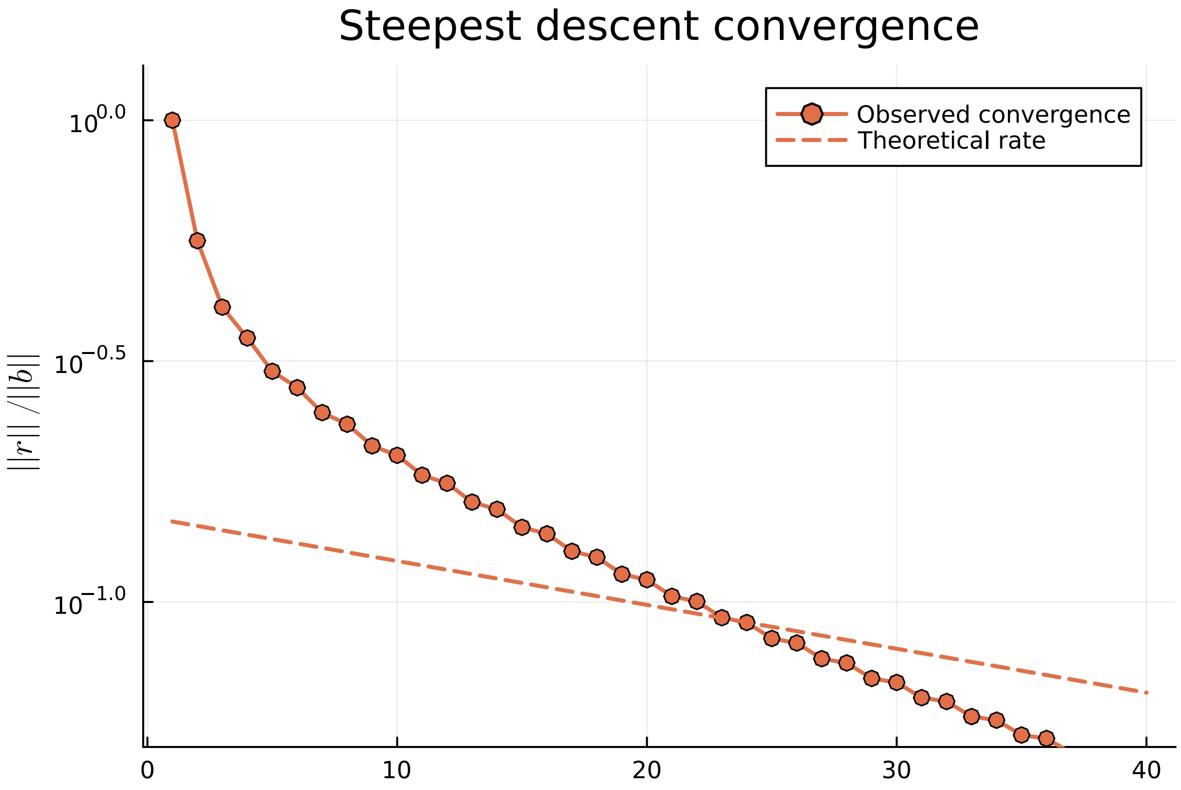
```
95.37546819057002
```

```julia
cond(Aₛ)
```

Still, as the theory predicts, the method converges. However, convergence gets slow after a while and approaches the theoretical rate

```
rate_steep_desc = 0.9792478310347063
```

```julia
rate_steep_desc = (cond(Aₛ) - 1) / (cond(Aₛ) + 1)
```



```julia
let
    descent = steepest_descent_simple(Aₛ, bₛ; tol=1e-6, maxiter=40)

    plot(descent.relnorms;
        yaxis=:log, mark=:o, lw=2, ylabel=L"||r|| / ||b||", ylims=(5e-2, 1.3),
        title="Steepest descent convergence", label="Observed convergence", c=2)

    plot!(1:40, x -> 0.15rate_steep_desc^x, lw=2, ls=:dash, label="Theoretical
rate", c=2)
end
```

Optional: Preconditioned steepest descent

We noticed above that a condition number $\kappa(\mathbf{A}) \approx 1$ is providing the highest convergence rate for steepest descent. For cases where the condition number is large, a potential cure is to apply *preconditioning*. The idea is similar to Richardson iteration. Assume we have a matrix $\mathbf{P} \approx \mathbf{A}$. Then instead of applying steepest descent to $\mathbf{A}\mathbf{x} = \mathbf{b}$ we instead apply it to

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{x} = \mathbf{P}^{-1}\mathbf{b},$$

such that this new method will now converge as

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\kappa(\mathbf{P}^{-1}\mathbf{A}) - 1}{\kappa(\mathbf{P}^{-1}\mathbf{A}) + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\|$$

If we use a good preconditioner, then $\kappa(\mathbf{P}^{-1}\mathbf{A}) \ll \kappa(\mathbf{A})$ and convergence accelerates.

Without going into many details, we remark that there are a few subtleties that apply:

- Even if $\mathbf{P}$ *and* $\mathbf{A}$ are s.p.d. matrices, the matrix $\mathbf{P}^{-1}\mathbf{A}$ *may not be s.p.d.*. Therefore a practical implementation of preconditioned gradient descent is not realised by blindly applying the plain algorithm to $\mathbf{P}^{-1}\mathbf{A}$, but instead by "inlining" the application of $\mathbf{P}^{-1}$ into the actual algorithm. At the k-th iteration we thus want to update the solution $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$ using the *preconditioned residual*

  $$\mathbf{z}^{(k)} = \mathbf{P}^{-1}\mathbf{b} - \mathbf{P}^{-1}\mathbf{A}\mathbf{x}^{(k)} = \mathbf{P}^{-1}\mathbf{r}^{(k)}.$$

  Following through with this change also in the computation of the optimal step size leads to $\alpha_k = \frac{(\mathbf{z}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T \mathbf{A} \mathbf{z}^{(k)}}$, thus the following algorithm:

Algorithm 3: Preconditioned steepest descent method

Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be an s.p.d. system matrix, right-hand side $\mathbf{b} \in \mathbb{R}^n$, preconditioner $\mathbf{P} \in \mathbb{R}^{n \times n}$, an initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$ and convergence threshold ε.

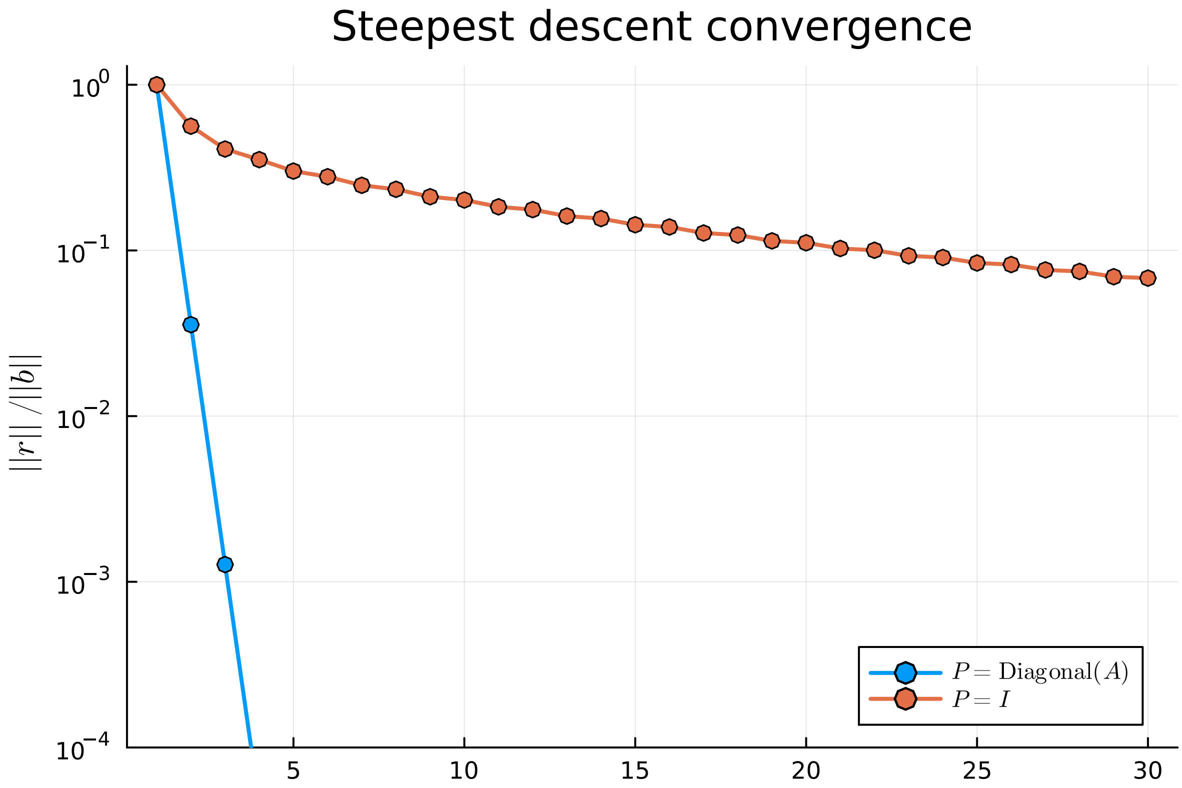
Compute the inital residual $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$. Then for $k = 0, 1, \ldots$ iterate:

1. Solve $\mathbf{P}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ for $\mathbf{z}^{(k)}$ *(This is the new step)*
2. Compute $\mathbf{w}^{(k)} = \mathbf{A}\mathbf{z}^{(k)}$ *(This has been changed)*
3. Step size: $\alpha_k = \frac{(\mathbf{z}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T \mathbf{w}^{(k)}}$
4. Take step: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$.
5. Update residual $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{w}^{(k)}$

The iteration is stopped as soon as $\frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} < \varepsilon$.

- Similar to our discussion in the context of Richardson iterations, the choice of the preconditioner is delicate. The "perfect" preconditioner $\mathbf{P} = \mathbf{A}$ brings our condition number down to $\kappa(\mathbf{A}^{-1}\mathbf{A}) = \kappa(\mathbf{I}) = 1$, but requires us to solve the full problem in the first step of **Algorithm 3**. On the other hand no preconditioning, i.e. $\mathbf{P} = \mathbf{I}$, gives a cheap computation of $\mathbf{z}^{(k)}$ from $\mathbf{P}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$, but does nothing to reduce the conditioning. Often a simple preconditioner, such as the diagonal of the matrix already provides noteworthy improvements. For example:

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



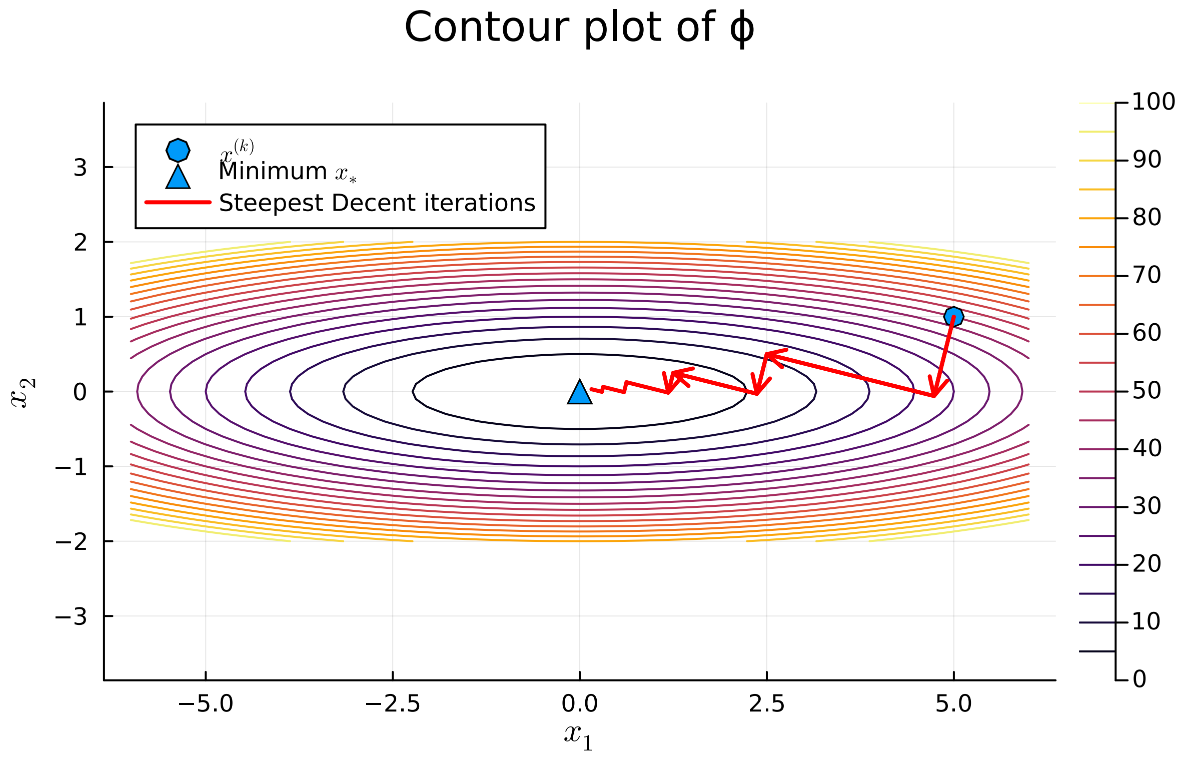
Conjugate gradient method

Recall that the key idea of steepest descent was to employ the update

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha_k \nabla\phi(\mathbf{x}^{(k)}) = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$$

that is to follow in the *direction of maximal descent* — i.e. along the direction of the negative gradient of ϕ.

While this is certainly a natural choice, we also saw that this can lead to a rather unsteady convergence behaviour:



One way to cure this behaviour is in fact make a rather different choice for the update direction. While for the first update we keep $\mathbf{p}^{(1)} = \mathbf{r}^{(k)}$ we subsequently choose directions $\mathbf{p}^{(k)}$ with the property

$$0 = \mathbf{p}^{(k)}\mathbf{A}\mathbf{p}^{(j)} \qquad \text{with } j = 0, 1, \ldots, k-1.$$

This property of the vectors $\mathbf{p}^{(k)}$ is usually called $\mathbf{A}$-orthogonality. Based on this update direction we iterate as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$$

with the optimal step size now being given by

$$\alpha_k = \frac{(\mathbf{p}^{(k)})^T\,\mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T\mathbf{A}\mathbf{p}^{(k)}}.$$

The next $\mathbf{p}^{(k+1)}$ is found by A-orthogonalising $\mathbf{r}^{(k+1)}$ against all previous $\mathbf{p}^j$ with $j = 1, \ldots, k$. Perhaps surprisingly this can be achieved by the following recurrent algorithm

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k\, \mathbf{p}^k \qquad \text{with} \quad \beta_k = \frac{(\mathbf{A}\mathbf{p}^{(k)})^T\, \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T \mathbf{A}\mathbf{p}^{(k)}}.$$

This is called the Conjugate gradient method and despite its invention in 1952 is still the state-of-the-art method for solving linear systems or optimisation problems involving s.p.d. matrices.

Studying its convergence leads to the following strong result:

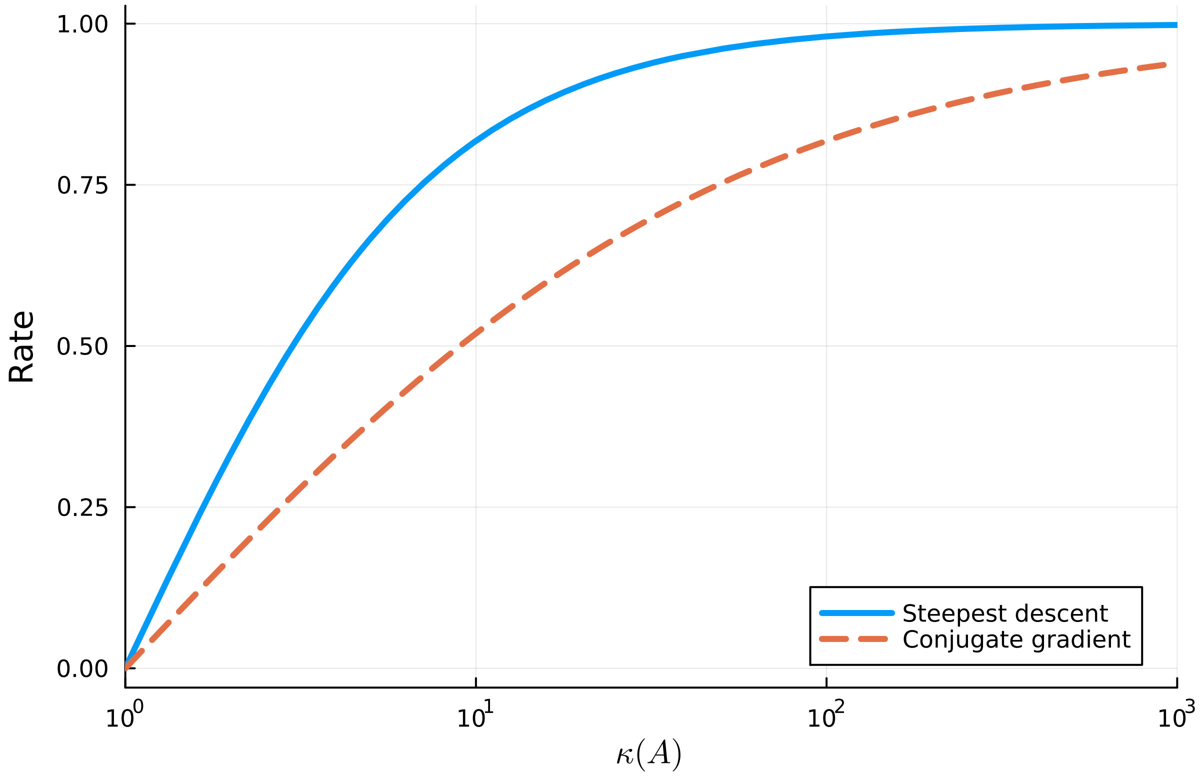
Theorem 4: Convergence of Conjugate Gradient (CG)

Given an s.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and a right-hand side $\mathbf{b}$ the solution of the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be obtained by applying the conjugate gradient algorithm starting from *any* initial position $\mathbf{x}^{(0)}$ **in at most n steps** in exact arithmetic. Moreover the following error estimate holds:

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\| \tag{14}$$

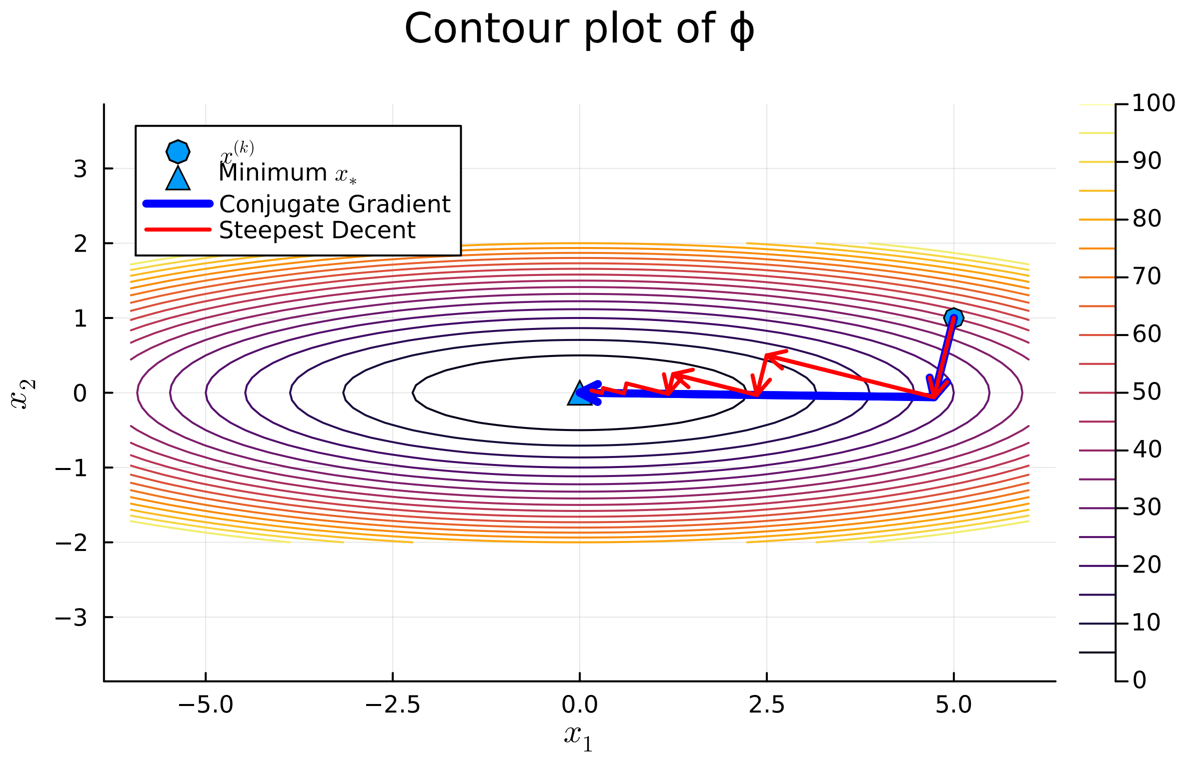
where C is a positive constant. This is **linear convergence** with rate $\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$.

Notice (1) that this method is **guaranteed to converge after n steps** and (2) that the rate deteriorates much slower as the condition number increases (recall that smaller rates are better):



```julia
let
    p = plot(κ -> (κ-1)/(κ+1), xaxis=:log, xlims=(1, 10^3), label="Steepest descent", xlabel=L"κ(A)", ylabel="Rate", lw=3)
    plot!(p, κ -> (sqrt(κ)-1)/(sqrt(κ)+1), lw=3, label="Conjugate gradient", ls=:dash)
end
```

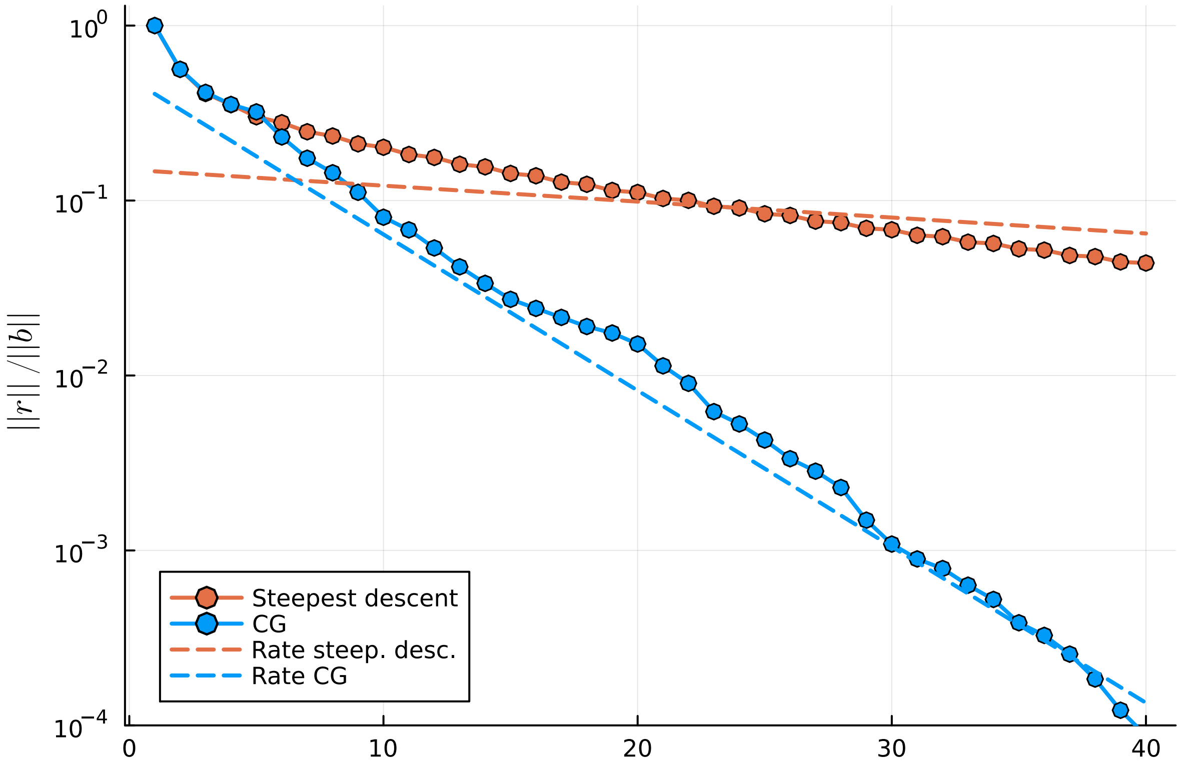
Indeed for our 2x2 matrix example two steps of CG are enough:



Furthermore we note our random matrix example to converge noticably faster:

```
rate_cg = 0.8142306124513649
```

```
rate_cg = (sqrt(cond(Aₛ)) - 1) / (sqrt(cond(Aₛ)) + 1)
```



A (non-optimised) CG implementation is:

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

```julia
function conjugate_gradient_simple(A, b; x=zero(b), tol=1e-6, maxiter=100)
    history  = [float(x)]  # History of iterates
    relnorms = Float64[]   # Track relative residual norm

    r = b - A * x
    p = r
    for k in 1:maxiter
        relnorm = norm(r) / norm(b)
        push!(relnorms, relnorm)
        if relnorm < tol
            break
        end

        # Descent along conjugate direction p (instead of along r)
        Ap  = A * p
        α   = dot(r, r) / dot(p, Ap)
        x   = x + α * p
        r_new = r - α * Ap

        # Update conjugate direction p
        β = dot(r_new, r_new) / dot(r, r)
        p = r_new + β * p
        r = r_new

        push!(history, x)
    end
    (; x, relnorms, history)
end
```

Numerical analysis