Error control in scientific modelling (MATH 500, Herbst)

Project 1: Numerical investigation of quantum tunnelling

To be handed in via moodle by 01.12.2023

```
begin
    using BenchmarkTools
    using DFTK
    using DoubleFloats
    using LinearAlgebra
    using LinearMaps
    using IntervalArithmetic
    using GenericLinearAlgebra
    using MatrixDepot
    using Printf
    using Plots
    using PlutoTeachingTools
    using PlutoUI
    using TimerOutputs
end
```

```
Code paths for generic floating-point types activated in DFTK. Remember to add
'using GenericLinearAlgebra' to your user script. See https://docs.dftk.org/sta
ble/examples/arbitrary_floattype/ for details.

verify download of index files...

reading database

adding metadata...

adding svd data...

writing database

used remote sites are sparse.tamu.edu with MAT index and math.nist.gov with HTM
L index
```

Table of Contents

Introduction and physical setting

The goal of this project is to develop an efficient numerical approach to investigate the effect of *quantum tunnelling* in a system where bosonic particles interact with an infinite 1D chain of identical atoms. Our goal is to control both algorithm error as well as the arithmetic error in the description of tunnelling.

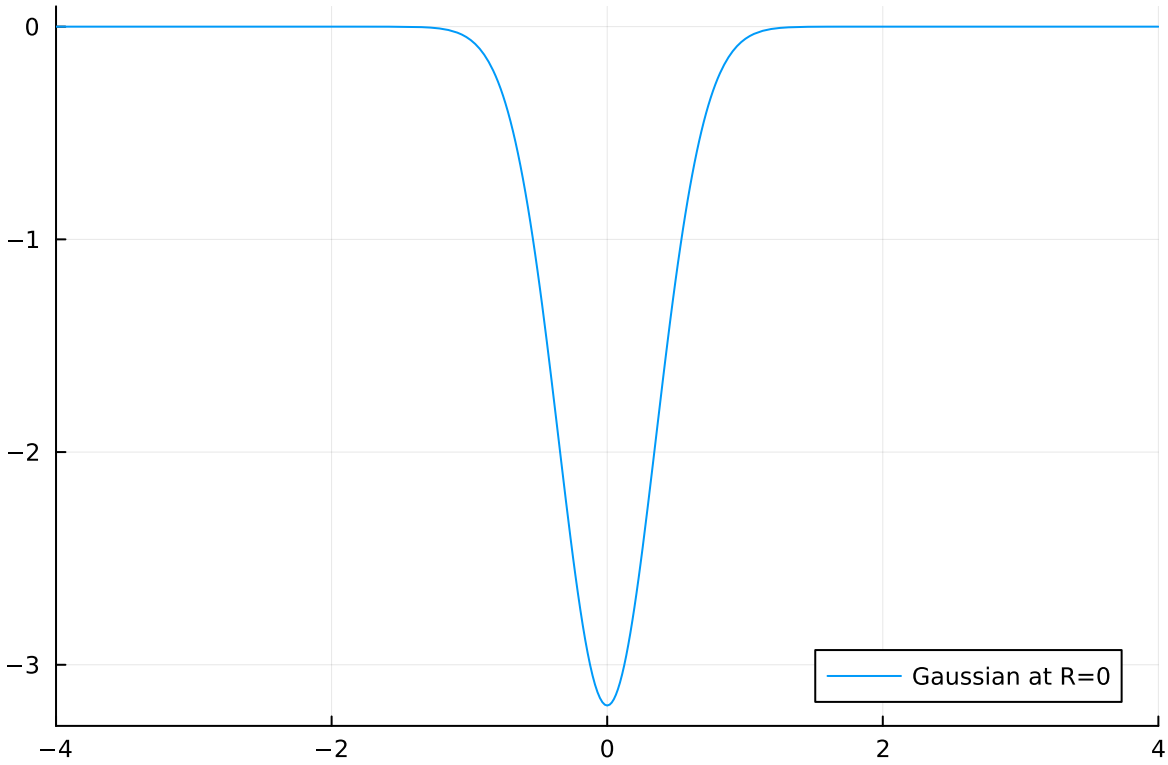
From the physical perspective we consider a system of N bosonic particles, which interact with a chain of M atoms. First we take M finite and small, then we will make it larger and larger. To simplify our life we consider the bosons to be non-interacting with each other and we take the interaction of a boson located at x and an atom located at R to be Gaussian, i.e.

$$v^{\text{atom}}(x-R) = -\frac{\alpha}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-R)^2}{4\sigma^2}\right)$$

with $\sigma, \alpha > 0$ given constants and $R \in \mathbb{R}$. Pictorially for $\alpha = 2$ and $\sigma = \frac{1}{4}$, this potential looks like

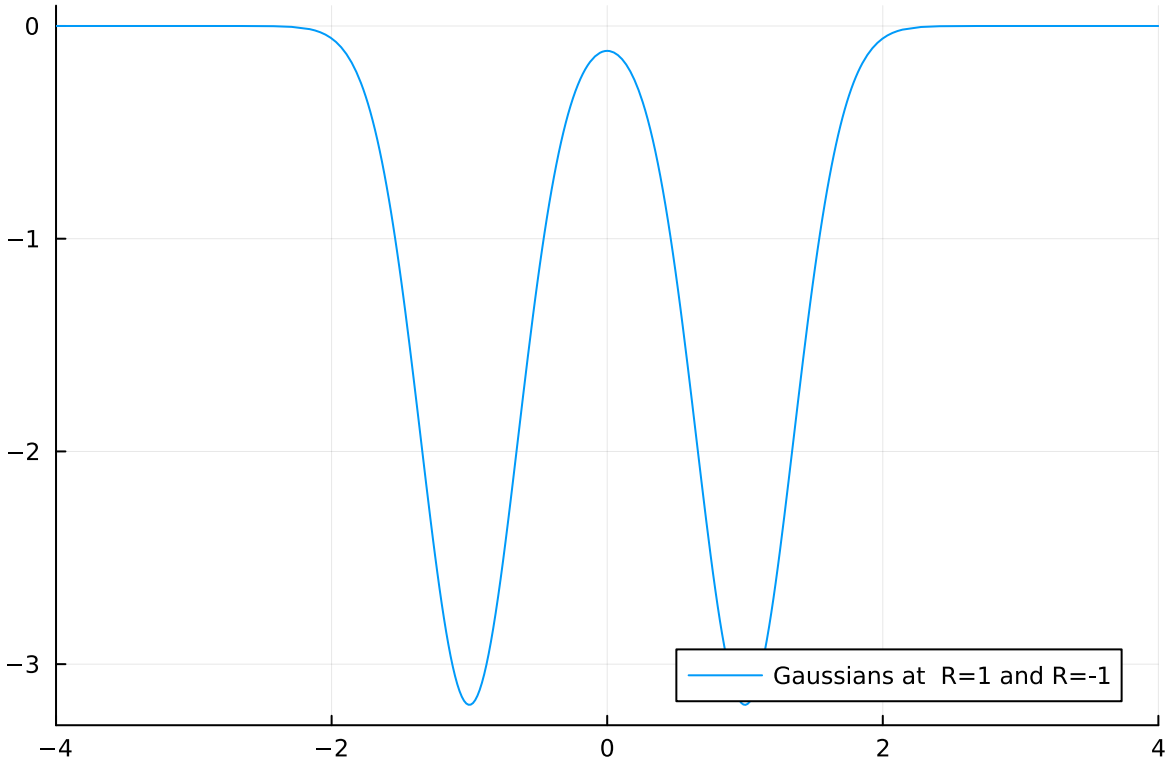
```
v_atom (generic function with 1 method)
```

```
1 v_atom(x::T; α=2, σ=1//4) where {T} = - α / (σ * sqrt(2T(π))) * exp(-x^2 ./ 4σ^2 )
```




```
1 plot(x -> v_atom(x), xlims=(-4, 4), label="Gaussian at R=0")
```

i.e. a confining Gaussian-shaped well. If two atoms are present, e.g. at $R = \pm 1$ one would argue their effect adds up, i.e. one would expect the total potential to be




```
1 plot(x -> v_atom(x + 1) + v_atom(x - 1), xlims=(-4, 4), label="Gaussians at  R=1
  and R=-1")
```

However, this is exactly where it starts to matters whether we allow for quantum tunnelling or not.

We defer the case without quantum tunnelling for later and first discuss the **full quantum-mechanical treatment**. In this setting the potential from multiple atoms indeed just adds up as plotted above, such that the total potential acting on a boson at position x subject to the full chain (with all atoms $j = 1, \dots, M$) is

$$v^{\text{chain}}(x) = \sum_{j=1}^{M} v^{\text{atom}}(x - R_j)$$

where the R_j are the positions of the M atoms.

Since the bosons are non-interacting with each other, the total potential generated by all bosons at positions $x_1, \dots, x_N$ is

$$V(x_1, \dots, x_N) = \sum_{i=1}^{N} v^{\text{chain}}(x_i) \qquad (*).$$

Similarly each boson contributes a kinetic operator term $-\frac{1}{2}\frac{\partial^2}{\partial x_i^2}$, resulting in the total Hamiltonian of the system as

$$\mathcal{H} = -\frac{1}{2}\sum_{i=1}^{N} \frac{\partial^2}{\partial x_i^2} + V(x_1, \dots, x_N),$$

which is an operator in $L^2(\mathbb{R}^N)$. We are interested in its ground state, i.e. the lowest eigenpair $(E_1, \Psi_1) \in \mathbb{R} \times L^2(\mathbb{R}^N, \mathbb{C})$ such that we need to solve

$$\mathcal{H}\Psi_1 = E_1\Psi_1.$$

on the open domain $\mathbb{R}^N$.

Task 1: Ground state energy if tunneling is allowed

Assume that all eigenstates Ψ of $\mathcal{H}$ factorise as

$$\Psi(x_1, x_2, \dots, x_N) = \prod_{i=1}^{N} \psi_i^{\text{chain}}(x_i),$$

where the $\psi_i^{\text{chain}} \in L^2(\mathbb{R}, \mathbb{C})$ are themselves eigenstates of the one-particle Hamiltonian

$$H^{\text{chain}} = -\frac{1}{2}\Delta + v^{\text{chain}},$$

i.e. they satisfy

$$\left(H^{\text{chain}}\psi_i^{\text{chain}}\right)(x) = -\frac{1}{2}\frac{\partial^2}{\partial x^2}\psi_i^{\text{chain}}(x) + v^{\text{chain}}(x)\psi_i^{\text{chain}}(x) = \varepsilon_i^{\text{chain}}\psi_i^{\text{chain}}(x).$$

Show that the energy eigenvalue of $\mathcal{H}$ corresponding to Ψ is the sum

$$E = \sum_{i=1}^{N} \varepsilon_i^{\text{chain}}.$$

Based on this result show that the ground state Ψ_1 is just

$$\Psi_1(x_1, x_2, \dots, x_N) = \prod_{i=1}^{N} \psi_1^{\text{chain}}(x_i),$$

i.e. the N-fold product of the lowest eigenstate of H^{chain}, with correspondingly energy $E_1 = N\varepsilon_1^{\text{chain}}$. *Hint:* Keep in mind our default ordering of the eigenvalues.

Note: If you are more familiar with Fermionic systems (like electrons) you might be surprised by this result. However, keep in mind that bosons don't follow the Pauli exclusion principle.

Solution:

Taking into account that

$$\mathcal{H} = -\frac{1}{2}\sum_{i=1}^{N}\frac{\partial^2}{\partial x_i^2} + V(x_1,\ldots,x_N) = \sum_{i=1}^{N} H_i^{\text{chain}},$$

we can derive the following:

$$\begin{aligned}
\mathcal{H}\Psi(x_1,x_2,\ldots,x_N) &= \sum_{i=1}^{N} H_i^{\text{chain}} \prod_{j=1}^{N}\psi_j^{\text{chain}}(x) \\
&= \sum_{i=1}^{N} H_i^{\text{chain}}\psi_i^{\text{chain}} \prod_{j\neq i}\psi_j^{\text{chain}}(x) \\
&= \sum_{i=1}^{N} \varepsilon_i^{\text{chain}}\psi_i^{\text{chain}} \prod_{j\neq i}\psi_j^{\text{chain}}(x) \\
&= \sum_{i=1}^{N} \varepsilon_i^{\text{chain}} \prod_{j=1}^{N}\psi_j^{\text{chain}}(x) = \sum_{i=1}^{N} \varepsilon_i^{\text{chain}}\Psi(x_1,x_2,\ldots,x_N).
\end{aligned}$$

Using the result above, let's consider the ground state Ψ_1. We have:

$$\mathcal{H}\Psi_1 = E_1\Psi_1,$$

where E_1 corresponds to the smallest eigenvalue and at the same time $E_1 = \sum_{i=1}^{N}\varepsilon_i^{\text{chain}}$.

Considering the default ordering of eigenvalues from smallest to largest, the ground state of H^{chain} is denoted as $\psi_1^{\text{chain}}(x)$ with corresponding the lowest eigenvalue denoted as $\varepsilon_1^{\text{chain}}$.

Therefore, the ground state energy is defined as follows:

$$E_1 = \sum_{i=1}^{N}\varepsilon_1^{\text{chain}} = N\varepsilon_1^{\text{chain}}.$$

This is true if the ground state Ψ_1 is the following:

$$\Psi_1(x_1,x_2,\ldots,x_N) = \prod_{i=1}^{N}\psi_1^{\text{chain}}(x_i).$$

Next we consider the setting **without quantum tunnelling**. In this case the behaviour of the bosonic particles with respect to the crossing of the barriers between the potential wells is classical, that is to say that they can only cross, if their kinetic energy on the high point of the barrier (i.e. at $x = 0$ in the second above plot) is non-zero. Since we are seeking the overall ground state of the system, i.e. where the energy is minimal, we argue that particles thus cannot traverse between the wells in the classical picture (otherwise their total energy is too large to be in the ground state).

Thus in this setting a particle is trapped in one specific well, i.e. if boson i is trapped in well j, then its potential contribution is $v^{\mathrm{atom}}(x_i - R_j)$. To simplify notation we will also refer to this potential as $v^{\mathrm{atom}}(x_i - \rho_i)$, understanding that $\rho_i = R_j$ for the appropriate j. The total potential then becomes

$$V^{\mathrm{cl}}(x_1 - \rho_1, \ldots, x_N - \rho_N) = \sum_{i=1}^{N} v^{\mathrm{atom}}(x_i - \rho_i),$$

where the subscript $_{\mathrm{cl}}$ indicates that we are dealing with the classical description of the barrier crossing. Thus the total Hamiltonian for this system is

$$\widetilde{\mathcal{H}}^{\mathrm{cl}} = -\frac{1}{2} \sum_{i=1}^{N} \frac{\partial^2}{\partial x_i^2} + V^{\mathrm{cl}}(x_1 - \rho_1, \ldots, x_N - \rho_N).$$

Since we solve on the open domain $\mathbb{R}^N$ this Hamiltonian can be simplified by performing a coordinate transformation $x_1 \to x_1 + \rho_1$ without changing its eigenvalues or eigenstates (apart from translation). Thus the final Hamiltonian is

$$\mathcal{H}^{\mathrm{cl}} = -\frac{1}{2} \sum_{i=1}^{N} \frac{\partial^2}{\partial x_i^2} + V^{\mathrm{cl}}(x_1, \ldots, x_N),$$

which has a ground state energy and state $(E_1^{\mathrm{cl}}, \Psi_1^{\mathrm{cl}})$ with

$$\mathcal{H}^{\mathrm{cl}} \Psi_1^{\mathrm{cl}} = E_1^{\mathrm{cl}} \Psi_1^{\mathrm{cl}}.$$

Similarly to *Task 1* we can show that all eigenstates Ψ_1^{cl} factorise and thus that the ground state is necessarily

$$\Psi_1^{\mathrm{cl}}(x_1, \ldots, x_N) = \prod_{i=1}^{N} \psi_1^{\mathrm{atom}}(x_i),$$

where the one-particle states ψ^{atom} are themselves the lowest-energy states of the single-particle eigenproblems

$$H^{\text{atom}}\psi_1^{\text{atom}} = \varepsilon_1^{\text{atom}}\psi_1^{\text{atom}} \quad \text{where} \quad H^{\text{atom}} = -\frac{1}{2}\Delta + v^{\text{atom}}(x).$$

The total energy is again the sum of these one-particle energies

$$E_1^{\text{cl}} = \sum_{i=1}^{N} \varepsilon_1^{\text{atom}} = N\varepsilon_1^{\text{atom}}.$$

Finally, the difference $\Delta\varepsilon = (E_1^{\text{cl}} - E_1)/N = \varepsilon_1^{\text{atom}} - \varepsilon_1^{\text{chain}}$, which we will refer to as **tunnelling energy per particle** is exactly what characterises the effect of quantum tunnelling in this system, which is the quantity of interest we want to obtain.

Overview of the computational procedure

To compute ΔE we thus need to compute the lowest-energy eigenpair of

$$\left(-\frac{1}{2}\Delta + v^{\text{atom}}(x)\right)\psi_1^{\text{atom}}(x) = \varepsilon_1^{\text{atom}}\psi_1^{\text{atom}}(x) \qquad (\text{CL}).$$

and

$$\left(-\frac{1}{2}\Delta + v^{\text{chain}}(x)\right)\psi_1^{\text{chain}}(x) = \varepsilon_1^{\text{chain}}\psi_1^{\text{chain}}(x) \qquad (\text{QM}),$$

both on the open domain $x \in \mathbb{R}$, where

$$v^{\text{chain}}(x) = \sum_{j=1}^{M} v^{\text{atom}}(x - R_j).$$

The tunnelling energy per particle is then simply obtained as the difference between ground state energies $\Delta\varepsilon = \varepsilon_1^{\text{atom}} - \varepsilon_1^{\text{chain}}$

Discretisation and numerical solution

Both (CL) and (QM) are Schrödinger-type eigenproblems

$$\left(-\frac{1}{2}\Delta + v(x)\right)\psi = \varepsilon\psi \qquad (*),$$

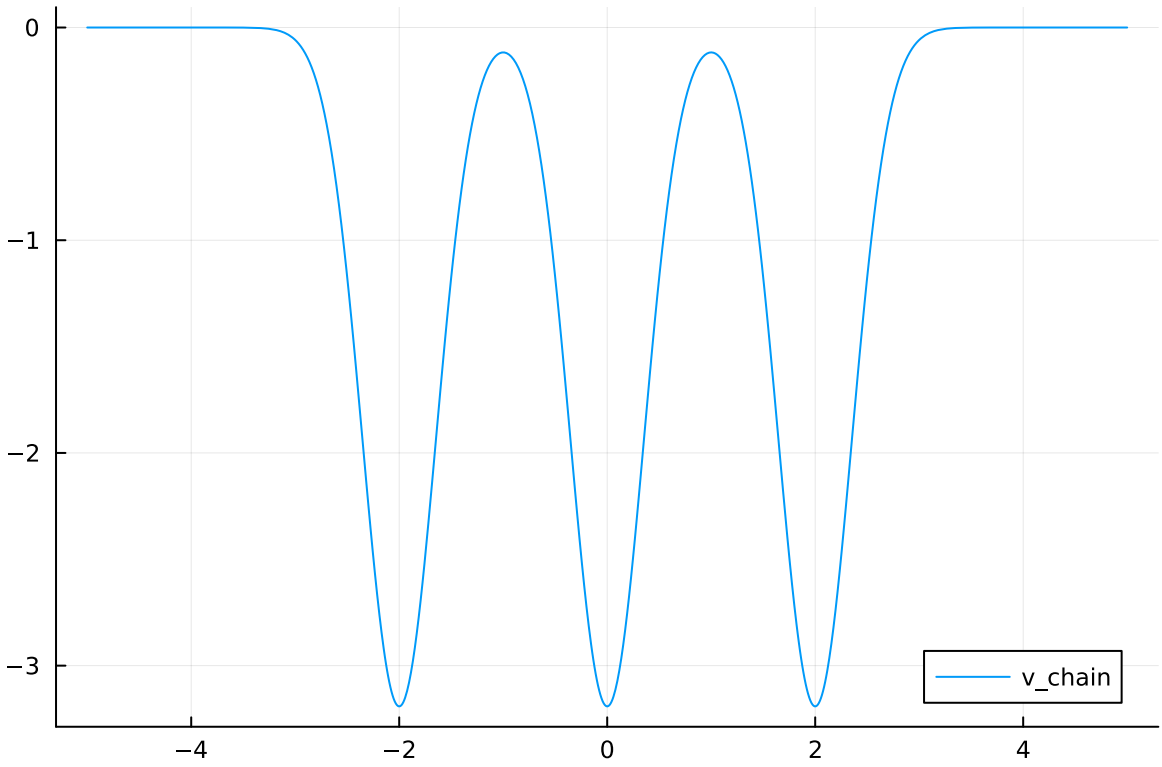
which is defined on the open computational domain $x \in \mathbb{R}$. In line with our previous discussion (e.g. in exercise sheet 2), we will reduce this computational domain to the interval $x \in (-a, a)$ together with Dirichlet boundary conditions $\psi(-a) = \psi(a) = 0$. The interior is modeled on N_b points using $(N_b + 2)$-point finite differences. This discretises the operator into a tridiagonal $\mathbb{R}^{N_b \times N_b}$ matrix.

Task 2: Developing a tailored iterative eigensolver

To develop an initial numerical solver, we consider the full quantum-mechanical setting with a chain of equal 3 atoms located at $R_1 = 0$ and $R_2 = -R_3 = 2$ with $\alpha = 2$ and $\sigma = \frac{1}{4}$ (the default arguments of `v_atom`). In $(*)$ this amounts to the potential

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

```
1 v_chain(x) = v_atom(x + 2) + v_atom(x) + v_atom(x - 2)
```




```
1 plot(v_chain, label="v_chain")
```

(a) Code up a function `fd_hamiltonian(V, Nb, a; T=Float64)`, which returns the finite-difference discretised Hamiltonian corresponding to a potential V using $(N_b + 2)$-point finite differences (i.e. where the domain $(-a, a)$ is split into N_b equispaced interior points for the FD scheme). Make sure your function is type-stable with respect to the floating-point type `T`: if `T=Float32`, for example, all computation should be done in `Float32`.

Solution:

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

```
function fd_hamiltonian(V, Nb, a; T=Float64)
    grid_points = range(-a, stop=a, length=(Nb+2))[2:end-1]
    h = 2a / T(Nb+1)

    diag = - 2ones(T, Nb) ./ h^2
    side_diag = ones(T, Nb-1) ./ h^2
    fd_laplacian = SymTridiagonal(diag, side_diag)

    Vm = Diagonal(vec(T.([V(point) for point in grid_points])))
    fd_Hm = - T(0.5) * fd_laplacian + Vm
end
```

(b) The following code performs a benchmark of the matrix returned by `fd_hamiltonian` for $N_b = 500$:

```
[-901.537, 3809.6, -4826.8, 302.7, 3758.28, -6583.05, 7561.03, -1981.17, 3386.36, -16035.
```

```
let
    H = fd_hamiltonian(v_chain, 500, 4);
    x = randn(size(H, 2))
    @btime $H * $x
end
```

```
  473.737 ns (1 allocation: 4.06 KiB)
```

Perform the same benchmarks for the `\` (backslash operator, `H \ x`) with a random vector. Repeat with a factorised form of the Hamiltonian (`factorize(H)`). Also repeat all three benchmarks when you take `H` to be a random dense matrix (`H = randn(500, 500)`) instead of the matrix returned by `fd_hamiltonian`. What do you observe? Comment on the balance of timings between `*` and `\` (in both factorised and unfactorised form). Can you describe why it is helpful to employ `SymTridiagonal` to exploit the special structure of our discretised Hamiltonian ?

Hint: It might be helpful to read up on `@btime` and `@benchmark` to understand the syntax of `@btime`, especially the implications of the `$`.

Solution:

```
[0.00147789, 0.00315347, 0.00465137, 0.0059189, 0.00648378, 0.00631407, 0.00619178, 0.00(
```

```
begin
    H = fd_hamiltonian(v_chain, 500, 4);
    x = randn(size(H, 2));
    H_factorized = factorize(H);

    @btime $H * $x
    @btime $H \ $x
    @btime $H_factorized \ $x
end
```

```
  471.717 ns (1 allocation: 4.06 KiB)
  5.383 µs (3 allocations: 12.19 KiB)
  1.978 µs (1 allocation: 4.06 KiB)
```

```
[-1.40779, 0.649064, 0.778667, 1.35221, -0.102099, 1.27988, 0.838614, 2.98139, -1.15299,
```

```
begin
    H_dense = randn(500, 500);
    H_dense_factorized = factorize(H_dense);

    @btime $H_dense * $x
    @btime $H_dense \ $x
    @btime $H_dense_factorized \ $x
end
```

```
  26.800 µs (1 allocation: 4.06 KiB)
  2.716 ms (4 allocations: 1.92 MiB)
  40.400 µs (1 allocation: 4.06 KiB)
```

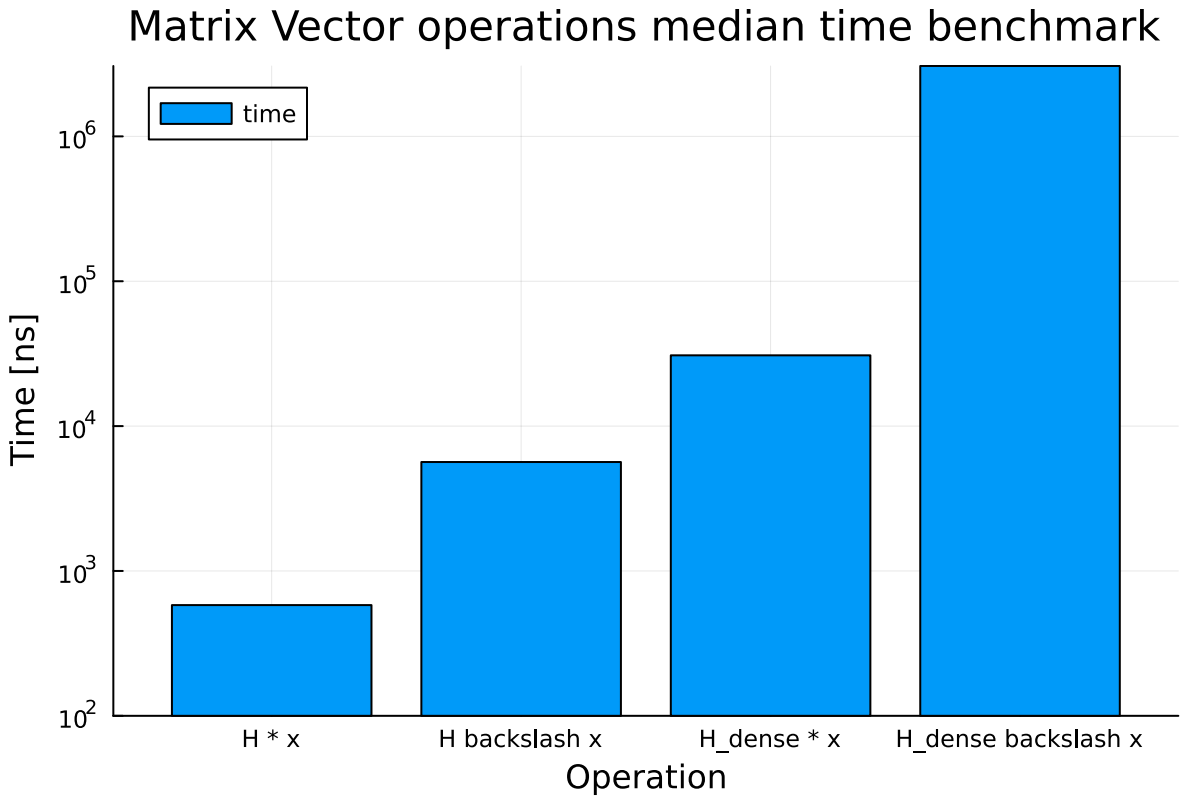
Observations:

- We can see that for both the specially structured matrix returned by `fd_hamiltonian` and a randomly generated dense matrix matrix-vector multiplication (`*`) is faster than the backslash operator (`\`).
- Using `SymTridiagonal` for the specific structure of the discretized Hamiltonian reduces storage needs and makes calculations more efficient. It also enables faster matrix-vector multiplications through optimized algorithms.
- The factorized form of the Hamiltonian (`factorize(H)`) speeds up the backslash operator in both cases: applied to structured matrix H and a randomly generated dense matrix.

```
BenchmarkTools.Trial: 1522 samples with 1 evaluation.
 Range (min … max):  2.731 ms …   6.987 ms  ┊ GC (min … max): 0.00% … 52.35%
 Time  (median):     3.062 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   3.272 ms ± 609.623 μs  ┊ GC (mean ± σ):  3.06% ±  8.59%
```


```
Memory estimate: 1.92 MiB, allocs estimate: 4.
```

```
begin
    H_mult = @benchmark $H * $x
    H_div=@benchmark $H \ $x

    H_dense_mult=@benchmark $H_dense * $x
    H_dense_div=@benchmark $H_dense \ $x
end
```




```
begin
    bar(["H * x", "H backslash x", "H_dense * x", "H_dense backslash x"],
        [Float64(median(H_mult).time), Float64(median(H_div).time),
        Float64(median(H_dense_mult).time), Float64(median(H_dense_div).time)],
        xlabel="Operation", ylabel="Time [ns]",
        yaxis=:log, labels="time",
        title="Matrix Vector operations median time benchmark")
end
```

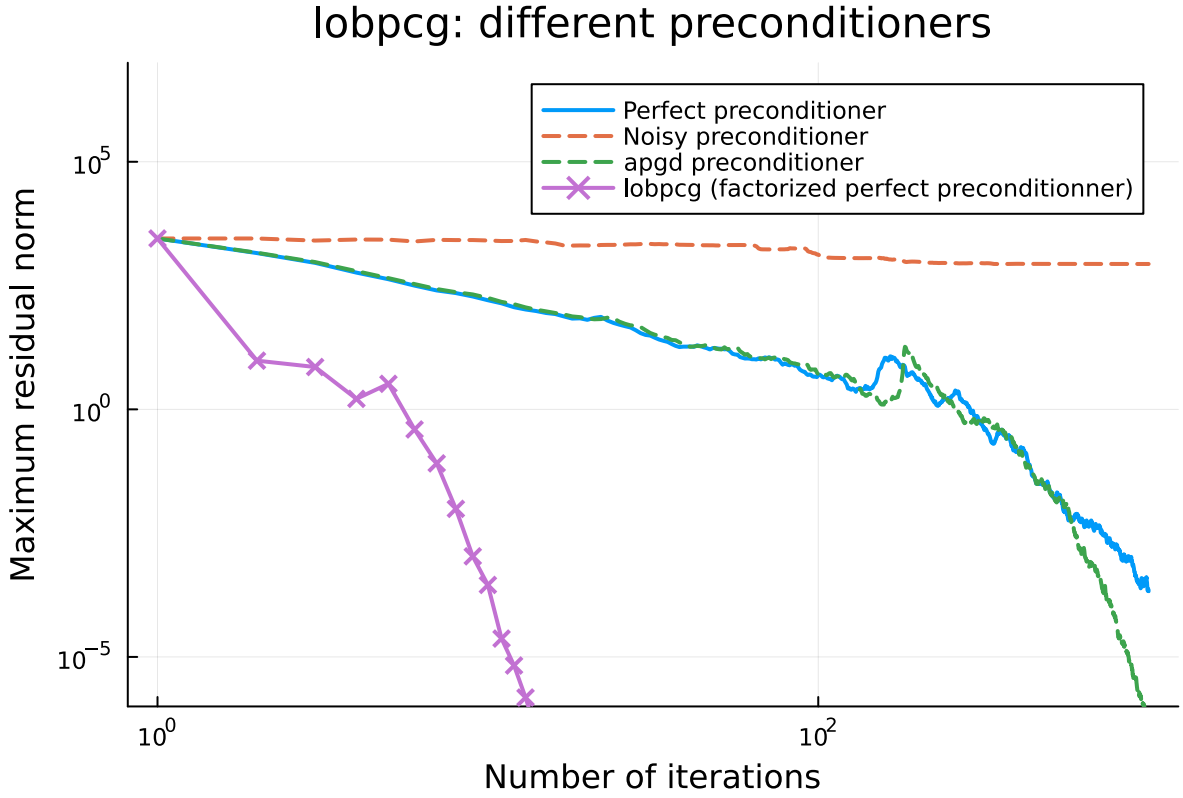
(c) In one of the later code boxes a copy of the `lobpcg` routine of the lectures is defined. Use this function to find the 3 smallest eigenpairs of the discretised Hamiltonian with `v_chain` as the potential. Use $a = 4$ and $N_b = 500$ and converge until `tol = 1e-6`. Try to experiment a bit with the preconditioners available to you. Take a look at the lecture on diagonalisation routines to get some inspiration. You should find that a good preconditioner is crucial to get this problem to converge within 20–30 iterations. Make sure that with your setup the convergence in 20–30 iterations is stable with respect to increasing N_b.

Solution:

```
begin
    a = 4
    Nb = 500
    tol = 1e-6
    H_fd = fd_hamiltonian(v_chain, Nb, a)

    n = 3
    X0 = randn(size(H_fd, 2), n)


    # Perfect preconditioner
    precond = diagm(1.0 ./ diag(H_fd))
    eigenvalues, eigenvectors = lobpcg(H_fd; X = X0, verbose = false, tol = tol,
    maxiter = 30, Pinv = precond)

    println("Eigenvalues: ", eigenvalues)
    println("Eigenvectors: ", eigenvectors)
end
```

```
Eigenvalues: [3.132962991799481, 4.607150330892259, 6.566513229512106]
Eigenvectors: [0.005165902670314775 0.0033815542069388716 -0.0014192426483634185; 0.01038445857841396 0.0066062938674489235 -0.0018437232768683766; … ; -0.00529370653245584 0.01790260768303232 0.0069221440300071884; -0.002815865516362055 0.009397619091781363 0.0038006250059556177]
```

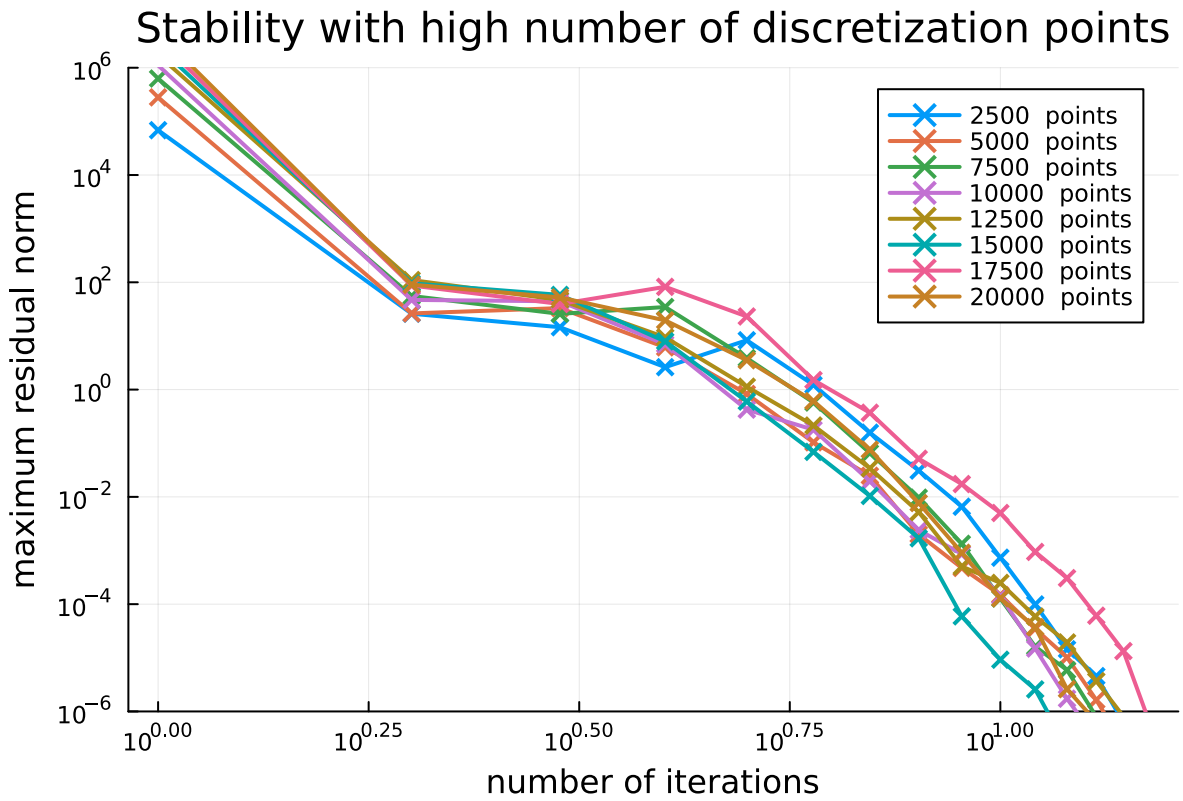
Preconditioner Noise Level: `log_prec_noise =` -2.5


lobpcg: different preconditioners
Perfect preconditioner
Noisy preconditioner
apgd preconditioner
lobpcg (factorized perfect preconditionner)
Maximum residual norm
10^5
10^0
10^{-5}
10^0
10^2
Number of iterations

```
begin
    p = plot(yaxis=:log,xaxis=:log,title=string(lobpcg) * ": different preconditioners" ,xlabel="Number of iterations",ylabel="Maximum residual norm",ylims=(1e-6, 1e7))

    # Perfect preconditioner
    inv_diag_residual_norms = lobpcg(H_fd; X = X0, verbose = false, tol = tol, maxiter = 1000, Pinv = precond).residual_norms
    max_rnorm_perfect=[maximum(r_norms) for r_norms in inv_diag_residual_norms]

    plot!(p, max_rnorm_perfect; label = "Perfect preconditioner", lw = 2)

    # Preconditioner plus noise
    prec_noise = 10 ^ log_prec_noise * randn(size(H_fd, 1))
    noisy_precond = Diagonal(1 ./ diag(H_fd) .+ prec_noise)
    noisy_rnorms = lobpcg(H_fd; X = X0, verbose = false, tol = tol, maxiter = 1000, Pinv = noisy_precond).residual_norms
    max_rnorm_noisy=[maximum(r_norms) for r_norms in noisy_rnorms]

    plot!(p, max_rnorm_noisy; label = "Noisy preconditioner", lw = 2,       ls = :dash)

    # Apgd preconditioner
    Alopcg = Diagonal(abs.(randn(Nb)).^0.1);
    Pinv = Diagonal(1 ./ diag(Alopcg))  # Diagonal preconditioner for Apgd
    apgd_rnorms = lobpcg(H_fd; X = X0, verbose = false, tol = tol, maxiter = 1000, Pinv).residual_norms
    max_rnorm_apgd=[maximum(r_norms) for r_norms in apgd_rnorms]

    plot!(p, max_rnorm_apgd; label = "apgd preconditioner", lw  = 2, ls = :dash)

    #perfect preconditionner using factorization
    Pinv= H_fd\I
    factorized_inv_diag_residual_norms = lobpcg(H_fd; X = X0, verbose = false, tol = tol, Pinv).residual_norms
    max_rnorms_factorized= [maximum(r_norms) for r_norms in factorized_inv_diag_residual_norms]

    plot(p, max_rnorms_factorized; label = string(lobpcg) * " (factorized perfect preconditionner)",  lw = 2,mark = :x)
    plot!()
end
```

As can be seen on the plot above,the factorization plays a crucial role to reduce the number of iteration steps required.

On the plot below we can assess the stability of the algorithm with increased number of discretization points N_b.




```
begin
        k = plot(yaxis=:log,xaxis=:log, ylims=(1e-6, 1e6),title="Stability with high number of discretization points",xlabel="number of iterations",ylabel="maximum residual norm")
    for Nb in 2500:2500:20000
        H_nb=fd_hamiltonian(v_chain, Nb, 4)
        X = randn(eltype(H_nb), size(H_nb,2), 3)

        Pinv_nb= factorize(H_nb) \I

        nb_inv_diag_residual_norms = lobpcg(H_nb; X = X, verbose = false, tol = 1e-6, Pinv=Pinv_nb,maxiter=30).residual_norms
        nb_factorized_max_residual_norm_perfect=[maximum(residual_norms) for residual_norms in nb_inv_diag_residual_norms]

        plot!(k,nb_factorized_max_residual_norm_perfect; label = string(Nb) * " points",    lw = 2,mark = :x)
    end
    plot!()
end
```

It appears that the algorithm does not suffer from unstability even for very thin discretizations.

Orthogonalisation routines

In a physical investigation one would want to run many diagonalisations to investigate different chains by varying their length or their parameters α and σ. We can thus expect that many eigenpair computations are necessary, justifying to study the overall performance of our diagonalisation routines.

In the lecture about diagonalisation algorithms we discussed a basic LOBPCG algorithm. In this implementation we used QR factorisation as a way to orthonormalise the columns of a matrix $X \in \mathbb{R}^{n \times p}$ with $p \ll n$:

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

```
1 ortho_qr(X) = Matrix(qr(X).Q)
```

The idea here was that QR factorisation $X = QR$ produces an orthogonal matrix Q and a upper-triangular matrix R, such that we may just drop the R and return the Q itself.

Based on this approach we repeat an LOBPCG implementation here, along with timers to track the time the algorithm spends in key parts:

```
1 const to = TimerOutput();  # Setup the timer to track timings
```

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

```julia
@timeit to function lobpcg(A; X=randn(eltype(A), size(A, 2), 2), ortho=ortho_qr,
                           Pinv=I, tol=1e-6, maxiter=100, verbose=true)
    T = real(eltype(A))
    m = size(X, 2)  # block size

    eigenvalues    = Vector{T}[]
    residual_norms = Vector{T}[]
    λ = NaN
    P = nothing
    R = nothing

    for i in 1:maxiter
        if i > 1
            Z = hcat(X, P, R)
        else
            Z = X
        end
        @timeit to "Orthogonalisation" begin
            Z = ortho(Z)
        end

        @timeit to "Matrix-vector products" begin
            AZ = A * Z
        end

        @timeit to "Rayleigh-Ritz step" begin
            λ, Y = eigen(Hermitian(Z' * AZ))
        end
        λ = λ[1:m]
        Y = Y[:, 1:m]
        new_X = Z * Y

        @timeit to "Residual computation" begin
            R = AZ * Y - new_X * Diagonal(λ)
            norm_r = norm.(eachcol(R))
        end
        push!(eigenvalues, λ)
        push!(residual_norms, norm_r)
        verbose && @printf "%3i %8.4g %8.4g\n" i λ[end] norm_r[end]
        maximum(norm_r) < tol && break

        @timeit to "Preconditioning" begin
            R = Matrix(Pinv * R)
        end
        P = X - new_X
        X = new_X
    end

    (; λ, X, eigenvalues, residual_norms)
end
```

With the following routine we run this algorithm for a test problem, which we take from the matrix depot, track and print its timings:

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

```
function runtime_lobpcg(n::Integer)
    H = fd_hamiltonian(v_chain, n, 4);
    X = randn(eltype(H), size(H, 2), 10)
    Pinv = InverseMap(factorize(H))

    reset_timer!(to)
    lobpcg(H; Pinv, X, verbose=false)
    to
end
```

In this case, where matrix-vector products are extremely cheap, for small problem sizes the Rayleigh-Ritz step is *usually* the most expensive (about 50%) with the Orthogonalisation being a close second (about 30%).

Note that the timings and their ratios may look a little different on your machine, which is fine.

		Time			Allocations		
Tot / % measured:		256ms / 1.0%			35.4MiB / 5.9%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
lobpcg	1	2.47ms	100.0%	2.47ms	2.08MiB	100.0%	2.08MiB
Rayleigh-Ritz step	11	1.13ms	45.9%	103µs	410KiB	19.2%	37.3KiB
Orthogonalisation	11	942µs	38.2%	85.7µs	704KiB	33.0%	64.0KiB
Preconditioning	10	126µs	5.1%	12.6µs	80.2KiB	3.8%	8.02KiB
Residual computation	11	91.4µs	3.7%	8.31µs	263KiB	12.3%	24.0KiB
Matrix-vector products	11	37.5µs	1.5%	3.41µs	243KiB	11.4%	22.1KiB

```
runtime_lobpcg(100)
```

However, as we increase the problem sizes, the Orthogonalisation starts to dominate more and more:

		Time			Allocations		
Tot / % measured:		8.50ms / 88.6%			10.8MiB / 99.8%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
lobpcg	1	7.53ms	100.0%	7.53ms	10.8MiB	100.0%	10.8MiB
Orthogonalisation	15	4.04ms	53.6%	269µs	3.58MiB	33.2%	244KiB
Rayleigh-Ritz step	15	1.91ms	25.3%	127µs	571KiB	5.2%	38.1KiB
Preconditioning	14	437µs	5.8%	31.2µs	549KiB	5.0%	39.2KiB
Residual computation	15	406µs	5.4%	27.1µs	1.72MiB	16.0%	117KiB
Matrix-vector products	15	222µs	2.9%	14.8µs	1.64MiB	15.2%	112KiB

```
1 runtime_lobpcg(500)
```

		Time			Allocations		
Tot / % measured:		109ms / 99.2%			107MiB / 100.0%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
lobpcg	1	109ms	100.0%	109ms	107MiB	100.0%	107MiB
Orthogonalisation	16	51.0ms	47.0%	3.19ms	35.4MiB	33.2%	2.21MiB
Residual computation	16	13.3ms	12.3%	834µs	18.3MiB	17.2%	1.14MiB
Rayleigh-Ritz step	16	8.48ms	7.8%	530µs	611KiB	0.6%	38.2KiB
Matrix-vector products	16	4.36ms	4.0%	272µs	17.5MiB	16.5%	1.10MiB
Preconditioning	15	4.25ms	3.9%	283µs	5.72MiB	5.4%	391KiB

```
1 runtime_lobpcg(5000)
```

This motivates to take a closer look at orthogonalisation routines as a way to make LOBPCG computations cheaper for larger systems.

Note: For more involved problems (e.g. electronic structure computations) the matrix-vector products are usually the dominating step, such that the orthogonalisation takes a smaller fraction of the overall runtime. Still, even in those settings the cost of orthogonalisations can usually not be completely neglected.

Task 3: Gram-Schmidt procedure

A classical approach to orthogonalising a set of vectors is the Gram-Schmidt procedure.

Given a matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ with $p \ll n$ a naive implementation using Gram-Schmidt to orthonormalising its columns is

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

```
@views function ortho_gs(X)
    Xnew = copy(X)
    for i in 1:size(X, 2)
        for j in 1:(i-1)
            Xnew[:, i] .-= Xnew[:, j] * dot(Xnew[:, j], X[:, i])
        end
        Xnew[:, i] ./= norm(Xnew[:, i])
    end
    Xnew
end
```

where the `@views` macro is used to suppress the copies Julia usually performs when accessing slices of vectors or matrices with `[ ... ]`.

(a) A simple benchmark using BenchmarkTools of the above `ortho_gs` function for orthogonalising a random matrix of size 1000×10 yields

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):   44.500 μs …  10.981 ms  ┊ GC (min … max):  0.00% … 95.61%
 Time  (median):      58.700 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   107.793 μs ± 259.520 μs  ┊ GC (mean ± σ):  18.40% ±  9.01%
```

```
 44.5 μs       Histogram: log(frequency) by time        629 μs <

 Memory estimate: 435.36 KiB, allocs estimate: 47.
```

```
let
    X = randn(1000, 10);
    @benchmark ortho_gs($X)
end
```

This is not bad, but this can still be improved. One idea is to remove the inner loop over `j`. This can be achieved by using matrix-matrix products instead of vector-only operations. Since matrix-matrix operations are generally faster, this gives a speedup. Rewrite `ortho_gs` by using to a larger extend matrix-matrix operations. Call the new function `ortho_gs_matrix`. Benchmark this second function as well. Compare the runtime of both functions to `ortho_qr`. How much faster is Gram-Schmidt compared to QR?

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

```
@views function ortho_gs_matrix(X)
    Xnew = copy(X)
    n, m = size(Xnew)

    Xnew[:, 1] ./= norm(Xnew[:, 1])

    for i in 2:m
        projections = Xnew[:, 1:(i-1)]' * Xnew[:, i]
        Xnew[:, i] .-= Xnew[:, 1:(i-1)] * projections
        Xnew[:, i] ./= norm(Xnew[:, i])
    end
    Xnew
end
```

```
begin
    X = randn(1000, 10);
    result_gs = @benchmark ortho_gs($X)
    result_gs_matrix = @benchmark ortho_gs_matrix($X)
    result_qr = @benchmark ortho_qr($X)

    println("Orthogonal GS Time: ", mean(result_gs).time)
    println("Ortho GS Matrix Time: ", mean(result_gs_matrix).time)
    println("Ortho QR Time: ", mean(result_qr).time)
end
```

```
Orthogonal GS Time: 121732.3
Ortho GS Matrix Time: 45260.8
Ortho QR Time: 129532.86
```

```
2.8619215745192306
```

```
mean(result_qr).time / mean(result_gs_matrix).time
```

We can see that Gram-Schmidt is approximately 3 times faster compared to QR.

```
md"""
We can see that Gram-Schmidt is approximately 3 times faster compared to QR.
"""
```

(b) Motivated by the speedup of Gram-Schmidt methods we want to employ `ortho_gs` and `ortho_gs_matrix` within our `lobpcg`. This can be achieved by setting the `ortho` keyword argument appropriately (e.g. `lobpcg(...; ortho=ortho_gs)`). Focus on computing the 4 smallest eigenpairs of the testproblem `Htest()` to `tol=1e-8` — using again its factorised form as a preconditioner.

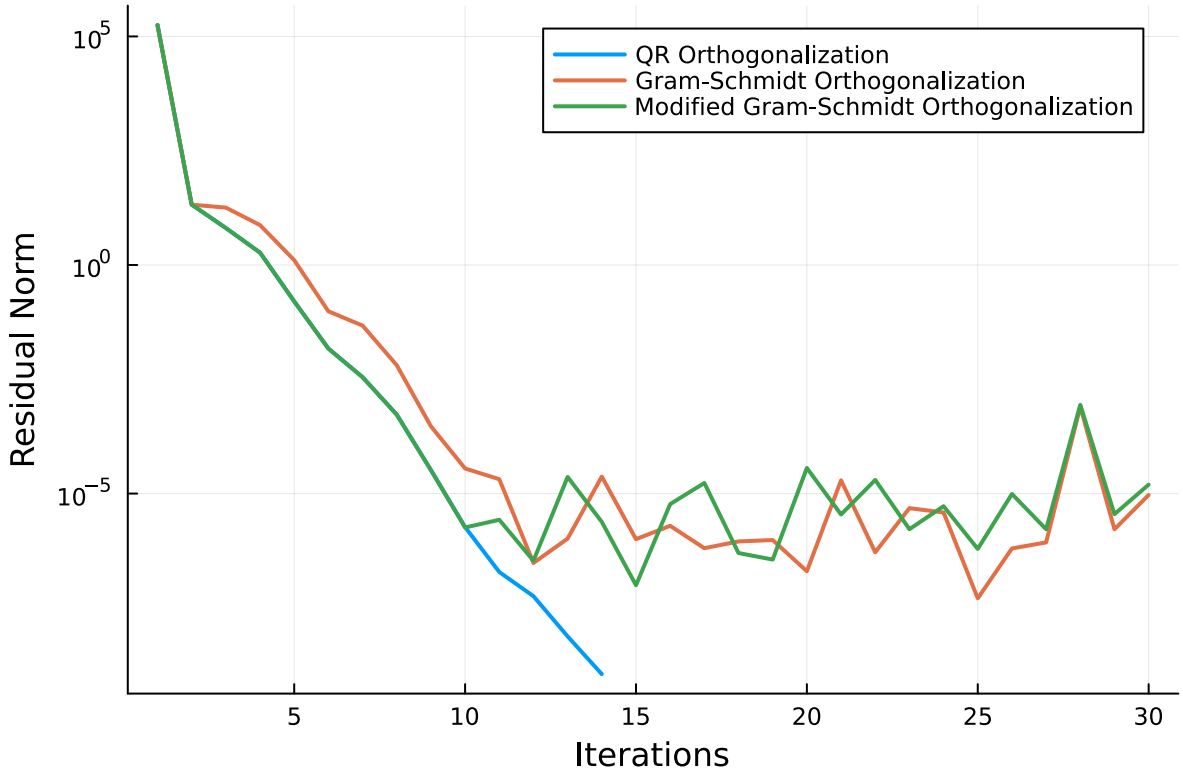
```
Htest(T=Float64) = fd_hamiltonian(v_chain, 4000, 4; T);
```

What do you notice with the Gram-Schmidt-based orthogonalisation? Do we win something using this faster technique? Plot the convergence history of the largest eigenpair when using `ortho_qr` and `ortho_gs`. Try to find an explanaition for the observed behaviour. *Hint:* Part (c) might be helpful.

Answer:

```
4000×4000 LinearMaps.InverseMap{Float64}
```

```
begin
    Htest(T=Float64) = fd_hamiltonian(v_chain, 4000, 4; T);
    X1 = randn(eltype(Htest()), size(Htest(), 2), 4)
    precond_inv = InverseMap(factorize(Htest()))
end
```




```
begin
    res_qr = lobpcg(Htest(); X=X1, Pinv=precond_inv, ortho=ortho_qr, tol=1e-8, maxiter = 30, verbose=false)
    res_gs = lobpcg(Htest(); X=X1, Pinv=precond_inv, ortho=ortho_gs, tol=1e-8, maxiter = 30, verbose=false)
    res_mgs = lobpcg(Htest(); X=X1, Pinv=precond_inv, ortho=ortho_mgs, tol=1e-8, maxiter = 30, verbose=false)

    rnorm_qr=[norms[1] for norms in res_qr.residual_norms]
    rnorm_gs=[norms[1] for norms in res_gs.residual_norms]
    rnorm_mgs=[norms[1] for norms in res_mgs.residual_norms]

    plot(rnorm_qr, label="QR Orthogonalization", yaxis=:log, xlabel="Iterations", ylabel="Residual Norm", lw=2)
    plot!(rnorm_gs, label="Gram-Schmidt Orthogonalization", yaxis=:log, lw=2)
    plot!(rnorm_mgs, label="Modified Gram-Schmidt Orthogonalization", yaxis=:log, lw=2)
end
```

Analyzing the plots we can see that while Gram-Schmidt is faster computationally, it may have numerical stability issues compared to QR orthogonalization.

```
md"""
Analyzing the plots we can see that while Gram-Schmidt is faster computationally, it may have numerical stability issues compared to QR orthogonalization.
"""
```

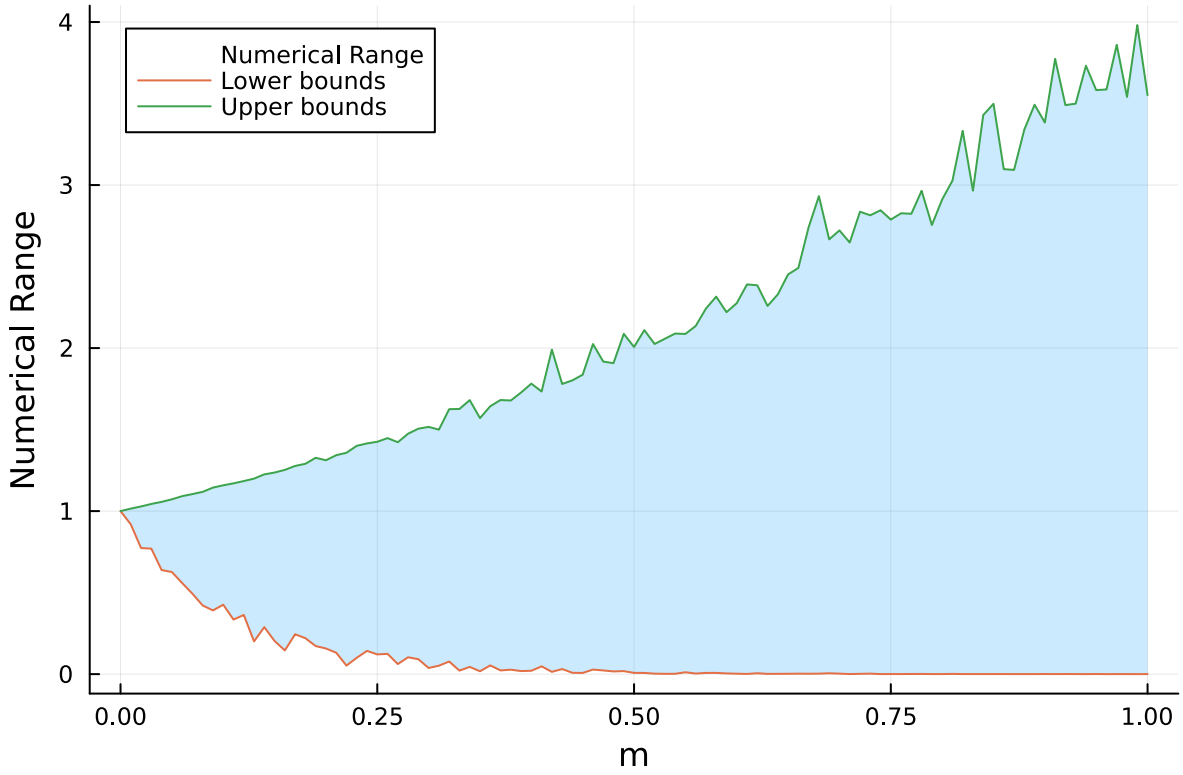
(c) Next we want to test the numerical stability of our orthogonalisation routines. Challenging problems for orthogonalisation arise in particular if the vectors are very similar, e.g. if their entries are within the same numerical range. With this in mind argue why the routine

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

```
1 testmatrix(m; T=Float64, N=1000, M=10) = abs.(randn(T, N, M)) .^ T(m)
```

is a good way of generating challenging benchmark matrices for orthogonalisation routines. *Hint:* Plot the numerical range using the `extrema` function for values $m \in (0, 1)$.

```
1 md"""
2 is a good way of generating challenging benchmark matrices for orthogonalisation
  routines. *Hint:* Plot the numerical range using the `extrema` function for values
  $m \in (0, 1)$.
3 """
```




```
begin
    msteps = 0:0.01:1
    ranges = [extrema(testmatrix(m)) for m in msteps]

    lower_bounds = [r[1] for r in ranges]
    upper_bounds = [r[2] for r in ranges]

    means = [mean(r) for r in ranges]

    # Create the plot with means, lower bounds, and upper bounds
    plot(msteps, means, ribbon=(abs.(lower_bounds .- means), abs.(upper_bounds .- means)), fillalpha=0.2, alpha=0., xlabel="m", ylabel="Numerical Range", label="Numerical Range")
    plot!(msteps, lower_bounds, label="Lower bounds")
    plot!(msteps, upper_bounds, label="Upper bounds")
end
```

If the vectors are very similar, especially when their entries are within the same numerical range, numerical errors may accumulate, leading to loss of orthogonality or other stability issues. In such cases, using Gram-Schmidt orthogonalization methods might be problematic due to the ill-conditioning of the vectors.

The `testmatrix` function is useful for creating matrices that test orthogonalization routines, helping to find numerical stability problems and assess the strength of the implemented algorithms.

```
md"""
If the vectors are very similar, especially when their entries are within the same numerical range, numerical errors may accumulate, leading to loss of orthogonality or other stability issues. In such cases, using Gram-Schmidt orthogonalization methods might be problematic due to the ill-conditioning of the vectors.

The `testmatrix` function is useful for creating matrices that test orthogonalization routines, helping to find numerical stability problems and assess the strength of the implemented algorithms.
"""
```

(d) Test `ortho_gs` and `ortho_qr` for various values of m, i.e. check the orthonormality error `maximum(abs, X'*X - I)` after having orthogonalised test matrices with various values of m. Plot this error for both methods versus m in the range 10^{-16} to 1. What do you notice in particular for the challenging cases ?

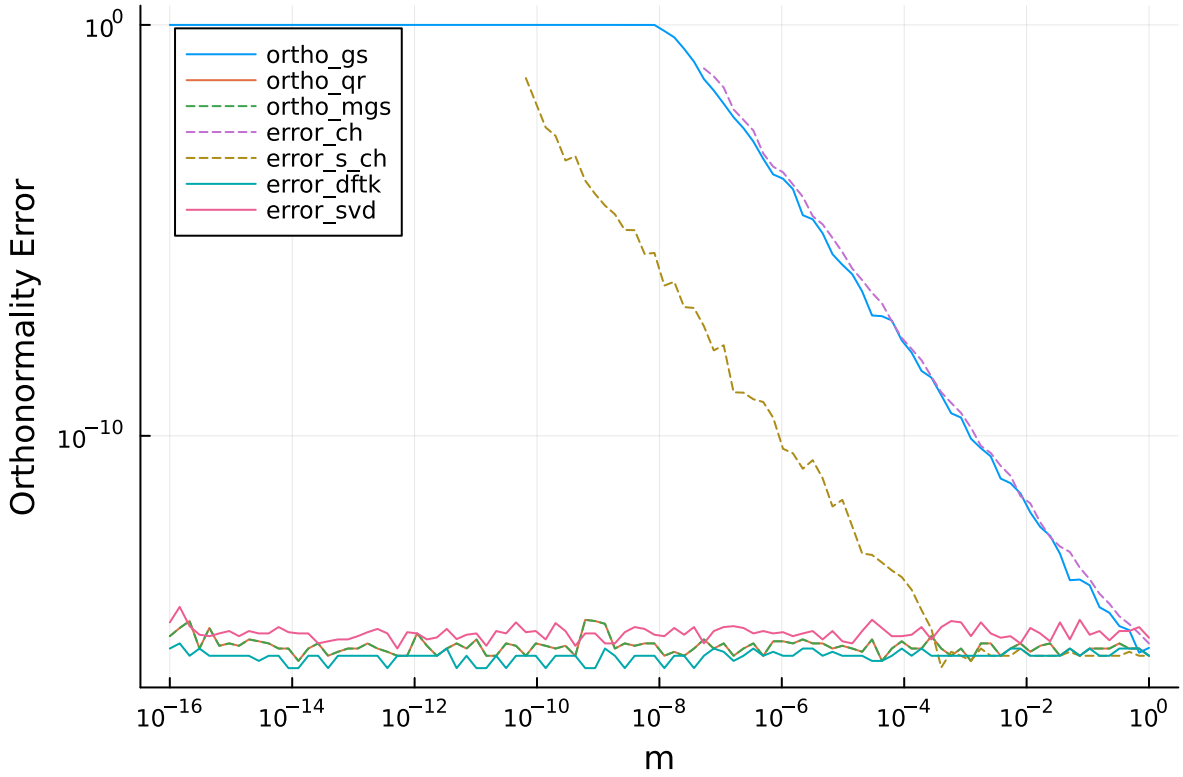
```
md"""
**(d)** Test `ortho_gs` and `ortho_qr` for various values of $m$, i.e. check the orthonormality error `maximum(abs, X'*X - I)` after having orthogonalised test matrices with various values of $m$. Plot this error for both methods versus $m$ in the range $10^{-16}$ to $1$. What do you notice in particular for the challenging cases ?
"""
```

Answer:

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

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

```
orthonormality_error(X) = maximum(abs, X' * X - I)
```


ortho_gs
ortho_qr
ortho_mgs
error_ch
error_s_ch
error_dftk
error_svd
Orthonormality Error
10^{0}
10^{-10}
10^{-16}
10^{-14}
10^{-12}
10^{-10}
10^{-8}
10^{-6}
10^{-4}
10^{-2}
10^{0}
m

```
begin
    error_gs = Float64[]
    error_mgs = Float64[]
    error_qr = Float64[]
    error_ch = Float64[]
    error_s_ch = Float64[]
    error_dftk = Float64[]
    error_svd = Float64[]

    mrange = 10.0 .^ range(-16, stop=0, length=100)

    for m in mrange
        X = testmatrix(m)
        gs_result = ortho_gs(X)
        qr_result = ortho_qr(X)
        mgs_result = ortho_mgs(X)
        dftk_result = ortho_dftk(X)
        svd_result = ortho_svd(X)

        if m > 0.5 * 10. ^ (-7)
            ch_result = ortho_cholesky(X)
            push!(error_ch, orthonormality_error(ch_result))
        else
            push!(error_ch, NaN)
        end

        if m > 0.5 * 10. ^ (-10)
            s_ch_result = ortho_shift_cholesky(X)
            push!(error_s_ch, orthonormality_error(s_ch_result))
        else
            push!(error_s_ch, NaN)
        end

        push!(error_gs, orthonormality_error(gs_result))
        push!(error_qr, orthonormality_error(qr_result))
        push!(error_mgs, orthonormality_error(mgs_result))
        push!(error_dftk, orthonormality_error(dftk_result))
        push!(error_svd, orthonormality_error(svd_result))
    end

    plot(mrange, error_gs, label="ortho_gs", xlabel="m", xaxis=:log10,yaxis=:log10,
    ylabel="Orthonormality Error", legend=:topleft)
    plot!(mrange, error_qr, label="ortho_qr", xticks=10.0 .^ (-16:2:0))
    plot!(mrange, error_qr, label="ortho_mgs", linestyle=:dash)
    plot!(mrange, error_ch, label="error_ch", linestyle=:dash)
    plot!(mrange, error_s_ch, label="error_s_ch", linestyle=:dash)
    plot!(mrange, error_dftk, label="error_dftk")
    plot!(mrange, error_svd, label="error_svd")
end
```

In challenging cases, we can observe higher orthonormality errors that indicate the difficulty of maintaining orthonormality using Gram-Schmidt procedure when the entries of the vectors are within the same small numerical range.

(e) The observed behaviour is a well-known flaw of the standard Gram-Schmidt procedure. A simple and common modification improves the situaton notably. This is known as the *modified Gram-Schmidt procedure* (MGS), which is described for example on wikipedia. Code up an MGS and call the function `ortho_mgs`. Add `ortho_mgs` to your plots in **(b)** and **(d)**. You should observe that the overall behaviour improves. *Hint:* Compared to `ortho_gs` you don't have to change much.

Answer:

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

```
@views function ortho_mgs(X)
    Xnew = copy(X)
    for i in 1:size(X, 2)
        for j in 1:(i-1)
            Xnew[:, i] .-= dot(Xnew[:, i], Xnew[:, j]) * Xnew[:, j]
        end
        Xnew[:, i] ./= norm(Xnew[:, i])
    end
    Xnew
end
```

Analyzing the plots above, we can see that the modified Gram-Schmidt procedure doesn't give high orthonormality errors and on the tested examples is as good as QR method. On the other hand, MGS doesn't show a big improvement within our lobpcg algorithm.

Task 4: Cholesky-based orthogonalisation

Another approach to orthogonalise columns of matrices is based on cholesky factorisation. The idea is that for any matrix $X \in \mathbb{R}^{n\times p}$ with $p \ll n$ the matrix $X^H X$ — the overlap or Gram matrix — is symmetric and positive definite. As a result we can perform a Cholesky decomposition into a lower-triangular matrix L such that $X^H X = LL^H$. A set of orthonormal vectors is then given by $X\left(L^H\right)^{-1}\big)$.

(a) Given an arbitrary $X \in \mathbb{R}^{n\times p}$ and Cholesky factors L such that $X^H X = LL^H$, prove that $X\left(L^H\right)^{-1}\big) \in \mathbb{R}^{n\times p}$ is indeed orthogonal.

Proof:

Let $M = \left(L^H\right)^{-1}$ and consider the product:

$$(XM)^H XM = M^H (X^H X) M = M^H (LL^H) M = \left(\left(L^H\right)^{-1}\right)^H LL^H \left(L^H\right)^{-1} = I,$$

On the other hand,

$$\begin{aligned} XM(XM)^H &= XMM^H X^H = X\left(L^H\right)^{-1} L^{-1} X^H \\ &= X\left(LL^H\right)^{-1} X^H = X\left(X^H X\right)^{-1} X^H \\ &= XX^{-1}(X^H)^{-1} X^H = I. \end{aligned}$$

As a result, we have:

$$\left(X\left(L^H\right)^{-1}\right)^H X\left(L^H\right)^{-1} = I$$

and

$$X\left(L^H\right)^{-1}\left(X\left(L^H\right)^{-1}\right)^H = I,$$

which means that $X\left(L^H\right)^{-1}$ is indeed orthogonal.

(b) Look up how the `cholesky` function works in Julia and use it to code up an `ortho_cholesky(X)`, which orthonormalises the columns of the passed matrix `X`. Benchmark `ortho_cholesky` on a `X = randn(1000, 10)` testmatrix and compare the timings to `ortho_qr` and `ortho_gs_matrix`. Can you explain the appealing feature of this method?

Answer:

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

```
function ortho_cholesky(X)
    L = cholesky(X' * X).L
    Xnew = X / L'
    return Xnew
end
```

```
begin
    Xmatrix = randn(1000, 10)

    ch_res = @benchmark ortho_cholesky($Xmatrix)
    qr_res = @benchmark ortho_qr($Xmatrix)
    gs_res = @benchmark ortho_gs_matrix($Xmatrix)

    println("Orthogonal GS Time: ", mean(gs_res).time)
    println("Ortho Cholesky Time: ", mean(ch_res).time)
    println("Ortho QR Time: ", mean(qr_res).time)
end
```

```
Orthogonal GS Time: 47864.76
Ortho Cholesky Time: 46999.76
Ortho QR Time: 135275.38
```

The Cholesky factorization method is computationally efficient since matrix L is lower triangular and can be more stable because it doesn't accumulate errors across iterations but computes all vectors simultaneously.

(c) Now run `ortho_cholesky` on `testmatrix(m)` for various values of m. You should notice this method to fail for too small values of m (`PosDefException`), because $X^H X$ is numerically no longer positive definite. By computing the eigenspectrum of `X^H X` check exactly what happens as m gets smaller. With this in mind add `ortho_cholesky` to your plot in Task 3 (d) by truncating the range of m appropriately for `ortho_cholesky`. How does it perform in contrast to the other methods we discussed so far?

Answer:

```
1000×10 Matrix{Float64}:
 0.0316228  -0.0721023    0.0434672   …   0.0371892    0.0149649   -0.00740255
 0.0316228   0.00472399  -0.00305279      0.0209644    0.0542603   -0.0251051
 0.0316228  -0.0272906   -0.00561328     -0.0122714   -0.0319019    0.0221593
 0.0316228  -0.00205001   0.00851062      0.0173743   -0.0898082    0.0256584
 0.0316228  -0.0236018   -0.00872023      0.020669    -0.018149     0.0331836
 0.0316228   0.0735455   -0.0286705   …   0.0504369    0.028004     0.00738949
 0.0316228  -0.0116373    0.0323626       0.0411005    0.00956944  -0.0959832
 ⋮                                   ⋱
 0.0316228   0.0256861    0.00718059     -0.0533568    0.0458433    0.0430533
 0.0316228  -0.0299502   -0.0261652   …   0.0140085   -0.0411235    0.0128453
 0.0316228  -0.0298812   -0.00868885     -0.00875839   0.0225573   -0.0140203
 0.0316228  -0.038603     0.0191456       0.00888661  -0.00816195   0.0174899
 0.0316228   0.039895     0.0322935      -0.0010333   -0.0242205    0.00903493
 0.0316228  -0.0395461   -0.0796087      -0.0129505   -0.0347837    0.00417
```

```
ortho_cholesky(testmatrix(10^(-7)))
```

```
[1.03689e-7, 1.10419e-7, 1.13977e-7, 1.1696e-7, 1.25398e-7, 1.29285e-7, 1.3511e-7, 1.389
```

```
let
    testm = testmatrix(10^(-5))
    eigvals(testm' * testm)
end
```

```
[-4.92381e-13, -1.89534e-13, -8.06327e-14, 1.01866e-13, 1.43692e-13, 2.70717e-13, 5.3965
```

```
let
    testm = testmatrix(10^(-8))
    eigvals(testm' * testm)
end
```

We can see that as m gets too small a part of eigenvalues approaches numerical zero. In comparison with other methods, it shows the same performance as `orth_gs` before m gets too small resulting in an ill-conditioned or nearly singular matrix $X^H X$ so that `orth_cholesky` fails.

(d) To avoid the breakdown of cholesky-based orthogonalisation approaches, a typical trick is to apply the cholesky factorisation to $X^H X + \beta * I$ instead of $X^H X$, where $\beta > 0$ is a small constant. With the LL^H factorisation at hand one then forms $\tilde{X} = X \left(L^H\right)^{-1}$ as usual. $\tilde{X}$ is now not yet orthogonal, but its closer than X is. A second application of (unshifted) `ortho_cholesky` to $\tilde{X}$ is then performed to finally obtain a matrix with orthonormal columns. Code up this procedure for $\beta = 1$ as the function `ortho_shift_cholesky` and add it to the plot in Task 3 (d). You should again need to truncate the range of m to avoid a `PosDefException`, but much smaller values for m should be feasible.

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

```
function ortho_shift_cholesky(X; β=1.0)
    L = cholesky(X' * X + β * I).L
    X_tilde = X / L'
    L_tilde = cholesky(X_tilde' * X_tilde).L
    return X_tilde / L_tilde'
end
```

```
1000×10 Matrix{Float64}:
 0.0316228   0.00444152   0.0372043   …  -0.0407069     0.00918084   0.0121572
 0.0316228   0.0187233    0.034438        0.0207581     0.0187209    0.00211363
 0.0316228  -0.0067666   -0.0163115       0.00998315    0.0019642    0.0122323
 0.0316228  -0.00652865  -0.0420049       0.00388271    0.0234498    0.0180698
 0.0316228  -0.0739452    0.00345561     -0.0834229    -0.0466276   -0.0248024
 0.0316228   0.0308162    0.00468279  …   0.0319579     0.0450505    0.0230694
 0.0316228  -0.0403766   -0.0301631       0.0353718     0.027108    -0.0148232
 ⋮                                    ⋱
 0.0316228  -0.00901145  -0.0324094      -0.0356785    -0.0370345   -0.0220257
 0.0316228  -0.00326536   0.00927847  …   0.0359724    -0.0169138    0.0112332
 0.0316228   0.0294623    0.0283473       0.0192486     0.0235156   -0.01275
 0.0316228  -0.0023233   -1.60144e-5     -0.0193533    -0.0330433    0.0381513
 0.0316228   0.00850994  -0.0506248       0.00852178    0.00643847  -0.0215051
 0.0316228   7.5138e-5   -0.0589984      -0.000820681  -0.03352       0.0177101
```

```
ortho_shift_cholesky(testmatrix(10^(-10)))
```

```
begin
    dftk_res = @benchmark ortho_dftk($Xmatrix)
    println("DFTK.ortho! Time: ", mean(dftk_res).time)
end
```

```
DFTK.ortho! Time: 74633.04
```

(e) A more sophisticated implementation of the idea of (d) is provided in the DFTK.jl package as the `DFTK.ortho!` function. Benchmark this function on a `X = randn(1000, 10)` testmatrix and add it to the plot in Task 3 (d) as well.

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

```
ortho_dftk(X) = DFTK.ortho!(copy(X)).X
```

Task 5: Conclusion on orthogonalisation methods

In our discussion so far one main orthogonalisation approach is missing, namely singular value decomposition. Given a matrix $X \in \mathbb{R}^{n \times p}$ the singular value decomposition $X = U \Sigma V^H$ produces two orthogonal matrices $U \in \mathbb{R}^{n \times m}$, and $V \in \mathbb{R}^{m \times p}$ as well as a diagonal matrix $\Sigma \in \mathbb{R}^{m \times m}$ with possibly $m < p$.

(a) Code up such an orthogonalisation routine `ortho_svd` based on Julia's `svd` funciton. Look up its documentation to get more details. Benchmark `ortho_svd` on `X = randn(1000, 10)` and add this function to your plot of Task 3 (d).

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

```
function ortho_svd(X)
    U, _, _ = svd(X)
    return U
end
```

```
begin
    svd_res = @benchmark ortho_svd($Xmatrix)
    println("SVD Time [ns]: ", mean(svd_res).time)
end
```

```
SVD Time [ns]: 105441.44
```

(b) You should now have a good overview of the runtimes and qualities of the orthogonalisation algorithms `ortho_gs_matrix`, `ortho_mgs`, `ortho_qr`, `ortho_svd`, `ortho_cholesky`, `ortho_shift_cholesky` and `ortho_dftk`. With this in mind try to explain the recent popularity of cholesky-based approaches like `ortho_dftk` for othogonalising vectors. If no cholesky-based approach should be chosen (i.e. `ortho_cholesky`, `ortho_shift_cholesky` and `ortho_dftk` are out), which other approach provides in your opinion the best compromise between runtime and accuracy and why?

Answer:

In general, Cholesky-based approach in comparison with other methods is computationally efficient as we can see by comparing the running times. It is also numerically stable for positive definite matrices as we observe from the plots above. However, when Cholesky-based approaches are not available, `ortho_qr` often provides a good compromise between runtime and accuracy among the other approaches.

In the following we will only employ `ortho_dftk` in combination with `lobpcg` as our main diagonalisation routine.

Mixed-precision techniques

Task 6: Low-precision initial guess

Lower precision generally leads to faster computations. Higher precision generally leads to more accurate results. A natural idea is thus to chain computations at multiple precisions to get the best out of both worlds.

(a) Using the `lobpcg` routine in combination with `ortho_dftk` as the orthogonalisation, and the factorisation of the Hamiltonian as the preconditioner, determine roughly the highest accuracy (smallest residual norm) to which the lowest three eigenpairs of `Htest(T)` can be determined if `Float32`, `Float64` and `Double64` are used as the floating-point precision. Since the outcome is strongly dependent on the problem and setup, only an order of magnitude estimate is required here. The best way to obtain this is to repeat your experiments a few times. For `Double64` there is no need to go below `tol=1e-25`.

In each run use the *the same* initial guess for each floating-point precision `T` to make the residual history comparable. However, ensure to convert this guess to the same floating-point precision before passing it to `lobpcg` (e.g. `lobpcg ...; X=Float32.(X), ...`). In one plot, show the residual norm history of the largest of the three eigenpairs (`last.(lobpcg( ... ).residual_norms)`) depending on the chosen floating-point precision `T`.

From your experiments: Roughly at which residual norm is it advisable to switch from one precision to the other in order to avoid impacting the rate of convergence ?

```
4000×4000 SymTridiagonal{Double64, Vector{Double64}}:
  250125.01562462916  -125062.5078125             ⋅            …          ⋅
 -125062.5078125       250125.01562461714  -125062.5078125                ⋅
        ⋅              -125062.5078125       250125.0156246047            ⋅
        ⋅                     ⋅             -125062.5078125               ⋅
        ⋅                     ⋅                    ⋅                      ⋅
        ⋅                     ⋅                    ⋅            …         ⋅
        ⋅                     ⋅                    ⋅                      ⋅
       ⋮                                                ⋱
        ⋅                     ⋅                    ⋅                      ⋅
        ⋅                     ⋅                    ⋅            …         ⋅
        ⋅                     ⋅                    ⋅                      ⋅
        ⋅                     ⋅                    ⋅                      ⋅
        ⋅                     ⋅                    ⋅          -125062.5078125
        ⋅                     ⋅                    ⋅           250125.01562462916
```

```
1 begin
2     H_f32 = Htest(Float32)
3     X_init = randn(size(H_f32, 2), 3)
4 
5     H_f64 = Htest(Float64)
6     H_d64 = Htest(Double64)
7 
8 end
```

```
r_norms32 =
```

```
[1.77066f5, 31.2207, 18.8843, 2.53602, 0.38801, 0.0495919, 0.0278133, 0.037632, 0.026316
```

```
1 r_norms32 = last.( lobpcg(H_f32; X=Float32.(X_init), ortho=ortho_dftk,
  Pinv=InverseMap(factorize(H_f32)), tol=1e-6, maxiter=100).residual_norms)
```

```
 1 2.538e+05 1.771e+05
 2    1.407    31.22
 3    0.605    18.88
 4   -1.146    2.536
 5   -1.151    0.388
 6   -1.151  0.04959
 7   -1.151  0.02781
 8   -1.152  0.03763
 9   -1.151  0.02632
10   -1.151  0.03416
11   -1.151   0.0333
12   -1.151  0.03859
13   -1.151  0.04333
14   -1.152  0.03205
15   -1.152  0.04387
16   -1.151   0.0581
17   -1.152  0.03747
18   -1.152  0.04523
19   -1.151  0.04088
20   -1.151  0.04495
21   -1.151  0.04279
22   -1.151  0.04167
23   -1.151  0.04071
24   -1.151  0.04356
25   -1.151  0.04567
26   -1.151  0.04533
27   -1.151  0.03848
28   -1.151  0.04754
29   -1.151   0.0581
30   -1.152  0.05171
```

```
r_norms64 =
```

```
[1.77066e5, 30.819, 18.7695, 2.49091, 0.386138, 0.0400761, 0.00649154, 0.000817893, 0.000
```

```
1 r_norms64 = last.(lobpcg(H_f64; X=Float64.(X_init), ortho=ortho_dftk,
  Pinv=InverseMap(factorize(H_f64)), tol=1e-12, maxiter=100).residual_norms)
```

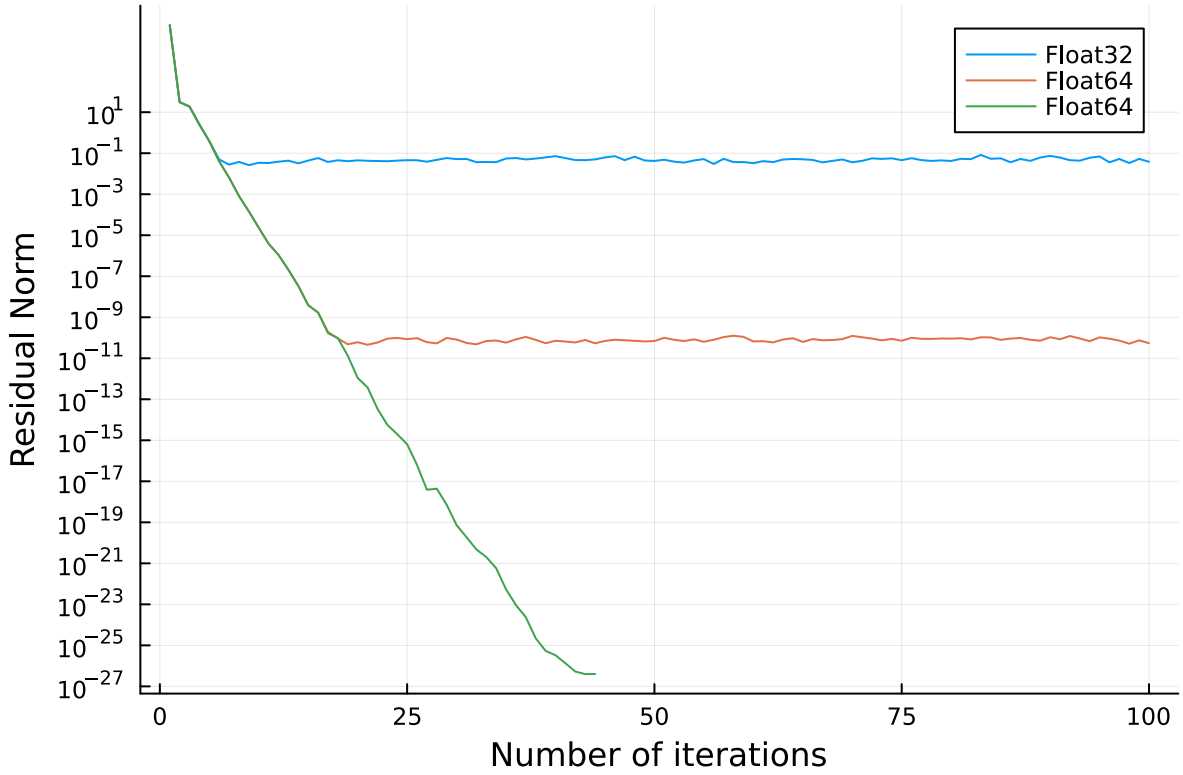
```
 1 2.538e+05 1.771e+05
 2     1.33    30.82
 3   0.6077    18.77
 4   -1.147    2.491
 5   -1.151   0.3861
 6   -1.151  0.04008
 7   -1.151 0.006492
 8   -1.151 0.0008179
 9   -1.151 0.0001453
10   -1.151 2.324e-05
11   -1.151 3.822e-06
12   -1.151 1.09e-06
13   -1.151 2.047e-07
14   -1.151 3.359e-08
15   -1.151 3.86e-09
16   -1.151 1.686e-09
17   -1.151  1.9e-10
18   -1.151 9.198e-11
19   -1.151 4.813e-11
20   -1.151 6.049e-11
21   -1.151 4.551e-11
22   -1.151 5.863e-11
23   -1.151 9.078e-11
24   -1.151 9.803e-11
25   -1.151 8.576e-11
26   -1.151 9.526e-11
27   -1.151 6.102e-11
28   -1.151 5.317e-11
29   -1.151 9.769e-11
30   -1.151 8.212e-11
```

```
r_normsd64 =
```

```
[177065.58575291518, 30.803868505276647, 18.760862680644255, 2.489719216536806, 0.385936
```

```
1 r_normsd64 = last.(lobpcg(H_d64; X=Double64.(X_init), ortho=ortho_dftk,
  Pinv=InverseMap(factorize(H_f64)), tol=1e-25, maxiter=100).residual_norms)
```

```
 1 2.538e+05 1.771e+05
 2     1.33     30.8
 3   0.6077    18.76
 4   -1.147     2.49
 5   -1.151   0.3859
 6   -1.151  0.04006
 7   -1.151 0.006488
 8   -1.151 0.0008177
 9   -1.151 0.0001453
10   -1.151 2.323e-05
11   -1.151 3.82e-06
12   -1.151 1.09e-06
13   -1.151 2.046e-07
14   -1.151 3.358e-08
15   -1.151 3.885e-09
16   -1.151 1.706e-09
17   -1.151 1.688e-10
18   -1.151 9.581e-11
19   -1.151 1.334e-11
20   -1.151  1.1e-12
21   -1.151 3.776e-13
22   -1.151 3.469e-14
23   -1.151 5.656e-15
24   -1.151 1.977e-15
25   -1.151 6.45e-16
26   -1.151 6.255e-17
27   -1.151 3.875e-18
28   -1.151 4.371e-18
29   -1.151 7.151e-19
30   -1.151 7.302e-20
```




```
begin
    plot(r_norms32, label="Float32", yaxis=:log, ylabel="Residual Norm")
    plot!(r_norms64, label="Float64", yticks=10.0 .^ (-27:2:2))
    plot!(r_normsd64, label="Float64", xlabel="Number of iterations")
end
```

As we can see from the plot, a good strategy would be to switch from one precision to the other at the moment when the residual norm stops decreasing significantly. In our case it would be:

- from `Float32` to `Float64`: 1e-2;
- from `Float64` to `Double64`: 1e-11.

```
md"""
As we can see from the plot, a good strategy would be to switch from one precision
to the other at the moment when the residual norm stops decreasing significantly.
In our case it would be:
- from `Float32` to `Float64`: 1e-2;
- from `Float64` to `Double64`: 1e-11.
"""
```

(b) With this outcome in mind code up a routine `solve_discretised(V, Nb, a; n_ep=3, tol32=XXX, tol=1e-6, maxiter=100)`, which first constructs the Hamiltonian using `fd_hamiltonian(V, Nb, a; T=Float32)` and then uses a preconditioned `lobpcg` to solve it for `n_ep` eigenpairs until `tol32` is reached, starting from a random guess. Then it switches to `Float64` as the working precision and continues the iterations by invoking `lobpcg` a second time. Note that for this you will need to recompute the Hamiltonian in `Float64`. Make sure the named tuple returned by the second call to `lobpcg` is also returned by `solve_discretised` itself. In your implementation replace `tol32=XXX` by a sensible default value for `tol32`.

Explain why you have to recompute the Hamiltonian with `T=Float64` instead of simply converting the `Float32` Hamiltonian. Is there a way to avoid computing the Hamiltonian twice ?

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

```
function solve_discretised(V, Nb, a; n_ep=3, tol32=1e-2, tol=1e-6, maxiter=100, verbose=false)
    H_float32 = fd_hamiltonian(V, Nb, a; T=Float32)
    X_32 = randn(Float32, size(H_float32, 2), n_ep)

    lobpcg_result_float32 = lobpcg(H_float32; X=X_32, tol=tol32, verbose=verbose, maxiter=maxiter, Pinv=InverseMap(factorize(H_float32)));

    H_float64 = fd_hamiltonian(V, Nb, a; T=Float64)
    X_64 = Float64.(lobpcg_result_float32.X)

    lobpcg_result_float64 = lobpcg(H_float64; X=X_64, tol=tol, maxiter=maxiter, verbose=verbose, Pinv=InverseMap(factorize(H_float64)))

    return lobpcg_result_float64
end
```

```
lobpcg_result_float64 =
  (λ = [-1.5835, -1.39656, -1.15143], X = 500×3 Matrix{Float64}:                    , eigen
                                          -0.000150807   0.000266352  -0.000259355
```

```
lobpcg_result_float64 = solve_discretised(v_chain, Nb, a; n_ep=3, tol32=1e-2, tol=1e-6, maxiter=100)
```

Converting a matrix from `Float32` to `Float64` change the precision of individual elements but because all computation should be done in a specific format we need to recompute the Hamiltonian once again using `T=Float64`. A way could be to first, compute the Hamiltonian with `T=Float64` and then convert it to the Float32 Hamiltonian.

Bounds on algorithm and arithmetic error

Finally before we try `solve_discretised` in action, we want to develop techniques to estimate the algorithm and arithmetic error of this method.

Task 7: Estimating algorithm and arithmetic error

By construction our `solve_discretised` routine drives the residual of the targeted eigenpairs to zero. If we therefore compute the residual norms

$$\|r_i\| = \|H_i\, x_i - \lambda_i\, x_i\|$$

for the returned eigenpairs (λ_i, x_i) in the same working precision (here `Float64`) this value is smaller than the selected `tol` *by construction*. However this could be an artificial outcome due to the finite-precision arithmetic. To estimate the arithmetic error we will therefore re-compute this residual norm using interval arithmetic.

(a) Write a function `fd_hamiltonian_interval(V, Nb, a)`, which obtains the discretised Hamiltonian in the form of `Float64` intervals enclosing the exact matrix elements (i.e. the elements without arithmetic error). There are multiple ways one could do this. By far the simpliest is to run `fd_hamiltonian` using `T=BigFloat` to get a highly accurate representation of the matrix elements, then convert each element to `Float64` intervals using the function

```
2×2 Matrix{Interval{Float64}}:
 [3.14159, 3.1416]  [2.71828, 2.71829]
 [0.0, 0.0]         [1.0, 1.0]
```

```
let
    values = BigFloat[π e;  # dummy, for illustration here
                      0 1]
    @show values

    interval.(Float64, values)
end
```

```
values = BigFloat[3.14159265358979323846264338327950288419716939937510582097494459230781640628619 2.71828182845904523536028747135266249775724709369995957496696762772407663035355 0.0 1.0]
```

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

```
function fd_hamiltonian_interval(V, Nb, a)
    H_bfloat = fd_hamiltonian(V, Nb, a; T=BigFloat)

    H_interval = interval.(Float64, H_bfloat)

    return H_interval
end
```

(b) Use `fd_hamiltonian_interval` to code up the function `residual_norms_interval(V, λ, X, Nb, a)`, which gets the solution eigenpairs `λ` and `X` in `Float64` as returned in the named tuple of `solve_discretised` and computes their residual norms using interval arithmetic. The return value should be a vector of n intervals if n eigenpairs are passed to the function. Test your function on the result of `solve_discretised` for `v_chain, Nb = 1000, a = 4` and `n_ep=3`. You should obtain narrow intervals with upper and lower bounds around the value chosen for `tol`.

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

```
function residual_norms_interval(V, λ, X, Nb, a)
    H_interval = fd_hamiltonian_interval(V, Nb, a)
    residual_norms =  norm.(eachcol(H_interval * X - X * Diagonal(λ)))

    return residual_norms
end
```

```
[[1.05015e-06, 1.05017e-06], [1.85153e-07, 1.85158e-07], [2.60811e-07, 2.60816e-07]]
```

```
begin
    result = solve_discretised(v_chain, 1000, 4; n_ep=3)
    residual_norms = residual_norms_interval(v_chain, result.λ, result.X, 1000, 4)
end
```

As we can see, this gives us narrow intervals around the value chosen for tol.

Recall that the residual norm itself is one of the main ingredients to obtain an upper bound to the error of the eigenvalue (see the Bauer-Fike and Kato-Temple bounds). Furthermore the intervals obtained from interval arithmetic are guaranteed to enclose the exact result (the result if *exact arithmetic* was employed). Therefore if we denote the exact residual norm by $\|r_i\| > 0$ and the returned intervals from `residual_norms_interval` by $\left[\|r_i\|_{\text{lo}}, \|r_i\|_{\text{hi}}\right]$, we are guaranteed to have $\|r_i\| \in \left[\|r_i\|_{\text{lo}}, \|r_i\|_{\text{hi}}\right]$. A guaranteed upper bound to the residual norm accounting for both algorithm *and* arithmetic error is thus to employ $\|r_i\|_{\text{hi}}$ instead of $\|r_i\|$. From an interval `pi_inter` this can be extracted as such:

```
3.1415926535897936
```

```
let
    pi_inter = interval(π)
    pi_inter.hi
end
```

In turn the width of the interval $\rho_i^{\text{hi}} - \rho_i^{\text{lo}}$. –- also called radius –- provides an upper bound to the arithmetic error in the residual computation, which by the Bauer-Fike bound provides an upper bound to the arithmetic contribution to the eigenvalue error. This width can be obtained as

```
4.440892098500626e-16
```

```
let
    pi_inter = interval(π)
    radius(pi_inter)
end
```

(c) Using the computational setup of (b) estimate a guaranteed upper bound to the residual norms $\|r_i\|$ for each eigenpair use this in conjunction with the Kato-Temple bound to estimate an upper bound for the combined algorithm and arithmetic error of the first eigenvalue. You may assume that `lobpcg` did not miss any eigenpair, i.e. indeed returns approximations to the first three eigenpairs.

(d) Use the width of the residual interval as an estimate for the arithmetic error in the first eigenvalue. In combination with (c) can you clearly say which contribution dominates ?

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

```
function get_upper_bound(result, residual_norms)

    δ1 = max(0, abs(result.λ[1] - result.λ[2]) - residual_norms[2].hi)

    # upper bound for the combined algorithm and arithmetic error
    err_KT_λ1 = residual_norms[1].hi .^2 ./ δ1
end
```

```
guaranteed_upper_bound = [-1.58346, -1.39651, -1.15137]
```

```
guaranteed_upper_bound = [interval(result.λ[i]).hi for i in 1:3]
```

```
alg_arith_ub = 5.899090316099484e-12
```

```
alg_arith_ub = get_upper_bound(result, residual_norms)
```

```
arithm_upper_bound = 4.518195139534021e-12
```

```
# estimate for the arithmetic error in the first eigenvalue
arithm_upper_bound = residual_norms[1].hi - residual_norms[1].lo
```

```
alg_upper_bound = 1.3808951765654635e-12
```

```
alg_upper_bound = alg_arith_ub - arithm_upper_bound
```

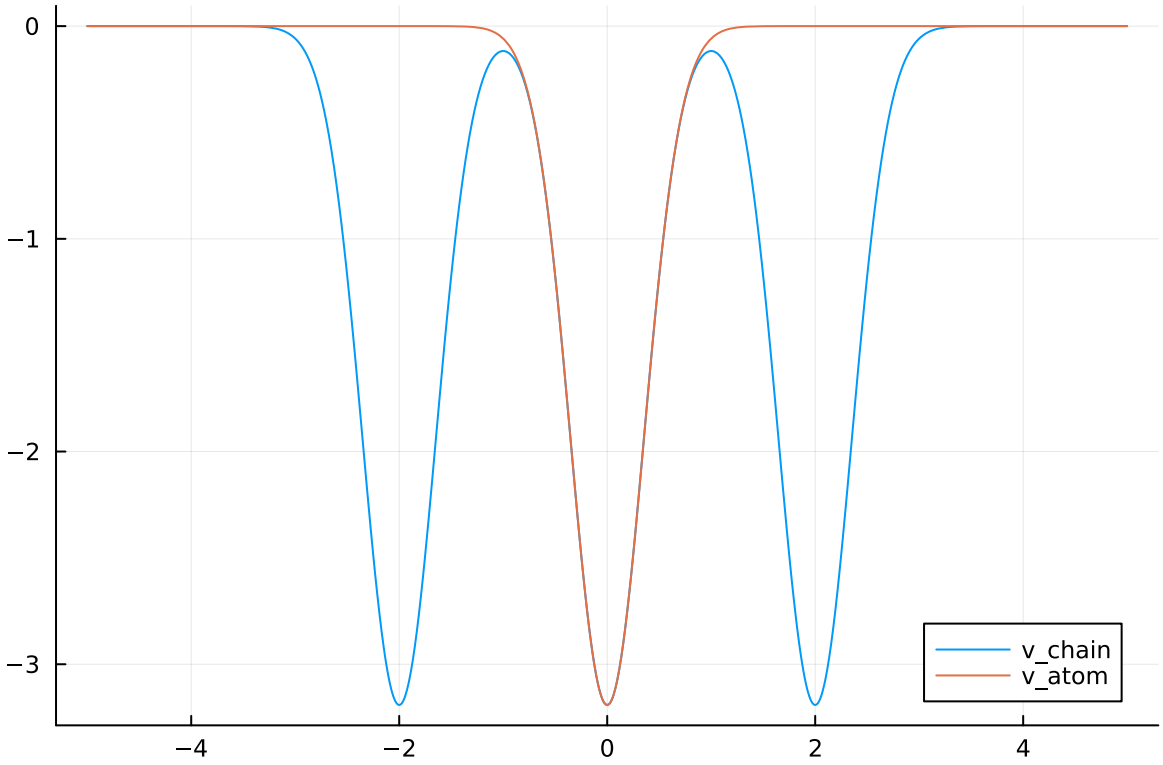
From comparing arithmatic and algorithmic errors it's hard to say clearly which of them dominates. In some examples, we noticed that they both contribute more or less equally.

Computing the effect of tunnelling

Finally we have everything in place to return to our original problem, namely computing the tunnelling energy for the chain of three atoms with

$$v^{\text{chain}}(x) = v^{\text{atom}}(x-2) + v^{\text{atom}}(x) + v^{\text{atom}}(x+2)$$

and v^{atom} being the Gaussian potential with $\alpha = 2$ and $\sigma = 1/4$. Pictorially:




```
let
    p = plot(v_chain, label="v_chain")
    plot!(p, v_atom, label="v_atom")
end
```

Task 8: Putting it all together

(a) Solve both (CL) and (QM) employing `solve_discretised` for the first 3 eigenpairs in each case. Converge your calculations with respect to a and N_b until you obtain the tunnelling energy $\Delta\varepsilon$ to 5 digits.

Hint: A good strategy is to first select a reasonable value for N_b, then keep $h = \frac{2a}{N_b - 1}$ fixed and converged wrt. a, then use that value for a to converge wrt. N_b (by decreasing h), then repeat until the desired tolerance is found.

```
begin
    fixed_a = 4
    found_Nb = 0
    prev = 1
    for Nb in collect(500:200:2600)
        ε_QM = solve_discretised(v_chain, Nb, fixed_a; n_ep=3).λ[1]
        ε_CL = solve_discretised(v_atom, Nb, fixed_a; n_ep=3).λ[1]

        Δε = ε_CL - ε_QM
        println(Δε)
        if abs(Δε - prev) < 1e-6
            found_Nb = Nb
            break
        else
            prev = Δε
        end

    end
end
```

```
0.18785978179246632
0.18786454174393197
0.1878665031082396
0.18786749670132963
```

1100

```
found_Nb
```

```
begin
    found_a = 4
    local prev = 1
    for a in collect(4:1:10)
        ε_QM = solve_discretised(v_chain, found_Nb, a; n_ep=3).λ[1]
        ε_CL = solve_discretised(v_atom, found_Nb, a; n_ep=3).λ[1]

        Δε = ε_CL - ε_QM
        println(Δε)
        if abs(Δε - prev) < 1e-6
            found_a = a
            break
        else
            prev = Δε
        end
    end
end
```

```
0.1878674967015863
0.18938349250544118
0.18942528522798918
0.1894248813723276
```

(1100, 7)

```
found_Nb, found_a
```

h_found = 0.012738853503184714

```
h_found = 2found_a/(found_Nb - 1)
```

```
begin
    ε_QM = solve_discretised(v_chain, found_Nb, found_a; n_ep=3).λ[1]
    ε_CL = solve_discretised(v_atom, found_Nb, found_a; n_ep=3).λ[1]
    println(ε_CL - ε_QM)
end
```

```
0.18942488137241886
```

(b) Employ the Kato-Temple bound employed in Task 7 (c) to verify that the combined algorithm and arithmetic error of $\Delta\varepsilon$ is less than the 3 digits of convergence, i.e. that the algorithm and arithmetic error can be neglected.

2.85617484590579e-11

```
begin
    result_QM = solve_discretised(v_chain, found_Nb, found_a; n_ep=3)
    residual_norms_QM = residual_norms_interval(v_chain, result_QM.λ, result_QM.X,
    found_Nb, found_a)
    upper_bound_QM = get_upper_bound(result_QM, residual_norms_QM)
end
```

```
7.484779029347892e-13
```

```
begin
    result_CL = solve_discretised(v_atom, found_Nb, found_a; n_ep=3)
    residual_norms_CL = residual_norms_interval(v_atom, result_CL.λ, result_CL.X,
    found_Nb, found_a)
    upper_bound_CL = get_upper_bound(result_CL, residual_norms_CL)
end
```

```
0.1894248813723749
```

```
result_CL.λ[1] - result_QM.λ[1]
```

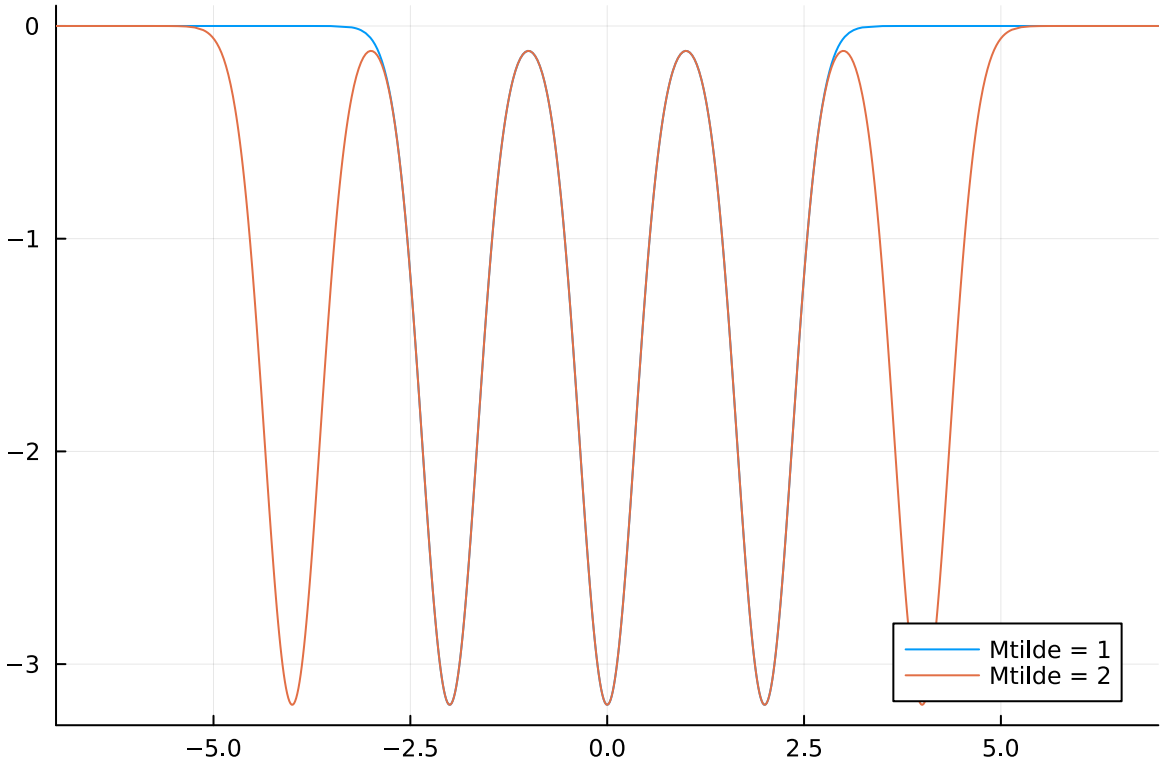
(c) We now generalise the potential to

$$v^{\text{extended chain}}(x) = v^{\text{atom}}(x) + \sum_{i=1}^{\widetilde{M}} v^{\text{atom}}(x-2i) + v^{\text{atom}}(x+2i),$$

i.e. a chain consisting of $2\widetilde{M}+1$ atoms, or in code:

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

```
function v_extended_chain(Mtilde::Integer)
    function evaluate(x)
        accu = v_atom(x)
        for i in 1:Mtilde
            accu += v_atom(x - 2i) + v_atom(x + 2i)
        end
        accu
    end
end
```




```
let
    plot( v_extended_chain(1), xlims=(-7, 7), label="Mtilde = 1")
    plot!(v_extended_chain(2), label="Mtilde = 2")
end
```

Assume that a needs to be chosen such that the same amount of empty space between the rightmost / leftmost atom and the domain boundary is required as in Task 8 (a). Similarly assume that the same grid spacing $h = \frac{2a}{N_b - 1}$ as in Task 8 (a) is sufficient for convergence to 3 digits in $\Delta\varepsilon$.

Within this setup investigate the limit $\widetilde{M} \to \infty$ numerically. Does the tunneling energy converge? Justify your argument from a physical perspective. If it does converge, determine the value to 2 digits, if not determine the kind of divergence (e.g. rough dependency on $\widetilde{M}$ determined from a plot.).

The numerical scheme to achieve this needs a little care as solving the discretised QM problem numerically gets more and more challenging when $\widetilde{M}$ increases. In a naive approach your tunnelling energy might oscillate artifically. You can remedy this by not starting from a random initial guess, but by starting from the solution you obtained at $\widetilde{M} - 1$. Sometimes adding in a little random noise (e.g. `randn` but scaled by a small constant) can be helpful (Why ?). Describe and justify your developed numerical scheme in a few sentences.

Extra hint: How many times do you need to solve (CL) ?

```
0.18942488137236912
```

```
begin
    M = 1
    QM = solve_discretised(v_extended_chain(M), found_Nb, found_a; n_ep=3).λ[1]
    CL = solve_discretised(v_atom, found_Nb, found_a; n_ep=3).λ[1]

    tun_energy = CL - QM
end
```

```
-1.3956308319423587
```

```
CL
```

```
-1.5850557133147278
```

```
QM
```

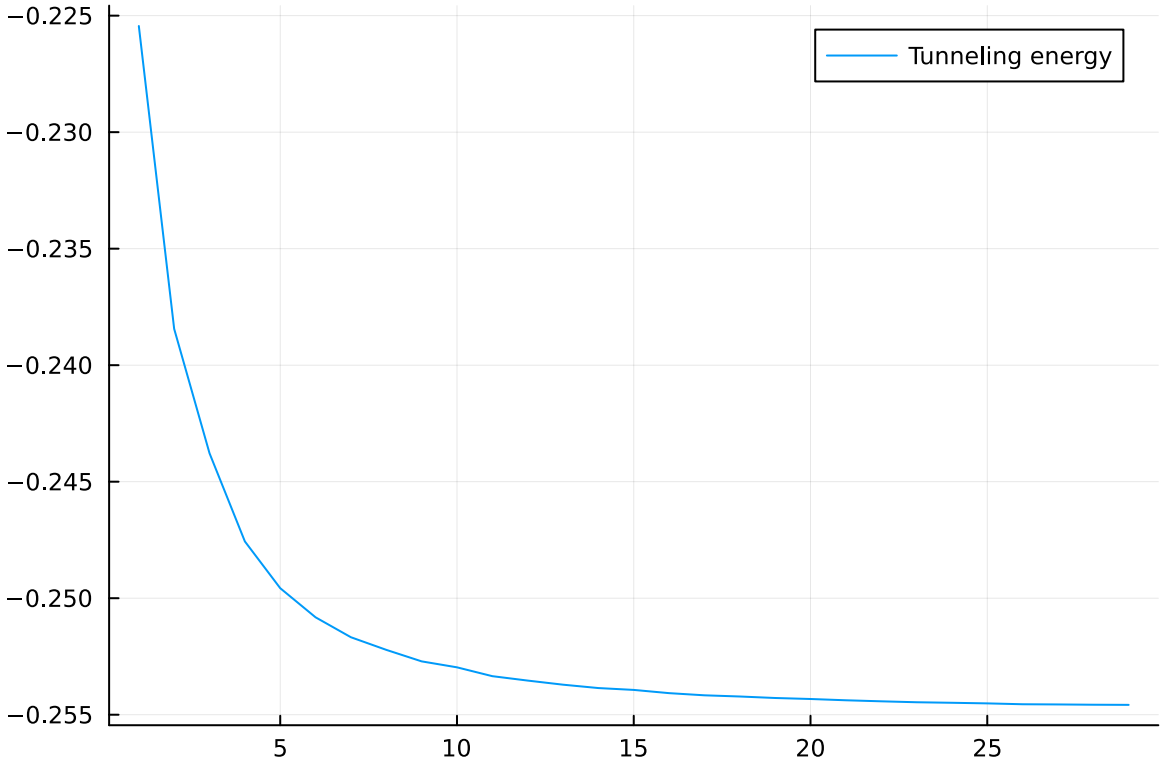
```
-1.5850557133146639
```

```
solve_discretised_QM(v_extended_chain(1), found_Nb, found_a; n_ep=3).λ[1]
```

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

```
function solve_discretised_QM(V, Nb, a; n_ep=3, X_32=randn(Float32, Nb, n_ep), tol32=1e-2, tol=1e-6, maxiter=100, verbose=false)
    H_float32 = fd_hamiltonian(V, Nb, a; T=Float32)

    lobpcg_result_float32 = lobpcg(H_float32; X=Float32.(X_32), tol=tol32, verbose=verbose, maxiter=maxiter, Pinv=InverseMap(factorize(H_float32)));

    H_float64 = fd_hamiltonian(V, Nb, a; T=Float64)
    X_64 = Float64.(lobpcg_result_float32.X)

    lobpcg_result_float64 = lobpcg(H_float64; X=X_64, tol=tol, maxiter=maxiter, verbose=verbose, Pinv=InverseMap(factorize(H_float64)))

    return lobpcg_result_float64
end
```

```
begin
    t_energy = Float64[]
    m = 1
    local a = 2 * m + 2
    local Nb = round(Int64, 2 * a / h_found) + 1

    qm_res = solve_discretised_QM(v_extended_chain(m), Nb, a; n_ep=3)

    X_prev = qm_res.X

    for m in 2:1:30
        a = 2 * m + 2
        Nb = round(Int64, 2 * a / h_found) + 1

        noise = 0.001 * randn(Float64, size(X_prev))
        X_prev = X_prev + noise

        n = round(Int64, 0.5 * (Nb - size(X_prev)[1]))
        padding = zeros(Float64, n, 3)
        X_prev = vcat(padding, X_prev, padding)

        res = solve_discretised_QM(v_extended_chain(m), Nb, a; X_32=Float32.
        (X_prev), n_ep=3)
        X_prev = res.X
        QM = res.λ[1]

        push!(t_energy, QM - CL)
    end
end
```




```
1 plot(t_energy, label="Tunneling energy")
```

```
-0.25457678510237547
```

```
1 t_energy[end]
```

By changing the value of $\widetilde{M}$ we can see that tunneling energy is converging, in our case to -0.25.

In optimization procedures such as lobpcg, there is a possibility that the algorithm will converge to a local minimum if the initial guess is near a stationary point. Adding small random noise helps the algorithm move in different directions and not get stuck in local minima.

Task 9: Group atmosphere and distribution of work

Each team member individually should provide 4-5 sentences on his role in the project and the overall experience of working on the project as a group. We will open a separate task on moodle for you to return this feedback there, if you prefer.

Some questions you should address:

- Which tasks and subtasks (*(a)*, *(b)*, etc.) of the project did you mostly work on ?
- How did you decide within the group to distribute the workload as such ?
- In your opinion did each group member contribute equally to the project ?
- Where could your specific expertise and background from your prior studies contribute most to the project and the exercises we did earlier in the semester?
- Can you pinpoint aspects about your team members' study subject, which are relevant to the course (either the exercises or this project), which you learned from them in your discussions ?

** Anna **

- 3, 4, 7, 8
- We collectively discussed our preferences and decided to distribute the workload accordingly.
- The goal was always to separate the number of tasks equally since individual strengths and interests play a role in specific tasks.
- Applying mathematical concepts to problem-solving and coding tasks.
- From Olivier, who is from Material Science program I learned some physics-related insights into our project.

```
md"""
** Anna **

- 3, 4, 7, 8
- We collectively discussed our preferences and decided to distribute the workload accordingly.
- The goal was always to separate the number of tasks equally since individual strengths and interests play a role in specific tasks.
- Applying mathematical concepts to problem-solving and coding tasks.
- From Olivier, who is from Material Science program I learned some physics-related insights into our project.
"""
```

Olivier

- *Which tasks and subtasks ((a), (b), etc.) of the project did you mostly work on ?*

I was in charge of tasks 1-2 and 5-6.

- *How did you decide within the group to distribute the workload as such ?*

We thought that 2 exercises per person at both both begining and at the end of the project would be a good way of splitting the workload, ensuring that we have both been acquainted with all parts of the project.

- *In your opinion did each group member contribute equally to the project ?*

On top of doing her parts, Anna has done more work because she helped me (re)write my parts to have cleaner code and checked that everything made sense Whereas I just re-read what she had done without changing much (maybe a bit on the plots).

- *Where could your specific expertise and background from your prior studies contribute most to the project and the exercises we did earlier in the semester?*

To be honest I am not sure anything we have seen in the mandatory material science background gave me an edge over someone that studied math/ computational science like Anna did, maybe on the solid-states physics question in task 1?

- *Can you pinpoint aspects about your team members' study subject, which are relevant to the course (either the exercises or this project), which you learned from them in your discussions ?*

It was interesting to work with Anna because she has a lot more experience so I can take what she codes as an example to write cleaner code. I also think her mathematical background made her more rigorous than what I am usually used to working with people from engineering.

```
md"""
**Olivier**

- _Which tasks and subtasks ((a), (b), etc.) of the project did you mostly work on
?_
I was in charge of tasks 1-2 and 5-6.

- _How did you decide within the group to distribute the workload as such ?_
We thought that 2 exercises per person at both both begining and at the end of the
project would be a good way of splitting the workload, ensuring that we have both
been acquainted with all parts of the project.

- _In your opinion did each group member contribute equally to the project ?_
On top of doing her parts, Anna has done more work because she helped me (re)write
my parts to have cleaner code and checked that everything made sense Whereas I just
re-read what she had done without changing much (maybe a bit on the plots).

- _Where could your specific expertise and background from your prior studies
contribute most to the project and the exercises we did earlier in the semester?_
To be honest I am not sure anything we have seen in the mandatory material science
background gave me an edge over someone that studied math/ computational science
like Anna did, maybe on the solid-states physics question in task 1?

- _Can you pinpoint aspects about your team members' study subject, which are
relevant to the course (either the exercises or this project), which you learned
from them in your discussions ?_
It was interesting to work with Anna because she has a lot more experience so I can
take what she codes as an example to write cleaner code. I also think her
mathematical background made her more rigorous than what I am usually used to
working with people from engineering.

"""
```