Sheet 2 solution sketch

```
begin
    using Plots
    using LinearAlgebra
end
```

```
begin
    D  = 20.0
    ω  = 2.25
    x0 = 0.0
end;
```

Exercise 1

(a) Clearly

$$\exp\left(-\frac{\omega}{\sqrt{2D}}\left(x-x_0\right)\right) = 1 - \frac{\omega}{\sqrt{2D}}\left(x-x_0\right) + O((x-x_0)^2)$$

such that

$$\begin{aligned} V^M(x) &= D\left(\frac{\omega}{\sqrt{2D}}\left(x-x_0\right)\right)^2 + O((x-x_0)^4) \\ &= \frac{1}{2}\omega^2(x-x_0)^2 + O((x-x_0)^4) \end{aligned}$$

as desired.

(b)

Use a standard Gaussian integral.

The normalised function is

$$\varphi_0^H(x) = \sqrt[4]{\frac{\omega}{\pi}}\exp\left(-\frac{\omega x^2}{2}\right).$$

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

```
phinorm(x) = (ω/π)^(1/4) * exp(-ω * x^2 / 2)
```

Exercise 2

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

```
function fd_laplacian(N, a; T=Float64)
    h = 2a / T(N+1)
    diagonal = -2ones(T, N) ./ h^2
    side_diagonal = ones(T, N-1) ./ h^2
    SymTridiagonal(diagonal, side_diagonal)
end
```

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

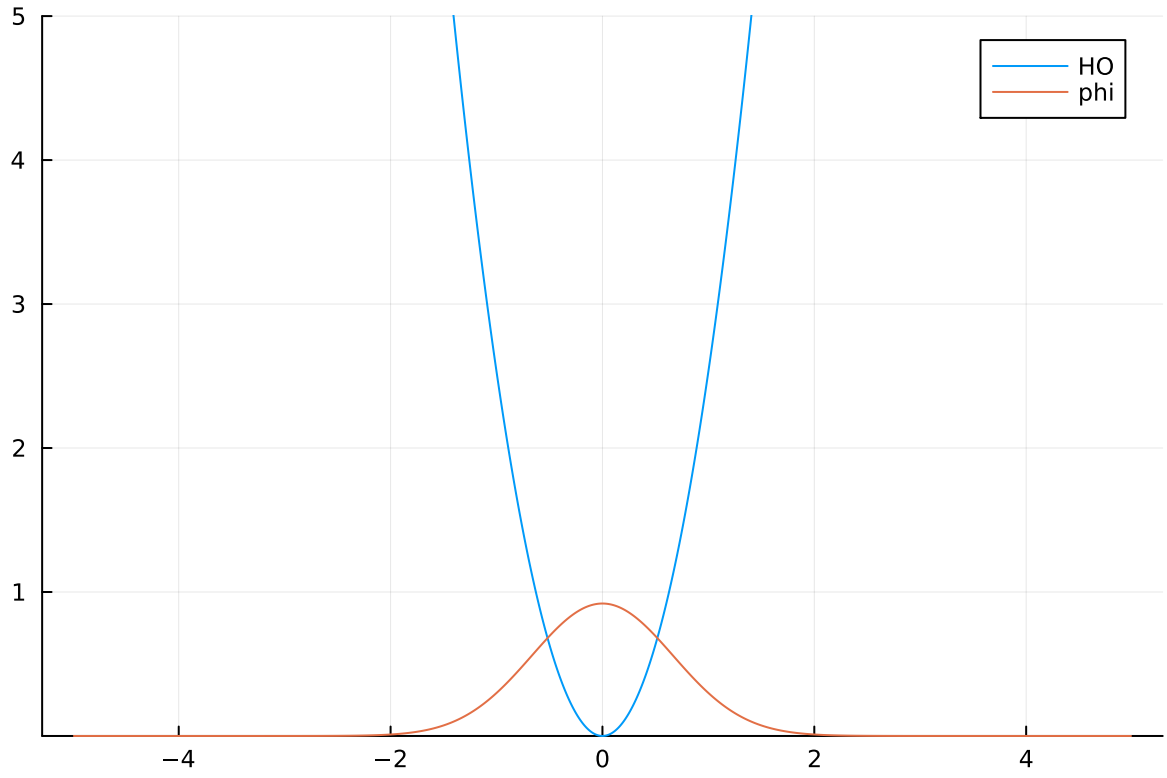
```
function coordinates(N, a; T=Float64)
    x = range(-T(a), T(a), length=N+2)
    x[2:end-1]  # Discard endpoints with Dirichlet condition
end
```

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

```
Vho(x) = 0.5 * ω^2 * (x-x0)^2
```

```
begin
    a = 5
    N = 1000
end;
```

(a)




```
let
    c = coordinates(N, a)
    p = plot(c, Vho.(c), label="HO", ylims=(-Inf, 5))
    p = plot!(p, c, phinorm.(c), label="phi")
end
```

(b)

```
begin
    Hho = -0.5 * fd_laplacian(N, a) + Diagonal(Vho.(coordinates(N, a)))
    λ, X = eigen(Hho)
end;
```

(c)

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

```
Eho(n) = (n + 0.5) * ω
```

We note (using the Trapezoidal rule and Dirichlet boundary conditions)

$$\int_{\mathbb{R}} f(x)dx \approx \int_{\Omega} f(x)dx \approx \sum_{i=1}^{N} f(x_i)h$$

```
discretized_l2_norm (generic function with 1 method)
```

```
function discretized_l2_norm(fx, a)
    # Approximate L^2([-a, a]) norm of f discretized on a uniform grid of N points
    # within the the interval [-a, a].
    N = length(fx)
    h = 2a / (N+1)
    sqrt(sum(abs2, fx) * h)  # = norm(x) * sqrt(h)
end
```

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

```
function properly_normalise(fx)
    # Properly normalise the numerical eigenfunctions as discrete
    # representations of L^2(R) functions
    L2norm = discretized_l2_norm(fx, a)
    fx ./ L2norm
end
```

```
(error_λ = 1.57889e-5, error_X1 = 1.03294e-5)
```

```
let
    error_λ = abs(λ[1] - Eho(0))

    X1 = properly_normalise(X[:, 1])
    error_X1 = discretized_l2_norm(X1 - phinorm.(coordinates(N, a)), a)

    (; error_λ, error_X1)
end
```

(d)

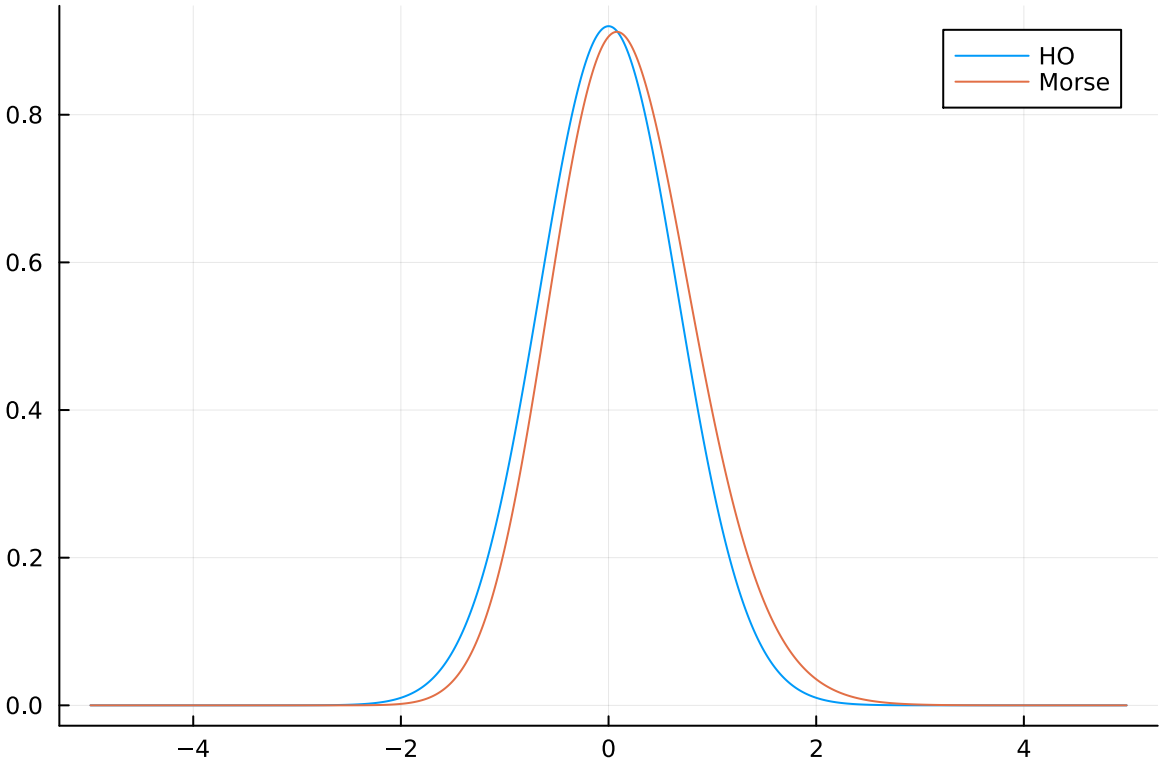
```
Vm (generic function with 1 method)
```

```
begin
    Hm = -0.5 * fd_laplacian(N, a) + Diagonal(Vm.(coordinates(N, a)))
    λm, Xm = eigen(Hm)
end;
```

```julia
begin
    χ = ω / (4D)
    n = 0

    println("Numerical  deviation: ", abs(λm[1] - λ[1]))
    println("Analytical deviation: ", ω * χ * (n + 0.5)^2)
end
```

```
Numerical  deviation: 0.015819724588810535
Analytical deviation: 0.0158203125
```




```julia
let
    X1  = properly_normalise(X[:, 1] )
    Xm1 = properly_normalise(Xm[:, 1])

    c = coordinates(N, a)
    plot(c, X1, label="HO")
    plot!(c, Xm1, label="Morse")
end
```

Exercise 3

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

```
function inverse_power_method(A; u=randn(eltype(A), size(A, 2)),
                                 tol=1e-6, maxiter=500)
    norm_Δu = NaN
    Afac = factorize(A)  # Factorise A to make A \ x more economical
    for i in 1:maxiter
        u_prev = u
        u = Afac \ u
        normalize!(u)
        norm_Δu = norm(u - u_prev)
        norm_Δu < tol && break
    end
    μ = dot(u, A, u)
    norm_Δu ≥ tol && @warn "Inverse power not converged $norm_Δu"
    (; μ, u)
end
```

```
Enter cell code...
```

(a) We saw before, that the error in the $l^2(\mathbb{R}^N)$-norm used by our linear algebra and the error in the $L^2(\mathbb{R})$-norm that we target are related by a factor of $\sqrt{h}$. Therefore we need

```
9.995003746877732e-6
```

```
begin
    h = 2a / (N+1)
    tol = sqrt(h) * 1e-4
end
```

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

```
function harmonic_hamiltonian(N, a; T=Float64)
    -0.5 * fd_laplacian(N, a; T) + Diagonal(Vho.(coordinates(N, a; T)))
end
```

```
(error_λ = 1.57889e-5, error_X1 = 1.03309e-5)
```

```
let
    # Test that I did not mess up
    T = Float64
    H = harmonic_hamiltonian(N, a; T)
    λpow, Xpow = inverse_power_method(H)
    error_λ = abs(λpow - Eho(0))

    X1 = properly_normalise(Xpow)
    phivalues = phinorm.(coordinates(N, a))
    signfix = sign(X1[1] ./ phivalues[1])
    error_X1 = discretized_l2_norm(X1 .- signfix .* phivalues, a)

    (; error_λ, error_X1)
end
```

Float32 or Float64 makes negligible difference here

(b)

We are looking at the following sequence of ground-state energy approximations:

μ0: the analytical energy of the Morse

μ1: the analytical energy of the HO

μ2: the min eigval of the discretized HO matrix (e.g. eigen + high accuracy)

μ3: the min eigval of the HO with power method finite iterations

μ4: the min eigval of the HO with power method and Float32

And decompose the total error (assuming these are all the relevant errors) as

$$\mu_0 - \mu_4 = (\mu_0 - \mu_1) + (\mu_1 - \mu_2) + (\mu_2 - \mu_3) + (\mu_3 - \mu_4)$$

μ0 = 1.1091796875

```
μ0 = let
    n = 0
    χ = ω / 4D
    ω * (n + 0.5) - ω * χ * (n + 0.5)^2
end
```

μ4 = 1.1249841413149477

```
μ4 = inverse_power_method(harmonic_hamiltonian(N, a; T=Float32)).μ
```

total_error = -0.015804453814947772

```
total_error = μ0 - μ4
```

```
begin
    μ1 = Eho(0)

    # TODO MFH: use BigFloat with eigen for this !
    H_exact = harmonic_hamiltonian(N, a; T=Float64)
    μ2 = eigen(H_exact).values[1]

    μ3 = inverse_power_method(H_exact).μ
end;
```

model_error = -0.015820312500000044

```
model_error = μ0 - μ1
```

discretization_error = 1.5788936367000517e-5

```
discretization_error = μ1 - μ2
```

algorithm_error = 4.1742165279856636e-12

```
algorithm_error = μ2 - μ3
```

```
arithmetic_error = 6.97445110553474e-8
```

```
arithmetic_error = μ3 - μ4
```

And we verify the triangle inequality of our error estimate:

```
(0.0158045, 0.0158362)
```

```
abs(total_error), abs(model_error) + abs(discretization_error) +
abs(algorithm_error) + abs(arithmetic_error)
```

```
true
```

```
abs(total_error) ≤ abs(model_error) + abs(discretization_error) +
abs(algorithm_error) + abs(arithmetic_error)
```

(c)

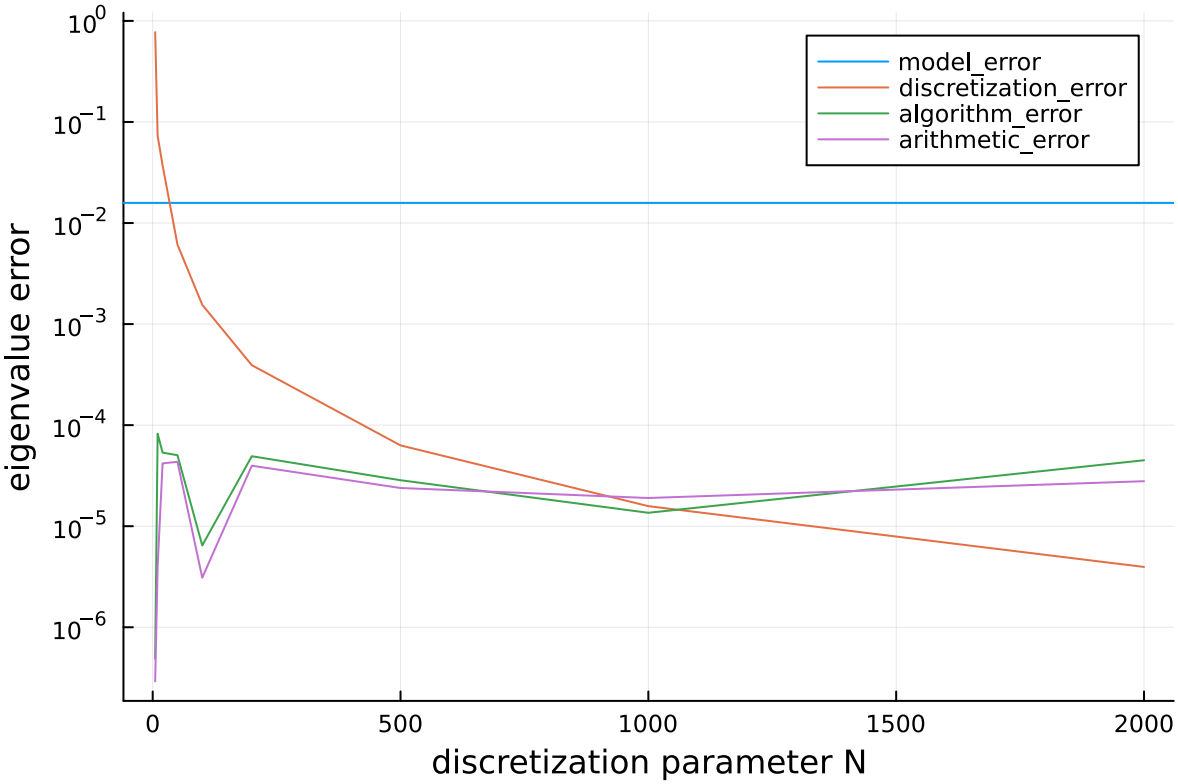
```
estimate_abs_errors (generic function with 2 methods)
```

```
function estimate_abs_errors(N, tol=1e-3)
    H_exact = harmonic_hamiltonian(N, a; T=Float64)
    μ2 = eigen(H_exact).values[1]

    μ3 = inverse_power_method(H_exact; tol).μ

    H_float32 = harmonic_hamiltonian(N, a; T=Float32)
    μ4 = inverse_power_method(H_float32; tol).μ

    Dict(
        :discretization_error=>abs(μ1 - μ2),
        :algorithm_error=>abs(μ2 - μ3),
        :arithmetic_error=>abs(μ3 - μ4),
        :total_error=>abs(μ1 - μ4),
    )
end
```



We see that for this problem a meager N=50 points and a high tolerance=1e-2 already undercut the model error.

Appendix on BigFloat. Option 1: using BigFloat to converge the algorithm+arithmethic error even further and isolate the discretization error

```
1 using GenericLinearAlgebra
```

```
H_exact_big =
1000×1000 SymTridiagonal{BigFloat, Vector{BigFloat}}:
 10083.0  -5010.0       ⋅        ⋅        ⋅   …       ⋅        ⋅        ⋅        ⋅
 -5010.0  10082.8  -5010.0       ⋅        ⋅           ⋅        ⋅        ⋅        ⋅
      ⋅   -5010.0  10082.5  -5010.0       ⋅           ⋅        ⋅        ⋅        ⋅
      ⋅        ⋅   -5010.0  10082.3  -5010.0          ⋅        ⋅        ⋅        ⋅
      ⋅        ⋅        ⋅   -5010.0  10082.0          ⋅        ⋅        ⋅        ⋅
      ⋅        ⋅        ⋅        ⋅   -5010.0  …       ⋅        ⋅        ⋅        ⋅
      ⋅        ⋅        ⋅        ⋅        ⋅           ⋅        ⋅        ⋅        ⋅
     ⋮                                        ⋱
      ⋅        ⋅        ⋅        ⋅        ⋅           ⋅        ⋅        ⋅        ⋅
      ⋅        ⋅        ⋅        ⋅        ⋅   …  -5010.0       ⋅        ⋅        ⋅
      ⋅        ⋅        ⋅        ⋅        ⋅      10082.3  -5010.0       ⋅        ⋅
      ⋅        ⋅        ⋅        ⋅        ⋅      -5010.0  10082.5  -5010.0       ⋅
      ⋅        ⋅        ⋅        ⋅        ⋅           ⋅   -5010.0  10082.8  -5010.0
      ⋅        ⋅        ⋅        ⋅        ⋅           ⋅        ⋅   -5010.0  10083.0
```

```
1 H_exact_big = harmonic_hamiltonian(N, a; T=BigFloat)
```

```
res =
  (μ = 1.12498, u = [1.25945e-14, 2.53446e-14, 3.84066e-14, 5.19393e-14, 6.61059e-14, 8.107
```

```
1 res = inverse_power_method(H_exact_big, tol=1e-50, maxiter=1000)
```

```
1.12498421105913246245319544587796587192900223907741831682669822726829242909261 5
```

```
1 res.μ
```

```
1.5788940867537546804554122034128070997760922581683173301772731707570907384986 22e-05
```

```
abs(μ1 - res.μ)  # The isolated discretization error approximated with BigFloat and high-precision power_method
```

Option 2 (not recommended): BigFloat + GenericLinearAlgebra.jl. Caution: N=100 still doable in seconds, but high N might become impractical for this example.

```
# eigen(H_exact_big) # TODO how long does this take?
```

```
(0.00155301, 0.00153722)
```

```
let
    N = 100
    H = harmonic_hamiltonian(N, a; T=BigFloat)
    μ = eigen(H).values[1]
    abs(μ1 - μ), abs(μ - μ2)
end
```