Solve the GPE in a 1D parabolic trap

Ashton Bradley

April 15, 2019

1 Introduction

In this simple example we find a ground state of the Gross-Pitaevskii equation in a harmonic trap.

The mean field order parameter evolves according to

$$i\hbar\frac{\partial\psi(x,t)}{\partial t} = \left(-\frac{\hbar^2\partial_x^2}{2m} + V(x,t) + g|\psi(x,t)|^2\right)\psi(x,t)$$

2 Loading the package

First, we load some useful packages.

```
using Plots, LaTeXStrings
gr(fmt="png",legend=false,titlefontsize=12,size=(600,300),grid=false,transpose=true,colorbar=false);
```

Now load `FourierGPE`

```
using FourierGPE
```

Let's define a convenient plot function

```
function showpsi(x,ψ)
    p1 = plot(x,abs2.(ψ))
    xlabel!(L"x/a_x");ylabel!(L"|\psi|^2")
    p2 = plot(x,angle.(ψ))
    xlabel!(L"x/a_x");ylabel!(L"\textrm{phase}(\psi)")
    p = plot(p1,p2,layout=(2,1),size=(600,400))
    return p
end
```

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

3 User parameters

We reserve a place for user parameters.

```
@with_kw mutable struct Params <: UserParams @deftype Float64
    # user parameters:
    κ = 0.1
end
par = Params();
```

Let's set the system size, and number of spatial points

```
L = (40.0,)
N = (512,)
μ = 25.0
```

```
25.0
```

Now we need to initialize the simulation object and the transforms

```
sim = Sim(L,N,par)
@pack! sim = μ
@unpack_Sim sim;
```

3.1 Declaring the potential

Let's define the trapping potential.

```
import FourierGPE.V
V(x,t) = 0.5*x^2
```

```
V (generic function with 3 methods)
```

We only require that it is a scalar function because alter we will evaluate it using a broadcasted dot-call.

4 Initial condition

Let's define a useful Thomas-Fermi wavefunction

```
ψ0(x,μ,g) = sqrt(μ/g)*sqrt(max(1.0-V(x,0.0)/μ,0.0)+im*0.0)
x = X[1];
```

The initial state is now created as

```
ψi = ψ0.(x,μ,g)
ϕi = kspace(ψi,sim)
@pack! sim = ϕi;
```

5 Evolution in k-space

The `FFTW` library is used to evolve the Gross-Pitaevskii equation in k-space

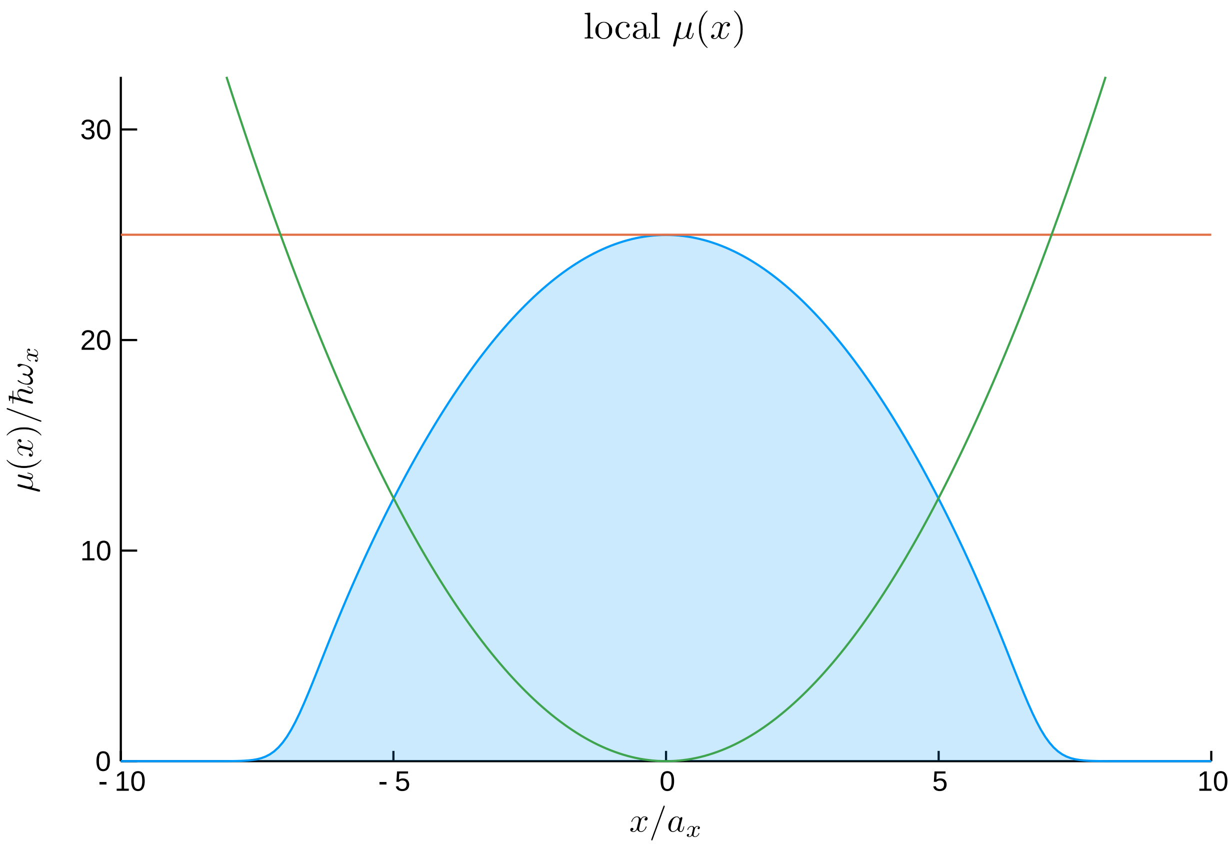
```
sol = runsim(sim);
```

```
0.924232 seconds (1.82 M allocations: 82.970 MiB, 4.23% gc time)
```

Here we save the entire solution as a single variable `sol`.

Let's have a look at the final state and verify we have a ground state

```
ϕg = sol[end]
ψg = xspace(ϕg,sim)
p=plot(x,g*abs2.(ψg),fill=(0,0.2))
plot!(x,one.(x)*μ)
plot!(x,V.(x,0.0))
xlims!(-10,10); ylims!(0,1.3*μ)
title!(L"\textrm{local}\; \mu(x)")
xlabel!(L"x/a_x"); ylabel!(L"\mu(x)/\hbar\omega_x")
plot(p)
```



The initial Thomas-Fermi state has been evolved for a default time $t = 2/\gamma$ which is a characteristic damping time for the dissipative system with dimensionless damping γ. The solution will approch the ground state satisfying $L\psi_0 = \mu\psi_0$ on a timescale of order $1/\gamma$. The figure shows a smooth density profile and a completely homogeneous phase profile over the region of finite atomic density, as required for the ground state. The indeterminate phase evident at large $|x|$ is unimportant.

5.1 Default simulation parameters

The default parameters are given in the declaration of `Sim`, which allows parameter interdependence. The struct `Sim` is declared as:

```
@with_kw mutable struct Sim{D} <: Simulation{D} @deftype Float64
    L::NTuple{D,Float64}
    N::NTuple{D,Int64}
    μ = 15.0
    g = 0.1
    γ = 0.5; @assert γ >= 0.0
    ti = 0.0
    tf = 2/γ
    Nt::Int64 = 200
    t::LinRange{Float64} = LinRange(ti,tf,Nt)
    ϕi::Array{Complex{Float64},D} = zeros(N...) |> complex
    alg::OrdinaryDiffEq.OrdinaryDiffEqAdaptiveAlgorithm = Tsit5()
    reltol::Float64 = 1e-6
    params::UserParams # optional parameters
    X::NTuple{D,Array{Float64,1}} = xvecs(L,N)
    K::NTuple{D,Array{Float64,1}} = kvecs(L,N)
    espec::Array{Complex{Float64},D} = 0.5*k2(L,N)
    flags::UInt32 = FFTW.MEASURE
    T::TransformLibrary = makeT(X,K;flags=flags)
end
```

where we see a set of default parameters, and then some useful transform fields built using the parameters. Note that the transforms have to be built after building `X,K`.

6 Dark soliton in harmonically trapped system

We found a ground state by imaginary time propagation. Now we can impose a phase and density imprint consistent with a dark soliton. We will use the solution for the homogeneous system, which will be a reasonable approximation if we impose it on a smooth background solution.

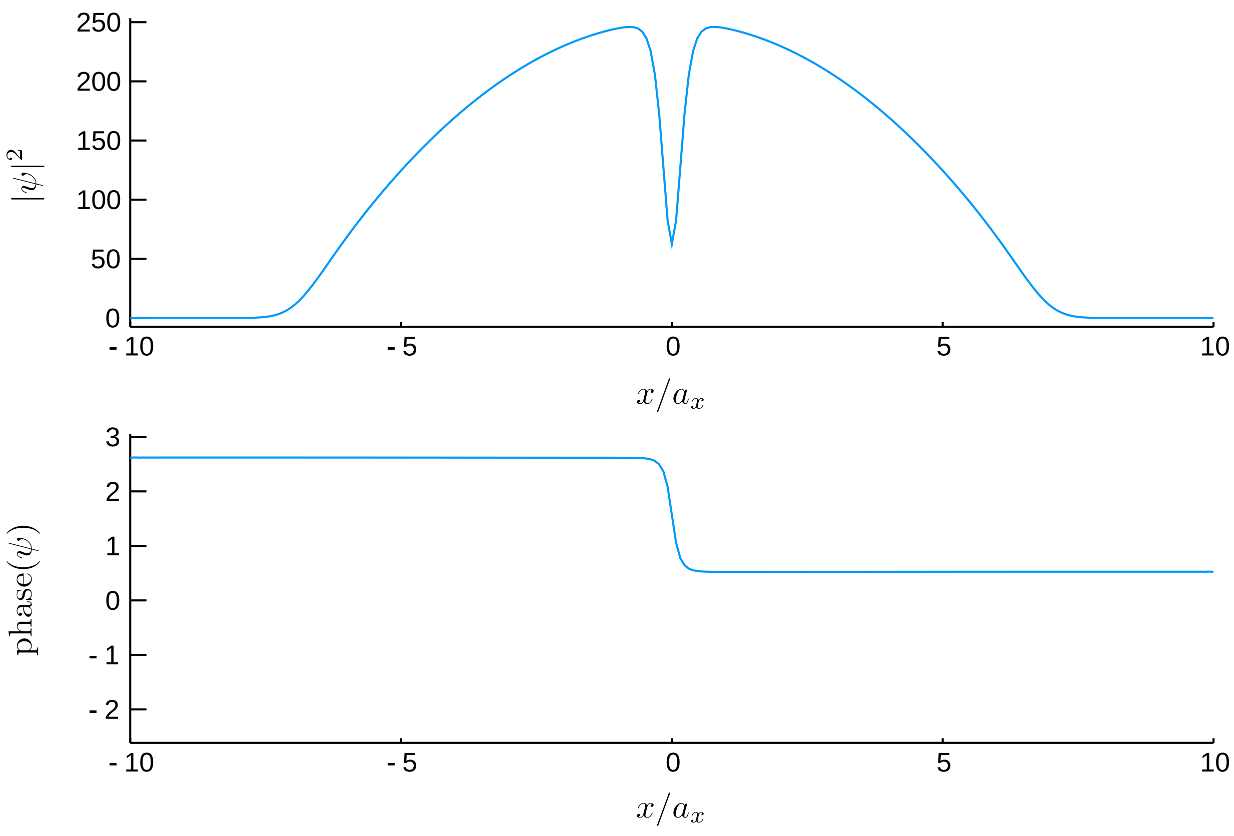
6.1 Imprinting a dark soliton

```
ψf = xspace(sol[end],sim)
c = sqrt(μ)
ξ = 1/c
v = 0.5*c
xs = 0.
f = sqrt(1-(v/c)^2)
```

```
0.8660254037844386
```

Soliton speed is determined by depth and local healing length. Start at $xs = 0.0$

```
ψs = ψf.*(f*tanh.(f*(x .-xs)/ξ).+im*v/c);
showpsi(x,ψs)
xlims!(-10,10)
```



6.2 Initilize Simulation

We can recycle our earlier parameter choices, modifying the damping and simulation timescale

```
γ = 0.0
tf = 8*pi/sqrt(2); t = LinRange(ti,tf,Nt)
dt = 0.01π/μ
simSoliton = Sim(sim;γ=γ,tf=tf,t=t)
ϕi = kspace(ψs,simSoliton)
@pack! simSoliton = ϕi
@unpack_Sim simSoliton;
```

In doing so, we have to specify the dimension of the simulation in this case (an improved constructor needed).

6.3 Solve equation of motion

As before, we specify the initial condition in momentum space, and evolve

```
sols = runsim(simSoliton);
```

```
5.261948 seconds (320 allocations: 1.757 MiB)
```

6.4 View the solution using Plots

Plots allows easy creation of an animated gif, as in the runnable example code below.

```
ϕf = sols[end-4]
ψf = xspace(ϕf,simSoliton)
```

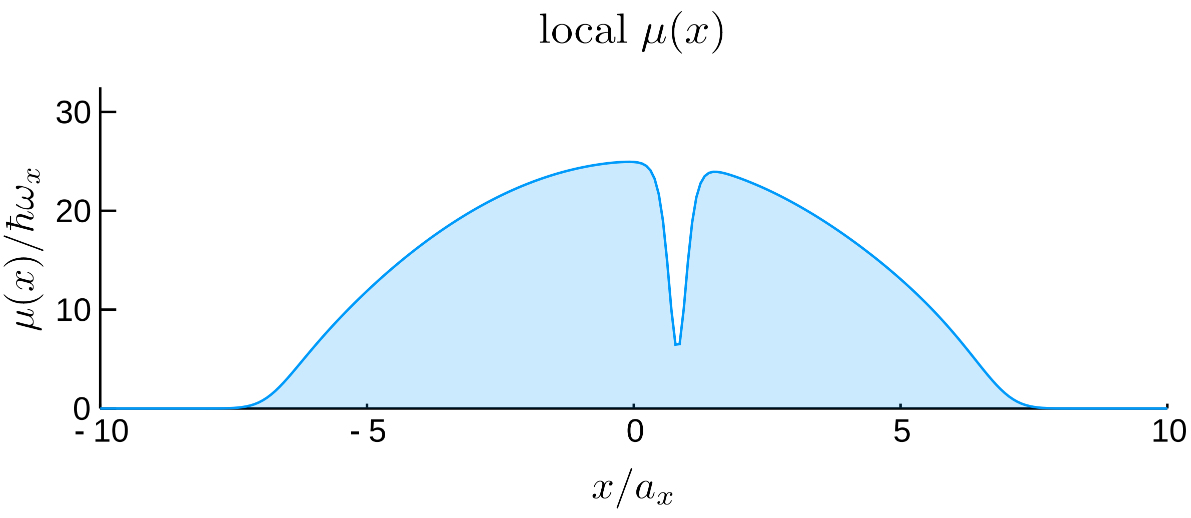
```
showpsi(x,ψf)

anim = @animate for i in 1:length(t)-4
    ψ = xspace(sols[i],simSoliton)
    y = g*abs2.(ψ)
    p = plot(x,y,fill=(0,0.2),size=(500,200))
    xlims!(-10,10); ylims!(0,1.3*μ)
    title!(L"\textrm{local}\; \mu(x)")
    xlabel!(L"x/a_x"); ylabel!(L"\mu(x)/\hbar\omega_x")
end
animpath = joinpath(@__DIR__,"media/soliton.gif")
gif(anim,animpath,fps=30)
```

The result is visible in the media folder.

Here we simply plot the final state:

```
ψ = xspace(sols[end],simSoliton)
y = g*abs2.(ψ)
p=plot(x,y,fill=(0,0.2),size=(500,200))
xlims!(-10,10); ylims!(0,1.3*μ)
title!(L"\textrm{local}\; \mu(x)")
xlabel!(L"x/a_x"); ylabel!(L"\mu(x)/\hbar\omega_x")
plot(p)
```


local $\mu(x)$
$\mu(x)/\hbar\omega_x$
x/a_x