Solve the GPE in a 1D parabolic trap

Ashton Bradley

November 6, 2019

1 Introduction

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

The mean field order parameter of a Bose-Einstein condensate far below the critical temperatrure for condensation evolves according to the GP-equation

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

with potential $V(x,t) = m\omega_x^2x^2/2$, and positive interaction strength g.

We work in harmonic oscillator units, taking length in units of $a_x = \sqrt{\hbar/m\omega_x}$ and time in units of $1/\omega_x$.

The equation of motion that we solve numerically is

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

where all quantities are now dimensionless.

2 Loading the package

First, we load some useful packages, and set up defaults for `Plots`.

```
using Plots, LaTeXStrings
gr(fmt="png",legend=false,titlefontsize=12,size=(500,200),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)
```

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

```
L = (40.0,)
N = (512,)
sim = Sim(L,N)
@unpack_Sim sim;
μ = 25.0
```

```
25.0
```

Here we keep most of the default parameters but increase the chemical potential.

2.1 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 the definition as a scalar function because it will be evaluated on the grid using a broadcasted dot-call.

3 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 on the grid and all modified variables are scooped up into `sim`:

```
ψi = ψ0.(x,μ,g)
ϕi = kspace(ψi,sim) #sim uses Fourier transforms that are norm-preserving
@pack_Sim! sim;
sim
```

```
FourierGPE.Sim{1}
  L: Tuple{Float64}
  N: Tuple{Int64}
  μ: Float64 25.0
  g: Float64 0.1
  γ: Float64 0.5
  ti: Float64 0.0
  tf: Float64 4.0
  Nt: Int64 200
  params: FourierGPE.Params
  V0: Array{Float64}((512,)) [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
0.0  ...  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
  t: LinRange{Float64}
  ϕi: Array{Complex{Float64}}((512,)) Complex{Float64}[70.07086743680145 +
0.0im, -59.798517655187986 - 0.7338750582771199im, 34.8556584873079 + 0.855
6583885626757im, -8.70747533342298 - 0.32071530748878 81im, -6.7825212654428
5 - 0.3332039032656252im, 8.506444735601493 + 0.5226049379922504im, -2.2145
262509741106 - 0.16335326999464506im, -3.530241087221503 - 0.30400618591783
03im, 4.029537717252456 + 0.39687482465281543im, -0.6106697231179278 - 0.06
772199696863206im  ...  -2.407115406137964 + 0.29688936837062363im, -0.610669
723117928 + 0.06772199696863168im, 4.029537717252456 - 0.39687482465281543i
m, -3.530241087221504 + 0.3040061859178303im, -2.2145262509741115 + 0.16335
326999464467im, 8.506444735601493 - 0.5226049379922505im, -6.78252126544285
 + 0.33320390326562527im, -8.70747533342298 + 0.32071530748878807im, 34.855
6584873079 - 0.8556583885626752im, -59.79851765518799 + 0.7338750582771209i
m]
  alg: OrdinaryDiffEq.Tsit5 OrdinaryDiffEq.Tsit5()
  reltol: Float64 1.0e-6
  flags: UInt32 0x00000000
  nfiles: Bool false
  path: String "/Users/abradley/.julia/packages/FourierGPE/YkBQV/src"
  filename: String "save"
  X: Tuple{Array{Float64,1}}
  K: Tuple{Array{Float64,1}}
  espec: Array{Complex{Float64}}((512,)) Complex{Float64}[0.0 + 0.0im, 0.01
2337005501361697 + 0.0im, 0.04934802200544679 + 0.0im, 0.11103304951225527
+ 0.0im, 0.19739208802178715 + 0.0im, 0.30842513753404244 + 0.0im, 0.444132
1980490211 + 0.0im, 0.6045132695667231 + 0.0im, 0.7895683520871486 + 0.0im,
 0.9992974456102974 + 0.0im  ...  1.2337005501361697 + 0.0im, 0.9992974456102
974 + 0.0im, 0.7895683520871486 + 0.0im, 0.6045132695667231 + 0.0im, 0.4441
321980490211 + 0.0im, 0.30842513753404244 + 0.0im, 0.19739208802178715 + 0.
0im, 0.11103304951225527 + 0.0im, 0.04934802200544679 + 0.0im, 0.0123370055
01361697 + 0.0im]
  T: FourierGPE.Transforms{1,1}
```

The important points to note here are that we have modified μ and the initial condition ϕ_i, and we have left the default damping parameter $\gamma = 0.5$ which means we are going to find a ground state of the GPE.

3.1 Default simulation parameters

The source code defining the simulation type `Sim` sets the default values and also has some further explanation of each variable:

```
@with_kw mutable struct Sim{D} <: Simulation{D} @deftype Float64
    # Add more parameters as necessary, or add to params (see examples)
    L::NTuple{D,Float64} # box length scales
    N::NTuple{D,Int64}  # grid points in each dimensions
    μ = 15.0    # chemical potential
    g = 0.1     # interaction parameter
    γ = 0.5; @assert γ >= 0.0 # damping parameter
    ti = 0.0    # initial time
    tf = 2/γ    # final time
    Nt::Int64 = 200     # number of saves over (ti,tf)
    params::UserParams = Params() # optional user parameters
    V0::Array{Float64,D} = zeros(N)
    t::LinRange{Float64} = LinRange(ti,tf,Nt) # time of saves
    ϕi::Array{Complex{Float64},D} = zeros(N) |> complex # initial condition
```

```
    alg::OrdinaryDiffEq.OrdinaryDiffEqAdaptiveAlgorithm = Tsit5() # default solver
    reltol::Float64 = 1e-6 # default tolerance; may need to use 1e-7 for corner cases
    flags::UInt32 = FFTW.MEASURE # choose a plan. PATIENT, NO_TIMELIMIT, EXHAUSTIVE
    # === saving
    nfiles::Bool = false
    path::String = nfiles ? joinpath(@__DIR__,"data") : @__DIR__
    filename::String = "save"
    # === arrays, transforms, spectral operators
    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(K)
    T::TransformLibrary = makeT(X,K,flags=flags)
end
```

4 Evolution in k-space

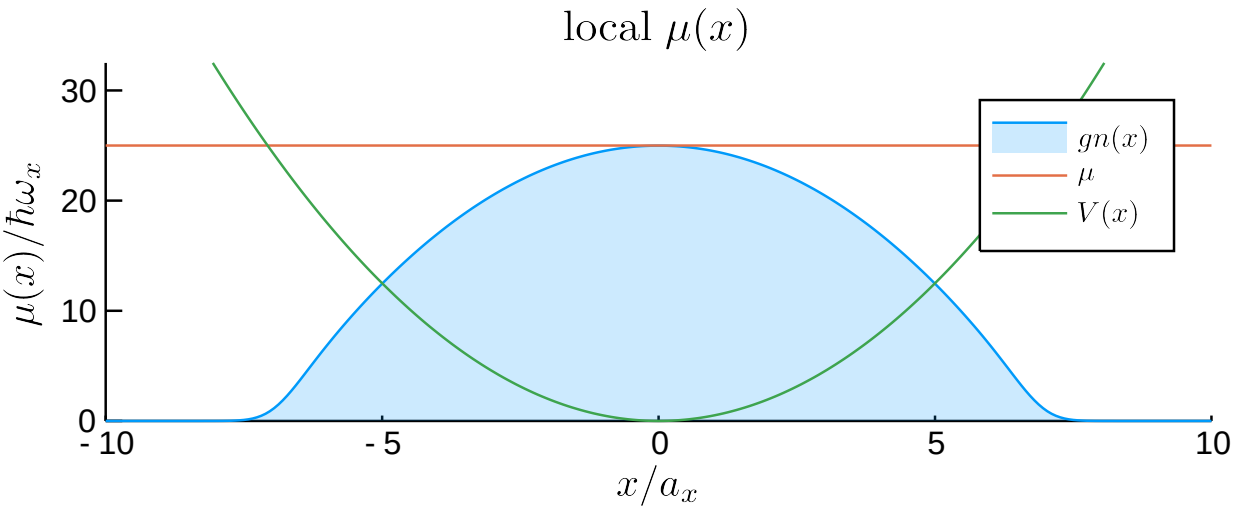
The `FFTW` library is used to evolve the Gross-Pitaevskii equation in k-space

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

By default the solver returns all time slices specified by the `t` vector (`t=LinRange(ti,tf,Nt)`) and solution information in a single variable `sol`.

Let's have a look at the final state and verify we have a ground state with the correct chemical potential:

```
ϕg = sol[end]
ψg = xspace(ϕg,sim)
p=plot(x,g*abs2.(ψg),fill=(0,0.2),size=(500,200),label=L"gn(x)")
plot!(x,one.(x)*μ,label=L"\mu")
plot!(x,V.(x,0.0),label=L"V(x)",legend=:topright)
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 $1/\gamma$.

5 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 state that varies slowly over the scale of the soliton (the healing length ξ).

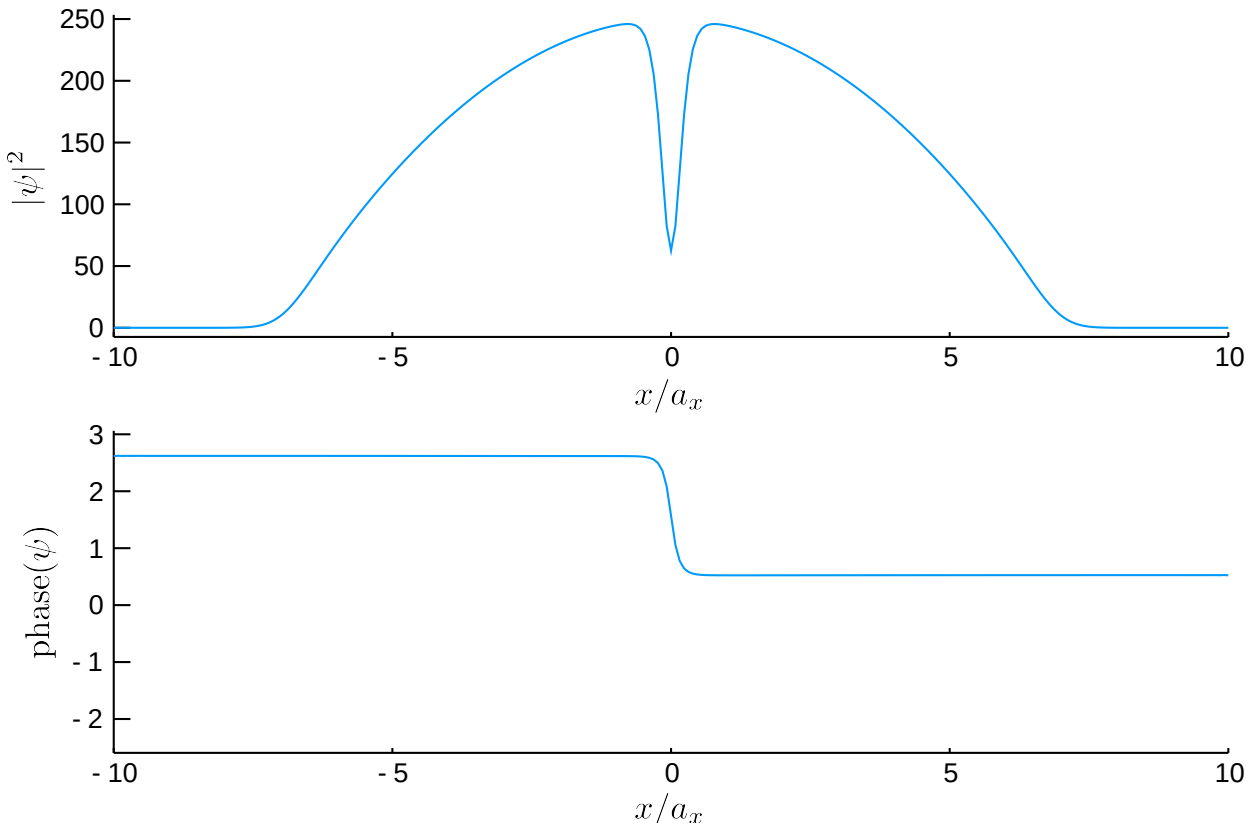
5.1 Imprinting a dark soliton

```
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, and is intialized at `xs=0.0`.

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



5.2 Initilize Simulation

We can use the previous parameters in `sim` to define a new simulation, while modifying parameters as required (in this case the damping and simulation timescale):

```
γ = 0.0
tf = 8*pi/sqrt(2); t = LinRange(ti,tf,Nt)
dt = 0.01π/μ
ϕi = kspace(ψs,sim)
simSoliton = Sim(sim;γ=γ,tf=tf,t=t,ϕi=ϕi) #define a new simulation, using keywords
# @pack_Sim! simSoliton; #we could instead pack everything into simSoliton, since we
have made all changes
```

5.3 Solve equation of motion

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

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

```
2.692537 seconds (15.88 M allocations: 1.232 GiB, 9.56% gc time)
```

5.4 View the solution using Plots

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

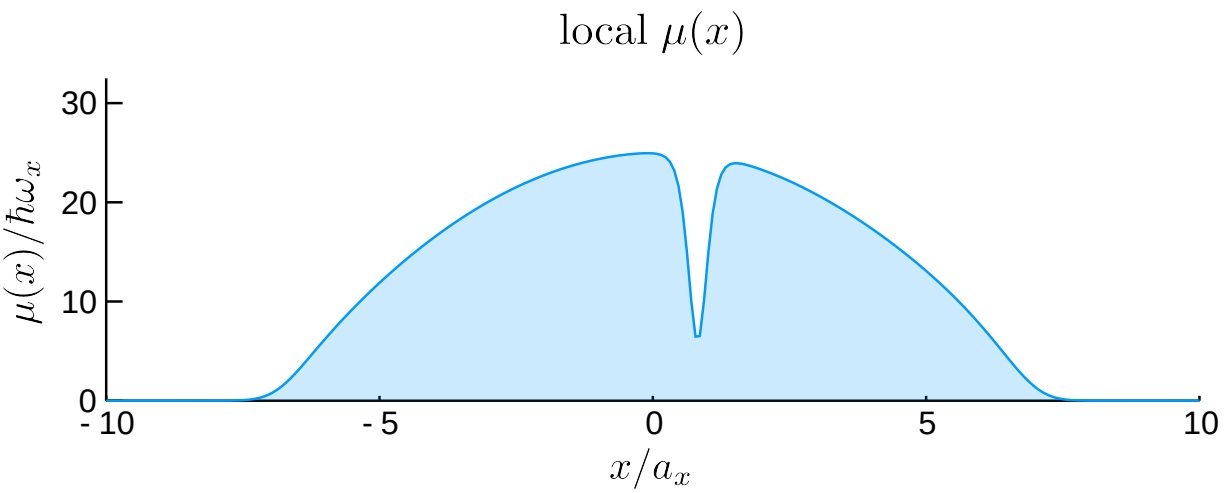
```
ϕf = sols[end-4]
ψf = xspace(ϕf,simSoliton)
showpsi(x,ψf)

anim = @animate for i in 1:length(t)-4 #make it periodic by ending early
    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 of this repository.

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)
```



The dark soliton executes simple harmonic motion with amplitude detemined by its depth.