[1]:
```
using Markdown
```

[2]:
```
using LaTeXStrings, DifferentialEquations, LinearAlgebra,␣
↪SparseArrays
```

[3]:
```
begin
        using Plots
        gr()
end
```

[3]: Plots.GRBackend()

1 2D Multiple Attractor System

Setup vector field equations. Can't split into jump process.

[4]:
```
begin
          = 0.1
          = 3
        n = 4
        dfx(x, y) = 9x + 9y - (1 + 2x^3 + 2y^3)
        dfy(x, y) = 1 + 2x^3 + 11y - (11x + 2y^3)
end
```

[4]: dfy (generic function with 1 method)

[5]:
```
begin
        n_min = -3
        n_max = 3
end
```

[5]: 3

Compute the vector field

[6]:
```
begin
        dq = (n_max - n_min) / 10.
        xs = n_min-1:dq:n_max+1
        ys = n_min-1:dq:n_max+1
```

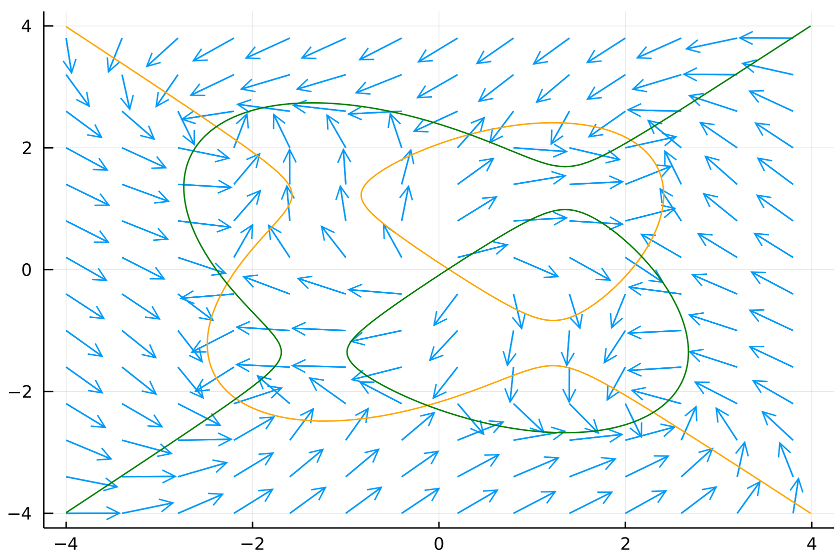
```
        dxs = n_min-1:.05:n_max+1
        dys = n_min-1:.05:n_max+1

        df(x, y) = normalize([dfx(x, y), dfy(x,y)]) ./ 1.75

        xxs = [x for x in xs for y in ys]
        yys = [y for x in xs for y in ys]

        quiver(xxs, yys, quiver=df)
        contour!(dxs, dys, dfx, levels=[0], color=:orange,␣
 ↪label="X Nullcline", colorbar = false)
        p1 = contour!(dxs, dys, dfy, levels=[0], color=:
 ↪green, label = "Y Nullcline", colorbar = false)
        p1
end
```

[6]:



Setup the Stochastic Differential Equation model with diagonal noise

[7]:

```
function f(du, u, p, t)
        du[1] = dfx(u...)
```

```julia
        du[2] = dfy(u...)
        du[1], du[2]
end
```

[7]: f (generic function with 1 method)

[8]:
```julia
begin
        g(du, u, p, t) = begin
          du[1] = 1.25
          du[2] = 1.25
        end
        sde_prob= SDEProblem(f, g, [0., 0.], (0.0, 1000.0))
        sde_sol = solve(sde_prob, SRIW1())
end
```

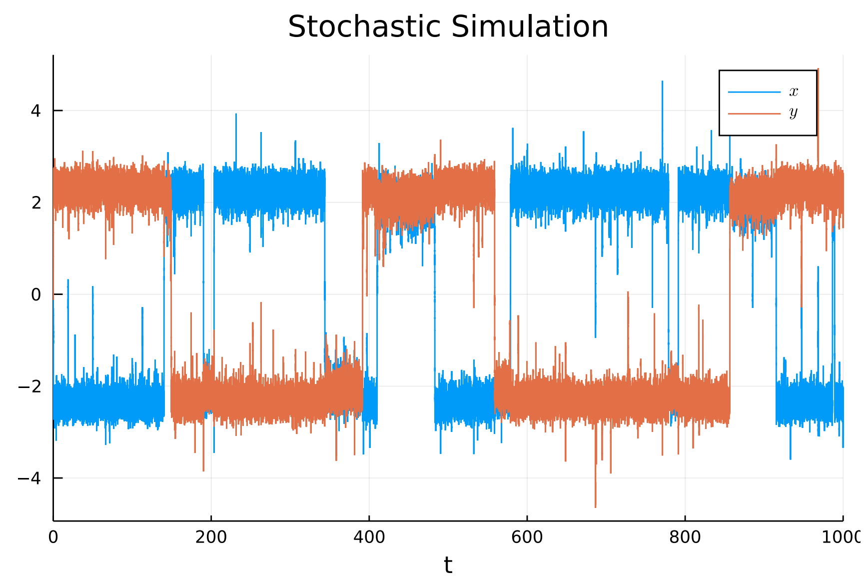
[8]:
```
retcode: Success
Interpolation: 1st order linear
t: 102399-element Vector{Float64}:
    0.0
    3.6514837167011073e-6
    7.759402897989854e-6
    1.2380811976939694e-5
    1.7579897190758263e-5
    2.3428868056304155e-5
    3.0008960280043283e-5
    3.7411564031749804e-5
    4.573949325241964e-5
    5.5108413625673204e-5
    6.564844904558346e-5
    7.750598889298251e-5
    9.084572122130643e-5
    ⋮
  999.9273990038436
  999.9327073681966
  999.9386792780938
  999.9453976767281
  999.9520086655147
  999.9594460278995
  999.9678130605824
```

```
  999.9753767113151
  999.9838858183894
  999.9918832268087
  999.9991628329426
 1000.0
u: 102399-element Vector{Vector{Float64}}:
 [0.0, 0.0]
 [-0.0003741681687235294, 0.0024860957421516724]
 [-9.830430075056926e-5, 0.0015009224897204894]
 [9.876643964970858e-5, 0.002018191652676208]
 [0.0025328542898177923, 0.006282886463371841]
 [0.004738240631971388, 0.005149982367181946]
 [0.0038682082140360854, 0.0012974430835765412]
 [0.003942582826542919, 0.0035411097035164566]
 [0.000754977532390429, 0.009616452490668628]
 [0.000543791598932073, 0.010589094282551944]
 [-0.00028879046305781305, 0.010007252881840378]
 [-0.00039664324483776206, 0.012183410472666134]
 [-0.007983593211170082, 0.0028498682300137777]
 ⋮
 [-2.325456398147698, 2.0404344901687943]
 [-2.3628674754624597, 2.2082195771081854]
 [-2.227806767926245, 2.274110981115315]
 [-2.138879059064676, 2.371801695012793]
 [-2.3019843996946925, 2.37157087112837]
 [-2.1222788519696083, 2.365766146632038]
 [-2.1902890837485196, 2.414110571405585]
 [-2.2842399282317642, 2.4797931393887933]
 [-2.122701267769504, 2.5261678936603373]
 [-2.1266255847053848, 2.4676722932543136]
 [-2.215309977168978, 2.544238824437627]
 [-2.2248472770642085, 2.547545760361412]
```

[9]:

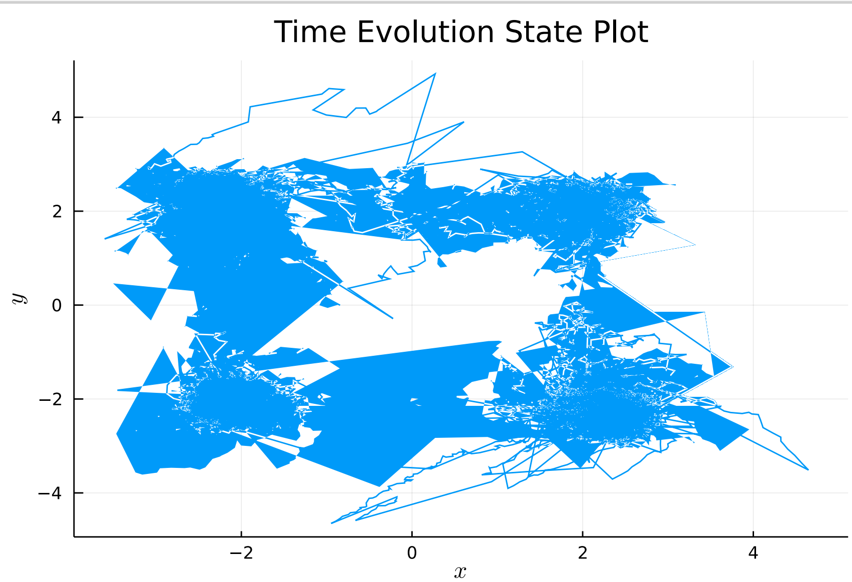
```
pstoch_1 = plot(sde_sol,title="Stochastic Simulation",␣
↪labels=[L"x" L"y"])
```

[9]:



[10]:
```
pstoch_2 = plot(sde_sol[1000:end], vars = [(1,2)], label="",␣
↪xlabel=L"x", ylabel = L"y", title="Time Evolution State␣
↪Plot")
```

[10]:



[11]:
```
# pstoch = plot(pstoch_1, pstoch_2, layout=(1, 2),␣
↪size=(900, 350))
```

[12]:
```
# savefig(pstoch, "ch3_four_attractor_stoch.pdf")
```

Helper function to tip towards different quadrants for each run of the simulation

[13]:
```
function get_u0(i)
        offset = 1.0
        new_u0 = [0.0, 0.0]
        if i % 4 == 0
                new_u0 .+= [offset, offset]
        elseif i % 4 == 1
                new_u0 .+= [-1* offset, offset]
        elseif i % 4 == 2
                new_u0 .+= [-1* offset, -1* offset]
        else
                new_u0 .+= [offset,-1* offset]
        end
        new_u0
end
```

[13]:
```
get_u0 (generic function with 1 method)
```

Add smaller noise term for longer time simulation.

[14]:
```
begin
        g2(du, u, p, t) = begin
          du[1] = 0.5
          du[2] = 0.5
        end
end
```

[14]:
```
g2 (generic function with 1 method)
```

Helper to compute the 2D histograms for unnomalized probability

[15]:
```
begin
        nbins = 257
        bin_min = -10.
        bin_max = 10.
        xbins = range(bin_min, bin_max, length=nbins)
        ybins = range(bin_min, bin_max, length=nbins)
        xbins

        function bucket_idx(val, vec)
                idx = 1
                while val > vec[idx]
                        idx += 1
                end
                idx - 1
        end

        function althist(x, y, dt, xedges, yedges)
           counts = spzeros(length(xedges)-1,␣
↪length(yedges)-1)

                for i=1:length(x) - 1
                        r = bucket_idx(x[i], yedges)
                        c = bucket_idx(y[i], xedges)
                        counts[r,c] += dt[i]
                end

                 counts
        end

        drop_num = 2000
end
```

[15]: 2000

[16]:
```
unnorm = spzeros(length(xbins)-1, length(ybins)-1)
```

[16]:
```
256×256 SparseMatrixCSC{Float64, Int64} with 0 stored␣
↪entries:
```

Rerun the following cell to add more data points to the simulation.

[17]:
```
begin
	for i in 1:4*100
		u00 = get_u0(i)
		sde_prob_long = SDEProblem(f, g2, u00, (0.0,␣
↪1e3))
		sde_sol_long= solve(sde_prob_long, SRIW1())
		unnorm .+= althist(sde_sol_long[1, drop_num:
↪end], sde_sol_long[2, drop_num:end], diff(sde_sol_long.
↪t[drop_num:end]), xbins, ybins)
	end
	unnorm
end
```

[17]:
```
256×256 SparseMatrixCSC{Float64, Int64} with 4391 stored␣
↪entries:
```

Normalized and visualize the probability and resulting Quasi-potential.

[18]:
```
prob = unnorm ./ sum(unnorm)
```

[18]:
```
256×256 SparseMatrixCSC{Float64, Int64} with 4391 stored␣
↪entries:
```

[19]:
```julia
begin
	plotly()
	p = surface(- log.(prob))
	gr()
	p
end
```

```
 Info: For saving to png with the Plotly backend PlotlyBase␣
↪has to be
installed.
 @ Plots /Users/bsm/.julia/packages/Plots/lzHOt/src/backends.
↪jl:372
```

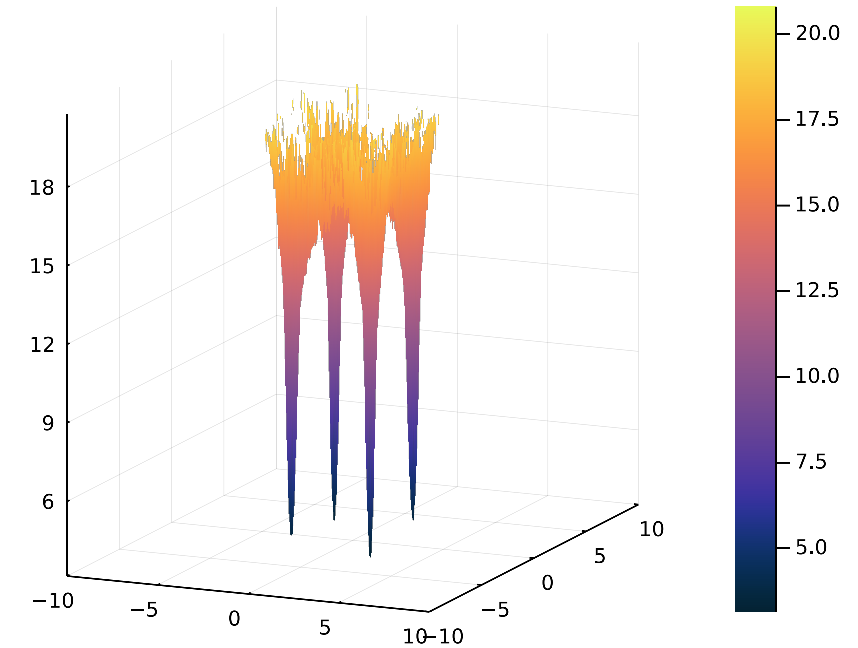
[20]:
```julia
begin
	plotly()
	pp = surface(xbins, ybins,prob,c = :grayC)
	gr()
	pp
end
```

[21]:
```julia
begin
	xplot = range(bin_min, bin_max, length=nbins-1)
	yplot = range(bin_min, bin_max, length=nbins-1)
end
```

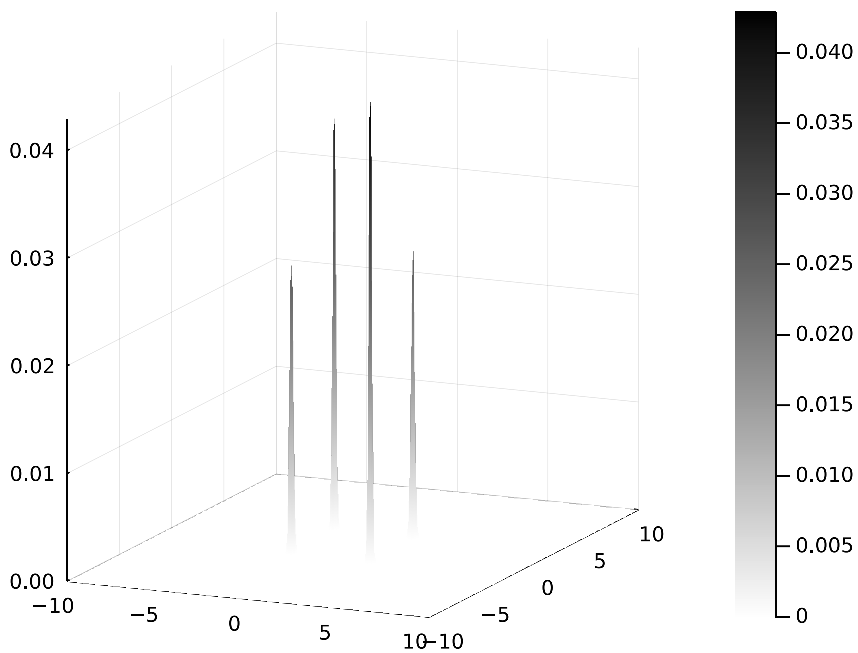
[21]: -10.0:0.0784313725490196:10.0

[22]:
```julia
begin
	pp1 = surface(xplot, yplot,-log.(prob), c = :thermal)
	pp2 =  surface(xplot, yplot, prob, c = :grayC)
	ps = plot(pp1, pp2, layout =(2, 1), size=(500, 800))
end
```

[22]:



18
15
12
9
6
−10
−5
0
5
10
−10
−5
0
5
10
20.0
17.5
15.0
12.5
10.0
7.5
5.0



0.04
0.03
0.02
0.01
0.00
−10
−5
0
5
10
−10
−5
0
5
10
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0

[23]:
```
# savefig(ps, "ch3_4a_ad_hoc.pdf")
```

Helper function to generate - ² to solve the Helmhotz equation

[24]:
```
begin
        import DiffEqOperators
        const DiffEqOp = DiffEqOperators

        eye(N, M) = sparse(I, N, M)
        eye(M) = eye(M, M)
        diff1(M) = [ [1.0 zeros(1, M - 1)]; diagm(1 =>␣
 ↪ones(M - 1)) - eye(M) ]
        sdiff1(M) = sparse(diff1(M))
        flatshape(X) = reshape(X, length(X))


        # make the discrete -Laplacian in 2d, with Dirichlet␣
 ↪boundaries
        function laplacian2d(Nx, Ny, Lx, Ly)
                dx = Lx / (Nx + 1)
                dy = Ly / (Ny + 1)
                Dx = sdiff1(Nx) / dx
                Dy = sdiff1(Ny) / dy
                Ax = Dx' * Dx
                Ay = Dy' * Dy
                return Dx, Dy, kron(eye(Ny), Ax) + kron(Ay,␣
 ↪eye(Nx))
        end
end
```

[24]:
```
laplacian2d (generic function with 1 method)
```

[25]:
```
# fg =  plot(p1, p2, p3, layout=(3,1), size=(750, 2250))
```

[26]:
```
# savefig(fg, "ch3_four_attractor.pdf")
```

1.0.1 Appendix: Helper decomposition functions

[27]:
```
## We are trying to solve the following equation
##  ²U  (x) = -   F(x)
## We are starting in 2 dimensions
function helmholtz(prob::DiscreteProblem, minval::Int64, maxval::Int64, nsample::Int64)
    dx = (maxval - minval) / (nsample + 1)
    dy = (maxval - minval) / (nsample + 1)
    xs = range(minval, maxval, length=nsample)
    ys = range(minval, maxval, length=nsample)
     x(x, y) = prob.f.f([0.0, 0.0], [x, y], prob.p, nothing)[1]
     y(x, y) = prob.f.f([0.0, 0.0], [x, y], prob.p, nothing)[2]
    # Let's first construct F = [f1, f2]
    f1 = [ x(x, y) for x in xs, y in ys]
    f2 = [ y(x, y) for x in xs, y in ys]
    _, _, Δ = laplacian2d(nsample, nsample, maxval - minval, maxval - minval)

    Dx = DiffEqOp.CenteredDifference{1}(1, 2, dx, nsample)
    Dy = DiffEqOp.CenteredDifference{2}(1, 2, dy, nsample)

    bcx = DiffEqOp.Dirichlet0BC{1}(eltype(f1), size(f1))
    bcy = DiffEqOp.Dirichlet0BC{2}(eltype(f2), size(f2))

    rhs =  Dx * bcx * f1 + Dy * bcy *  f2

    u = Δ \ flatshape(rhs)
    xs, ys, reshape(u, size(rhs)...)
end
```

[27]: helmholtz (generic function with 1 method)

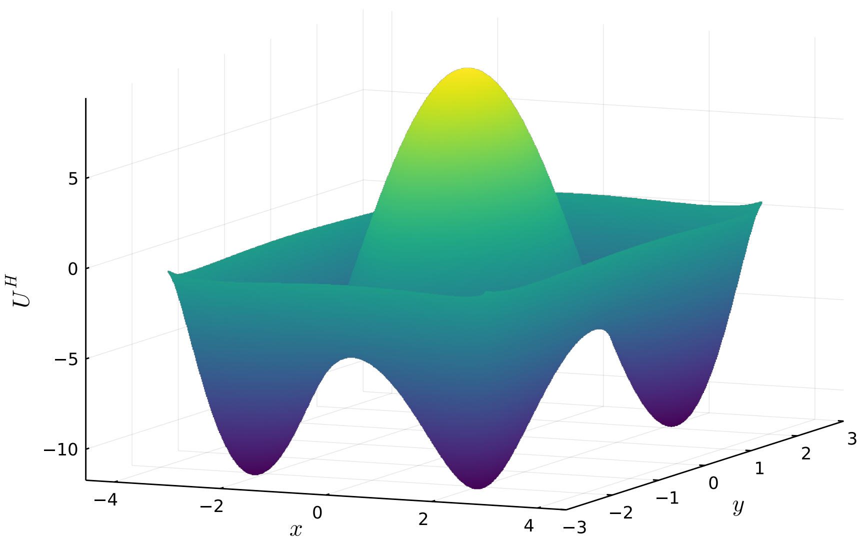
[28]:
```
begin
        nsample = 512
        dprob = DiscreteProblem(f, [0.0,0.0], (0, 100.), nothing)
```

```
    x, y, u_h = helmholtz(dprob, -3, 3, nsample)
end
```

[28]:
```
(-3.0:0.011741682974559686:3.0, -3.0:0.011741682974559686:3.
↪0,
[0.1294998777831836 0.16243703338537016 … 0.
↪016737557349006546
0.08870109675618942; 0.04578652079205767 0.
↪055757946469432204 …
0.015232510279881449 0.11016221774968435; … ; 0.
↪12185773514106535
0.026927638629895403 … 0.055757946469432114 0.
↪04116306520425123;
0.1003968086682542 0.02843307474038758 … 0.1670604889731765
0.12949987778318356])
```

[29]:
```
begin
    gr()
    fg = surface(x, y, u_h, colorbar=:none,  c=:viridis,␣
↪aspect_ratio=:equal, xlabel=L"x", ylabel=L"y",␣
↪zlabel=L"U^H")
end
```

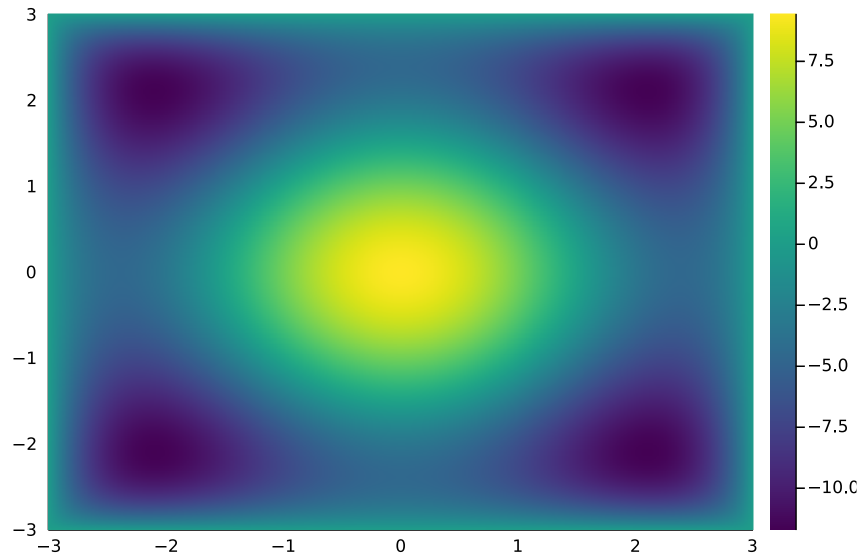
[29]:



[30]:

```
p3 = heatmap(x, y, u_h, c=:viridis)
```

[30]:



[31]:

```
function make_inplace(f::Function)
    function inplace_f(out, x)
        out .= f(x)
    end
    return inplace_f
end
```

[31]: make_inplace (generic function with 1 method)

[32]:

```
function normal(prob::DiscreteProblem, minval::Int64, maxval:
↪:Int64, nsample::Int64, u0::Matrix{Float64})
    dx = (maxval - minval) / (nsample + 1)
    dy = (maxval - minval) / (nsample + 1)

    xs = range(minval, maxval, length=nsample)
```

```
    ys = range(minval, maxval, length=nsample)
     x(x, y) = prob.f.f([0.0, 0.0], [x, y], prob.p,␣
↪nothing)[1]
     y(x, y) = prob.f.f([0.0, 0.0], [x, y], prob.p,␣
↪nothing)[2]

    f1 = [ x(x, y) for x in xs, y in ys]
    f2 = [ y(x, y) for x in xs, y in ys]

    Dx = DiffEqOp.UpwindDifference{1}(1, 2, dx, nsample, 1)
    Dy = DiffEqOp.UpwindDifference{2}(1, 2, dy, nsample, 1)

    bcx = DiffEqOp.Dirichlet0BC{1}(eltype(u0), size(u0))
    bcy = DiffEqOp.Dirichlet0BC{2}(eltype(u0), size(u0))

    function find_my_zero(u)
        dudx = Dx * bcx * u
        dudy = Dy * bcy * u
        return dot(dudx, f1 .+ dudx) + dot(dudy, f2 .+ dudy)
    end

    fmz! = make_inplace(find_my_zero)

    sol = nlsolve(fmz!, u0, show_trace=true, iterations=3000)
    sol.zero
end
```

[32]: normal (generic function with 1 method)