Photo: Wikipedia / F. Veronesi

Scaly-breasted Munia

斑文鳥

ASTR 660 / Class 13

Smoothed particle hydrodynamics

Smoothed particle hydrodynamics

The idea of SPH is to discretise the fluid into elements of fixed mass, which keep their identity as they move around (like N-body particles).

This is a “Lagrangian” rather than “Eulerian” representation of the fluid. The PDEs in the Eulerian formalism are converted to ODEs.

Unlike N-body particles, SPH particles experience short-range (e.g. pressure) forces and transformation of KE into internal energy (and vice versa). The ODE system is more complicated.

Smoothed particle hydrodynamics

SPH “particles” are not point-like, hard-edged atoms. They are “tracer points” that track the mass centers of fluid elements.

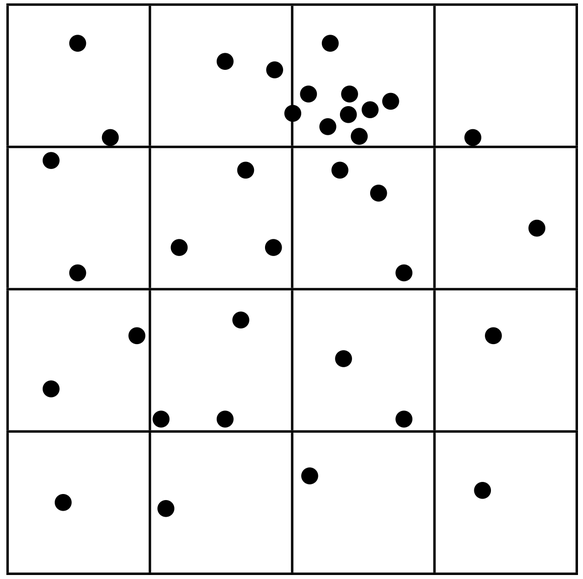
The distribution of the mass “between particles” (i.e. the smoothing of the particles to get a continuous field) is more important than the “numerical” trick convenience.

To find equations of motion for SPH particles, we need to assign a “density”. We have to be explicit about the “volume” each particle is supposed to represent.

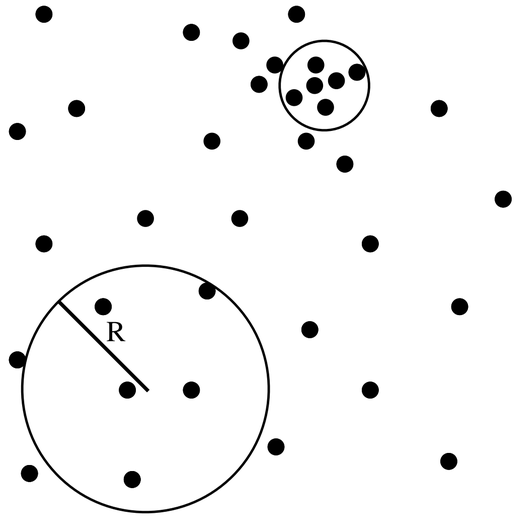
Since the mass of the particles is fixed, the volume must change from particle to particle and time to time: smaller in regions of higher density, larger in regions of lower density.

Smoothed particle hydrodynamics

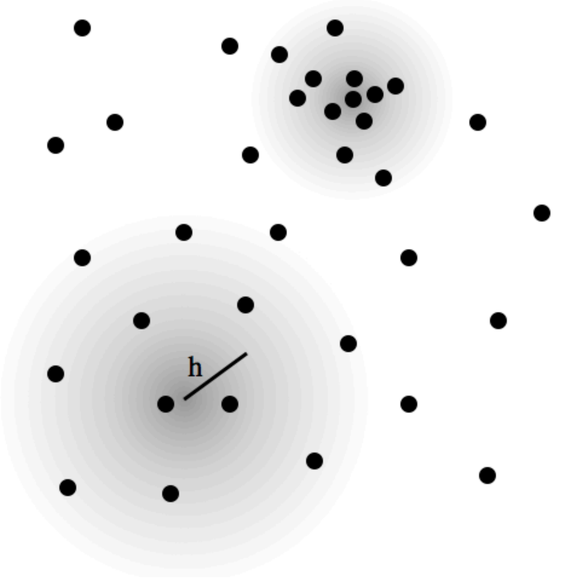
Figure: Daniel Price



Eulerian: density calculated on a mesh. Not adaptive, forces calculated on grid rather than at particle positions.



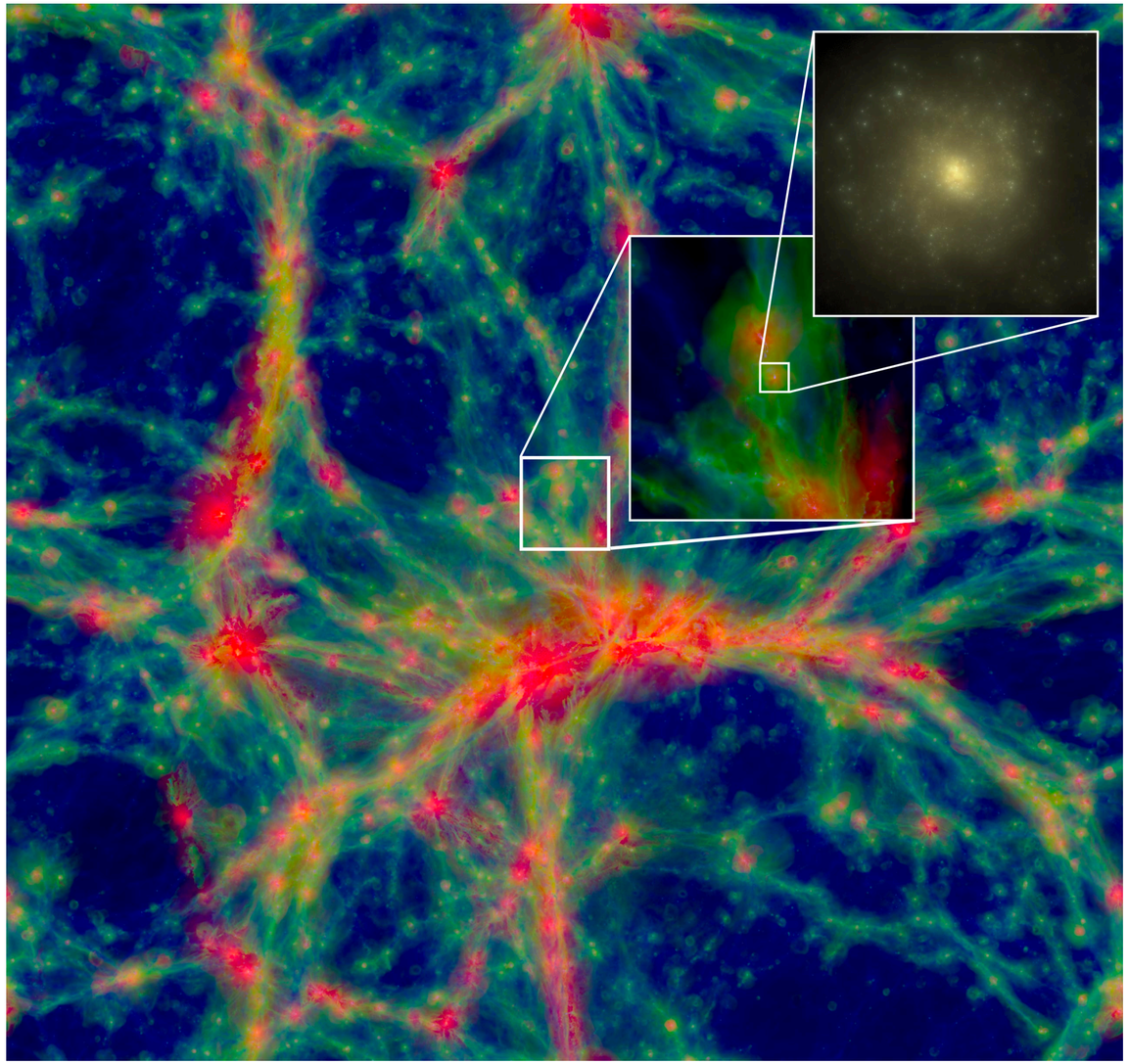
Lagrangian but not smoothed: density calculated in spheres with volume enclosing a fixed number of neighbours. Adaptive, but noisy.



SPH: density calculated as a **weighted** sum over neighbours. Naturally adaptive and less noisy.

Smoothed particle hydrodynamics

Figure: Schaye et al. / EAGLE



SPH is well-suited to astrophysics (especially galaxy formation and ISM).

- No grid: simpler code, less book-keeping.
- Naturally adaptive over a large dynamic range (typical case in gravity problems) with no extra work (much easier compared to mesh refinement schemes).
- Principles and code are very close to those for particle-based representation of collisionless matter (stars and DM).

Smoothing kernel

The weighting is done by **convolving** the particle distribution by a locally adaptive **kernel**:

$$\rho(\vec{r}) = \sum_{j=1}^{N_{ngb}} m_j W(\vec{r} - \vec{r}_j, h)$$

Some of the "art" of SPH is therefore in choosing an appropriate kernel function (although there is really only one standard choice).

Requirements: it should **integrate to unity** (conserve mass), **decrease monotonically** (more weight closer to the particle position), **not have a sharp peak** in the center (otherwise the density estimate will be noisy), have **smooth derivatives** (will see why shortly) and be **symmetric** (otherwise angular momentum won't be conserved).

A uniform (spherical top-hat) function doesn't satisfy the requirement of having more weight in the center.

Gaussian kernel

At face value, a Gaussian kernel seems like it would be OK. You can never go far wrong with a Gaussian…

$$W(\vec{r} - \vec{r}', h) = \frac{\sigma}{h^d} \exp\left[-\frac{(\vec{r} - \vec{r}')^2}{h^2}\right]$$

$d = 1,2,3$ for 1-d, 2-d and 3-d.

The normalization also depends on the number of dimensions (W must integrate to 1). In 1-d, $\sigma = 1/\sqrt{\pi}$; in 2-d, $\sigma = 1/\pi$; in 3-d, $\sigma = 1/(\pi\sqrt{\pi})$.

This kernel is positive all the way to $|\vec{r} - \vec{r}'| \to \infty$. The density estimate for every particle depends on the position of every other particle, even if just a little.

Cubic spline kernel

A more practical kernel would go to zero exactly, at some finite distance. The jargon for this is **compact support**. At small distances, the kernel should be more-or-less Gaussian.

The standard choice is the **cubic spline kernel** (Monaghan & Lattanzio 1985). Writing $q = |\vec{r} - \vec{r}'|/h$:

$$w(q) = \sigma \begin{cases} \frac{1}{4}(2-q)^3 - (1-q)^3, & 0 \le q < 1 \\ \frac{1}{4}(2-q)^3 & 1 \le q < 2 \\ 0 & q \ge 2 \end{cases}$$

The full kernel is $W(q) = \frac{1}{h^d} w(q)$, and $\sigma_{1d} = 2/3$, $\sigma_{2d} = 10/(7\pi)$, $\sigma_{3d} = 1/\pi$.

Cubic spline kernel

The origin of this kernel is quite complicated (see Price). It satisfies the requirements of being continuous in the first two derivatives, having compact support (goes to zero at $2h$) and otherwise looking somewhat Gaussian.

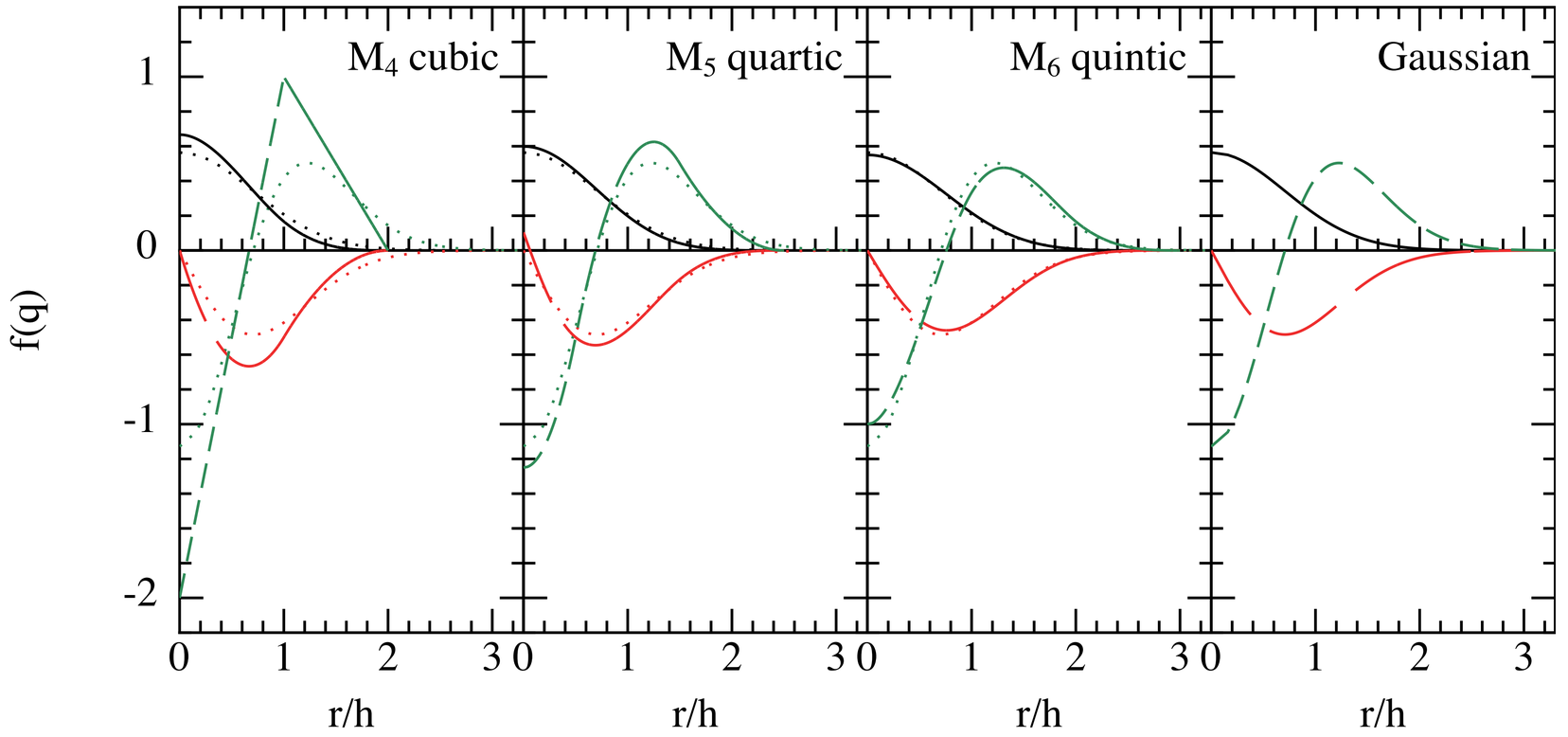


Figure: Daniel Price

Gadget

$$w(q) = \sigma \begin{cases} \frac{1}{4}(2-q)^3 - (1-q)^3, & 0 \le q < 1 \\ \frac{1}{4}(2-q)^3 & 1 \le q < 2 \\ 0 & q \ge 2 \end{cases}$$

The cubic spline kernel is used in Gadget, but written with the notation change $2h \to h$:

$$W(r,h) = \frac{8}{\pi h^3} \begin{cases} 1 - 6\left(\frac{r}{h}\right)^2 + 6\left(\frac{r}{h}\right)^3, & 0 \le \frac{r}{h} \le \frac{1}{2}, \\ 2\left(1 - \frac{r}{h}\right)^3, & \frac{1}{2} < \frac{r}{h} \le 1, \\ 0, & \frac{r}{h} > 1. \end{cases} \quad (4)$$

Notice Gadget uses the same kernel to soften the gravitational force. However, in that case, $h = 2.8\epsilon$ is **fixed** to a number we decide at the start of the simulation (apart from an unimportant issue to do with comoving coordinates; see also discussion in Power et al.).

For SPH particles, the h is **adaptive**, i.e. different for every particle.

Neighbours and the smoothing scale

The density of a given particle is $\rho_i = \sum_{j=1}^{N_{ngb}} m_j W(\vec{r}_i - \vec{r}_j, h_i)$, i.e. a weighted sum over neighbours.

There are different schemes for setting h_i. In Gadget, it is chosen such that the **mass** within the kernel volume is constant for any density, i.e.

$$\frac{4\pi}{3} h_i^3 \rho_i = \bar{N}_{\mathrm{ngb}} \bar{m}$$

The right-hand side is product of the *typical* number of neighbours and the *average* particle mass times the typical particle mass.

An alternative is to force the *number* of neighbours to be constant. If all the particles have equal mass this is the same thing (in galaxy simulations, star formation acts as a ‘sink’ of mass).

Neighbours and the smoothing scale

$$\frac{4\pi}{3} h_i^3 \rho_i = \bar{N}_{\rm ngb} \bar{m}$$

This equation is is **implicit**, i.e., roughly speaking, we have $h_a \propto \left(m_a/\rho_a\right)^{1/3}$, but ρ_a itself depends on h_a. We need to solve this simultaneous equation with a (quick and reliable) **root-finding** method.

In Gadget (for example), one has to choose a "number of neighbours" parameter. This sets the *typical* number of neighbours used to find the smoothing length for each particle. There is an associated tolerance parameter (e.g. $N_{\rm ngb} = 64 \pm 2$).

Increasing the number of neighbours smooths the fluid more, but does not directly control the convergence/accuracy (see discussion in Price).

Smoothing kernel

Any quantity can be interpolated to the particle locations using the same kernel:

$$\langle A_i \rangle = \sum_{j=1}^{N_{ngb}} \frac{m_j}{\rho_j} A_j W(|\vec{r}_{ij}|, h_i)$$

e.g $\langle v_i \rangle = \sum_{j=1}^{N_{ngb}} \frac{m_j}{\rho_j} v_j W(|\vec{r}_{ij}|, h_i)$ is the smoothed velocity field. The equations of motion (fluid PDEs) can be written in terms of these kernel weighted sums. For a given particle, we find total ("Lagrangian") time derivatives.

Equations of motion

For a given particle, total ("Lagrangian") time derivatives can be computed as kernel-weighted sums over neighbour particles. From the Gadget-2 code paper:

> Starting from a discretised version of the fluid Lagrangian, one can show (Springel & Hernquist, 2002) that the equations of motion for the SPH particles are given by
>
> $$\frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \left[f_i \frac{P_i}{\rho_i^2} \nabla_i W_{ij}(h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W_{ij}(h_j) \right], \quad (7)$$
>
> where the coefficients f_i are defined by
>
> $$f_i = \left[1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right]^{-1}, \quad (8)$$
>
> and the abbreviation $W_{ij}(h) = W(|\boldsymbol{r}_i - \boldsymbol{r}_j|, h)$ has been used. The particle pressures are given by $P_i = A_i \rho_i^{\gamma}$. Provided there are no shocks and no external sources of heat, the equations above already fully define reversible fluid dynamics in SPH. The entropy A_i of each particle stays constant in such a flow.

This is a "symmetrizied" version that accounts for the different smoothing scales in a way that satisfies Newton's 2nd. law. There is more than one technique for this.

A similar equation can be found for the adiabatic contribution to time derivative of the internal energy.

Gadget uses the so-called "entropy" formulation of SPH.

Artificial viscosity

If there are heat inputs into the system (or microphysics like shocks), the entropy will change. Since this isn't included naturally it has to be added as an artificial viscosity term: particles feel a repulsion, slowing them down and converting KE to heat when they get close to each other.

$$\left.\frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t}\right|_{\mathrm{visc}} = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla_i \overline{W}_{ij} \,, \tag{9}$$

where $\Pi_{ij} \geq 0$ is non-zero only when particles approach each other in physical space. The viscosity generates entropy at a rate

$$\frac{\mathrm{d}A_i}{\mathrm{d}t} = \frac{1}{2}\frac{\gamma-1}{\rho_i^{\gamma-1}} \sum_{j=1}^{N} m_j \Pi_{ij} \boldsymbol{v}_{ij} \cdot \nabla_i \overline{W}_{ij} \,, \tag{10}$$

Other things

The signal-velocity approach naturally leads to a Courant-like hydrodynamical timestep of the form

$$\Delta t_i^{(\mathrm{hyd})} = \frac{C_{\mathrm{courant}}\, h_i}{\max_j (c_i + c_j - 3w_{ij})} \tag{16}$$

which is adopted by GADGET-2. The maximum is here determined with respect to all neighbours j of particle i.

Following Balsara (1995) and Steinmetz (1996), GADGET-2 also uses an additional viscosity-limiter to alleviate spurious angular momentum transport in the presence of shear flows. This is done by multiplying the viscous tensor with $(f_i + f_j)/2$, where

$$f_i = \frac{|\nabla \times \boldsymbol{v}|_i}{|\nabla \cdot \boldsymbol{v}|_i + |\nabla \times \boldsymbol{v}|_i} \tag{17}$$

is a simple measure for the relative amount of shear in the flow around particle i, based on standard SPH estimates for divergence and curl (Monaghan, 1992).

Like mesh solvers, SPH solvers have to take timesteps that respect a Courant condition.

And take care with the geometric effects of the viscosity.

SPH shock tube (Gadget-2)

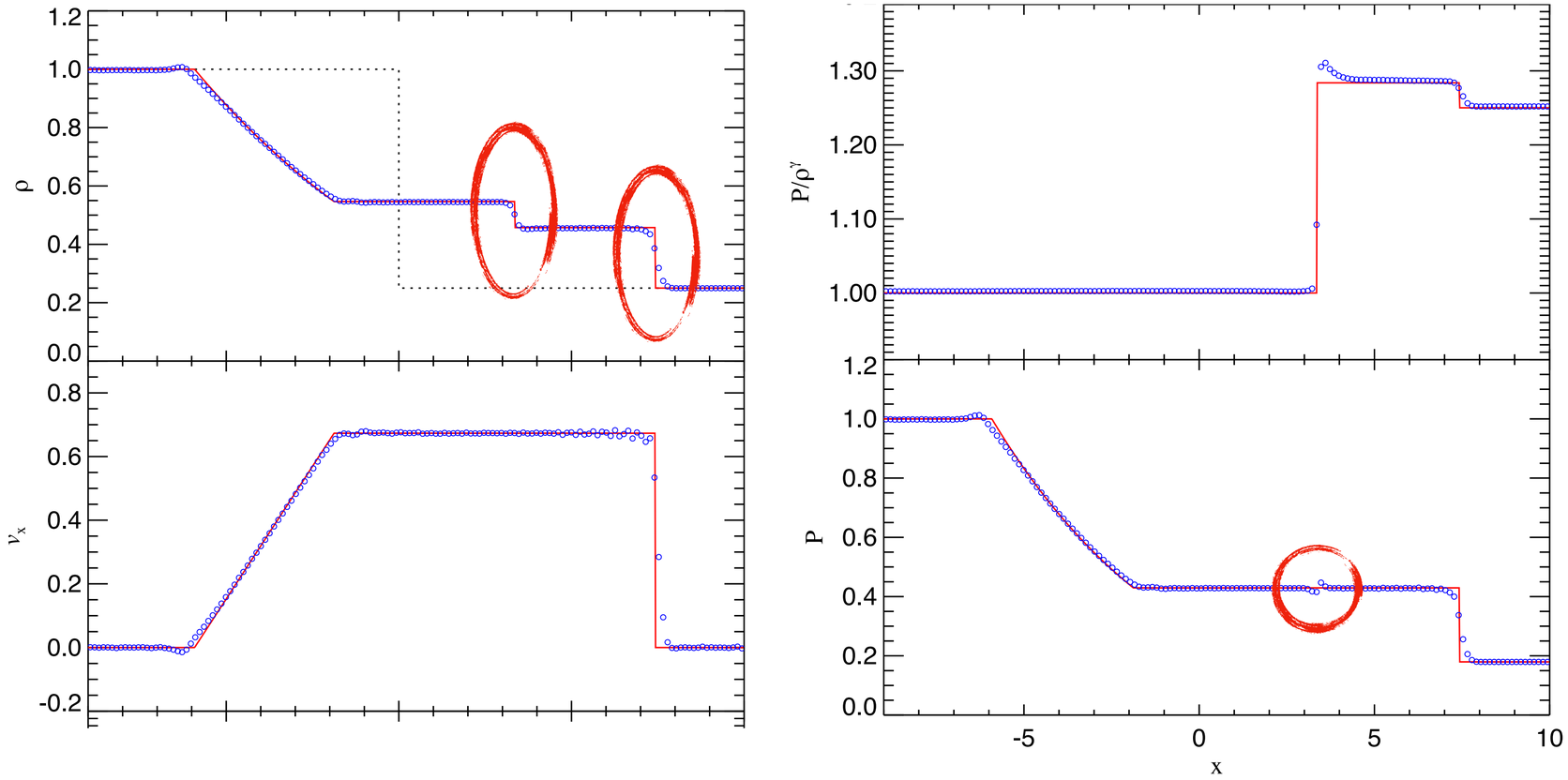


Figure 9. Sod shock test carried out in three dimensions. The gas is initially at rest with $\rho_1 = 1.0$, $P_1 = 1.0$ for $x < 0$, and $\rho_2 = 0.25$, $P_2 = 9.1795$ for $x > 0$. The numerical result is shown with circles (with a spacing equal to the mean particle spacing in the low-density region) and compared with the analytic result at $t = 5.0$. A shock of Mach-number $M = 1.48$ develops.

Limitations of SPH

In the old days (still now) it was common to say that SPH “over-smooths shocks”. This is a bit misleading; SPH smooths over shocks, but so do general mesh-based methods.

In both cases, the solution is using high enough resolution and adding artificial viscosity. The details, of course, are complicated.

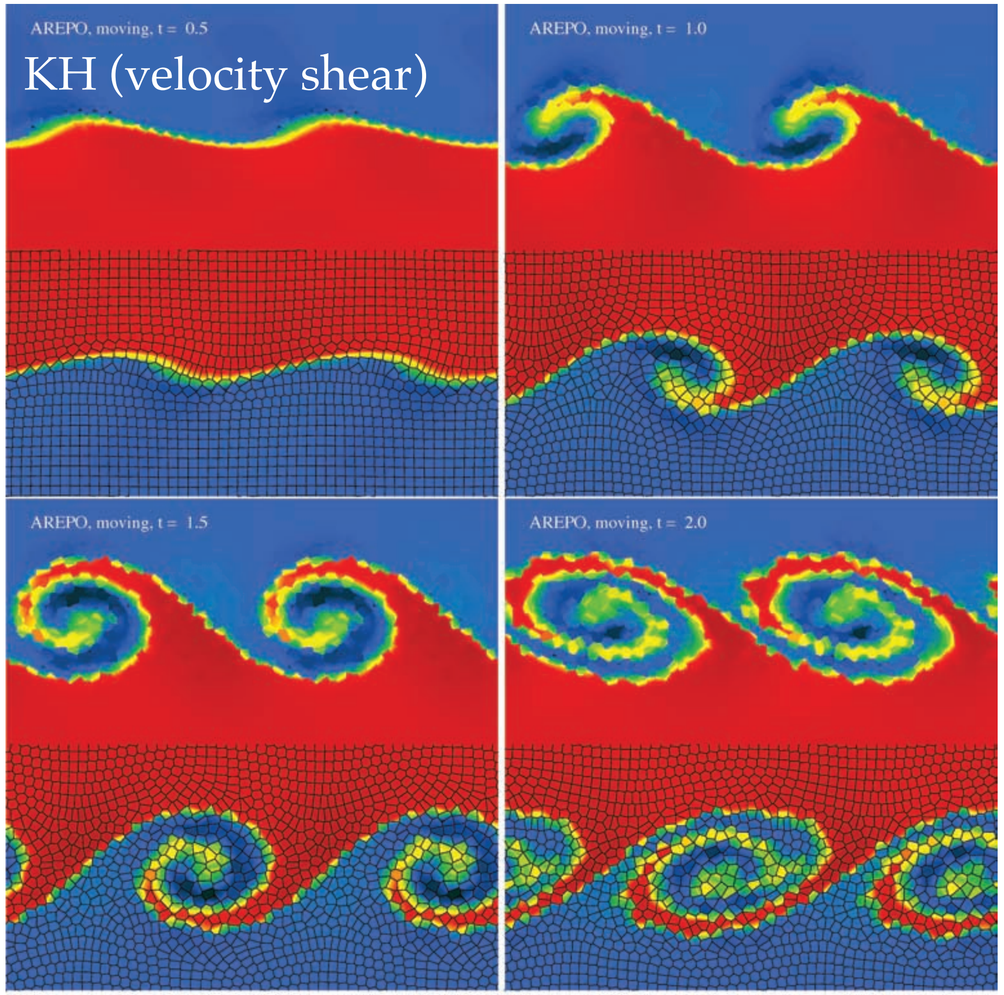
A more significant limitation of SPH is at contact discontinuities. The SPH formalism introduces a spurious “surface tension” that acts against mixing at sharp boundaries between different states.

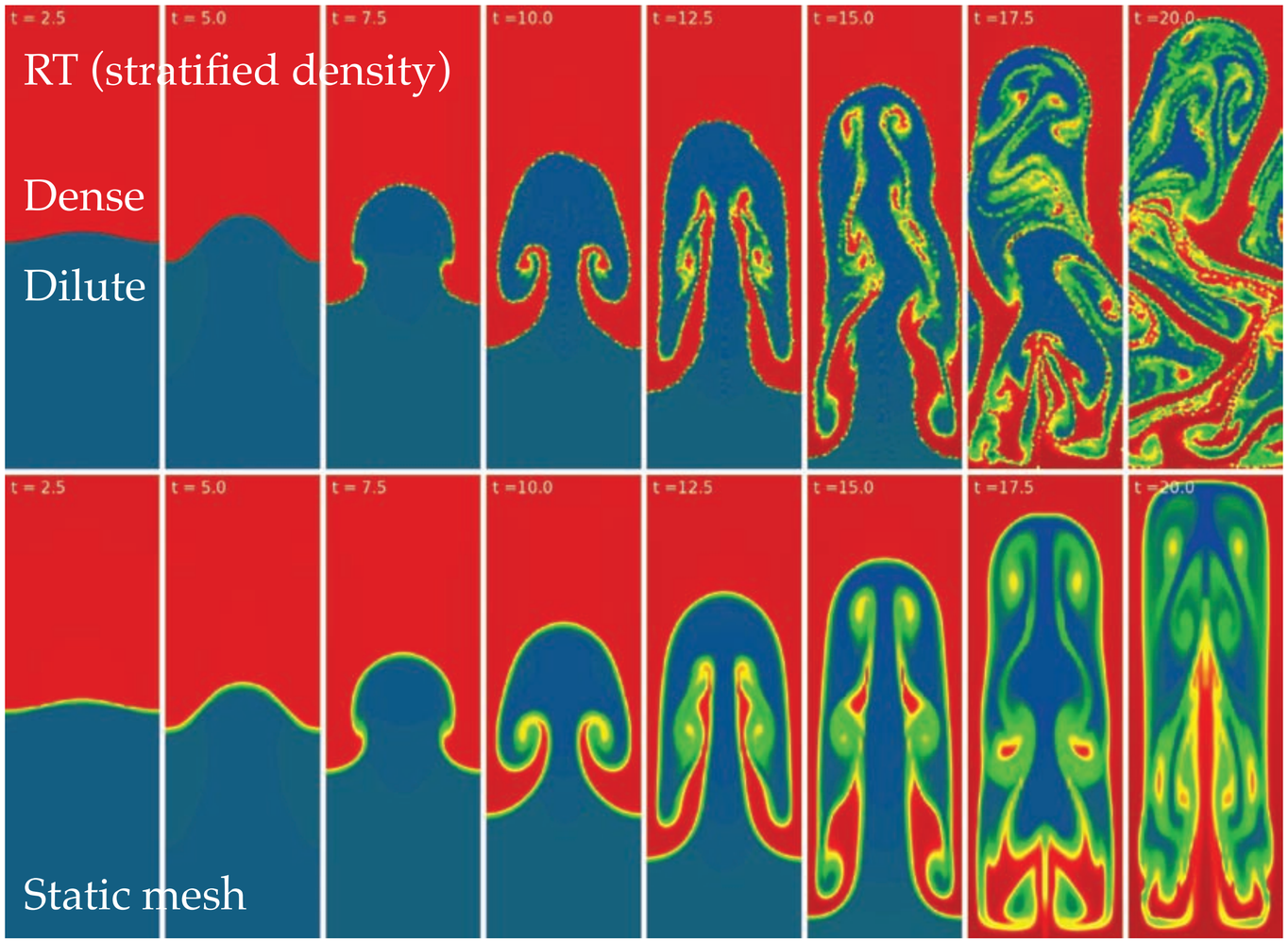
As a consequence, without some extra adjustment, SPH tends to under-develop fluid instabilities, such as the Rayleigh-Taylor and Kelvin-Helmholtz effects. No entropy is produced through mixing.

Unfortunately these instabilities are quite important in astrophysical fluids!

Fluid instabilities

Illustrated by state-of-the-art moving-mesh solutions (Springel 2010; AREPO)





Formulations of SPH

It turns out that the choice of the conserved thermodynamic quantity (energy, entropy) actually makes a difference — not fundamentally, but because of how the equations are written and solved.

Gadget uses the entropy (for reasons see Springel & Hernquist 2002).

Alternative SPH schemes have been proposed (e.g. the **pressure-based SPH** of Hopkins 2013). These improve the treatment of contact discontinuities, but involve some compromises elsewhere.

Photo: 調查局 (!)

White Wagtail
白鶺鴒

ASTR 660 / Class 13

Elliptic PDEs

Elliptic PDEs

Poisson's equation for the gravitational potential generated by a density distribution:

$$\nabla^2 \Phi(\vec{x}) = -4\pi G \rho(\vec{x})$$

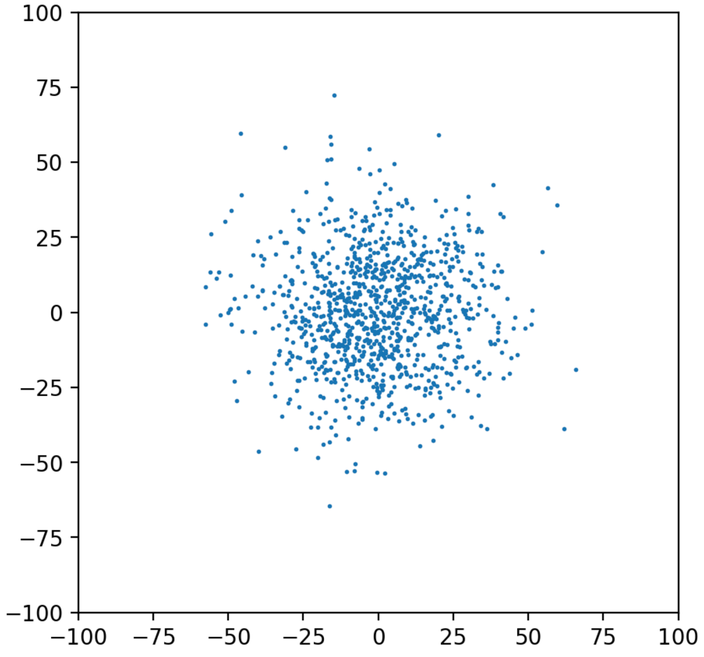
There are no functions of time, so this is a pure "boundary value" problem. In this case the "boundary" might be better called a constraint — the density field, $\rho(\vec{x})$.

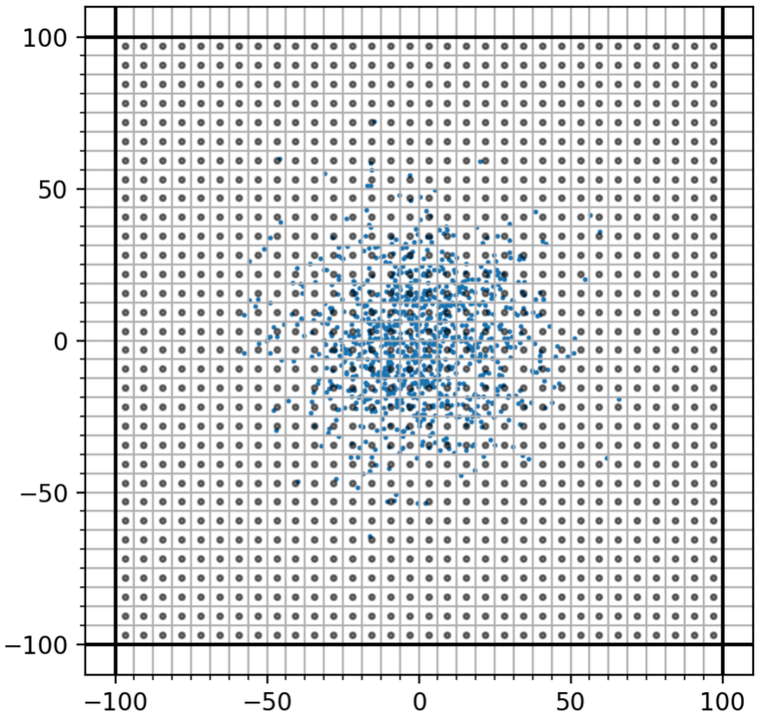
For discrete masses, the potential can be calculated using direct summation, perhaps via an efficient approximation (e.g. a tree). But what if the mass distribution is continuous?

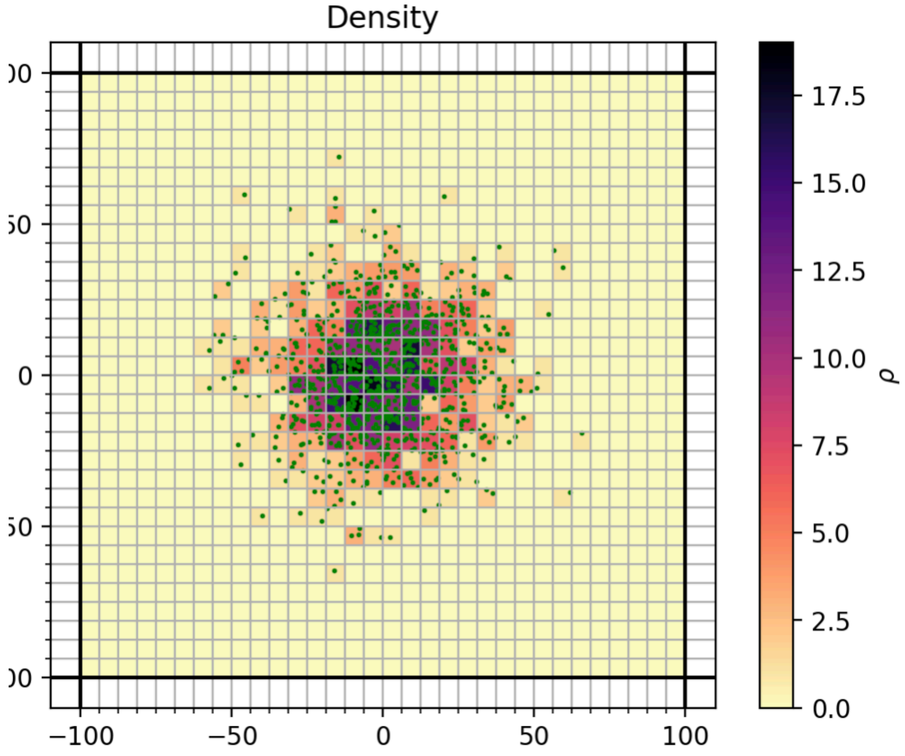
Discretization

The starting point is the assignment of mass to grid points. There are various ways to do this.

For example, map particles to the closest gridpoint (Particle-In-Cell) of smooth particles onto the grid using a kernel (Cloud-In-Cell). All these **smooth** the true distribution.



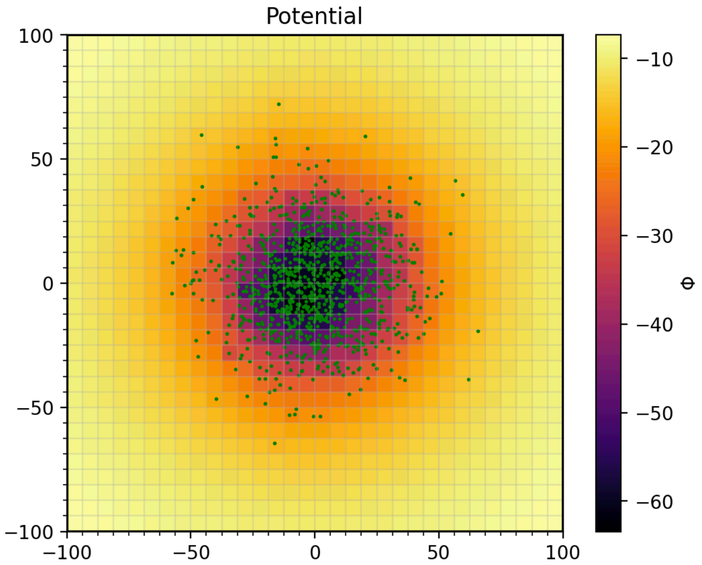




Potential

In the limit of a continuous distribution $\rho(\vec{x})$, the potential at $\vec{x}$ is given by:

$$\Phi(\vec{x}) = -G \int \frac{\rho(\vec{x}')}{|\vec{x} - \vec{x}'|} d^3x$$



For discrete masses,

$$\Phi(\vec{x}) = -G \sum_{i}^{N} \frac{m_i}{|\vec{x} - \vec{x}_i|}$$

Mass mapped to a grid is effectively discrete, with $\vec{x}$ interpreted as a representative position (e.g. the cell midpoint) and m_i as the total mass in the cell.

Potential

We therefore have $\Phi(\vec{x}) = -\sum G_r(\vec{x} - \vec{x}')\, M(x')$ where $G_r(\vec{x} - \vec{x}') = \dfrac{G}{|\vec{x} - \vec{x}'|}$.

This is a **convolution** of two functions, of the form

$$g(t) * h(t) = \int_{-\infty}^{\infty} g(\tau)\, h(t - \tau)\, d\tau$$

where $h(t - \tau)$ is a **kernel** convolved with the function $g(t)$.

In this case G_r is a particular type of kernel known as a Green's function (because it has some physical meaning related to a the solution of a differential equation, in this case Poisson's equation).

Potential

In Fourier space, convolutions become multiplications:

$$\hat{\Phi}(\vec{k}) = \hat{f}(\vec{k}) \cdot \hat{g}(\vec{k})$$

Taking the Fourier transform of G_r from the previous slide gives Poisson's equation in Fourier space:

$$\hat{\phi}_k = -\frac{4\pi G}{k_x^2 + k_y^2}\hat{\rho}_k$$

In principle, we can solve for the potential (on the grid) by taking the Fourier transform of the density field, multiplying it by the transformed Green's function ($\sim 1/k^2$) and then inverse-transforming back to real space.

Fast Fourier Transform

The FFT method is an efficient numerical technique for computing the discrete Fourier transform (see textbooks).

In numpy, this is available through `np.fft`. In Fortran, C etc., there is an industry standard library called FFTW (`module load FFTW` on CICA).

FFT routines are definitely not easy to use: even in simple cases they require a good understanding of what they're doing, the choices that need to be made, and the structure of the output. Reading the documentation is extremely important!

See `examples/fft.ipynb`.

Fourier solutions

The procedure for solving Poisson's equation in Fourier space sounds straightforward, but there are a few technical hurdles, even apart from using the FFT routines correctly.

The biggest headache is the boundary conditions — the FFT method only applies in the case of periodic boundaries.

Finite difference methods

Elliptic PDEs can also be solved by finite-differencing. This approach works in a much wider range of cases, but is usually slower.

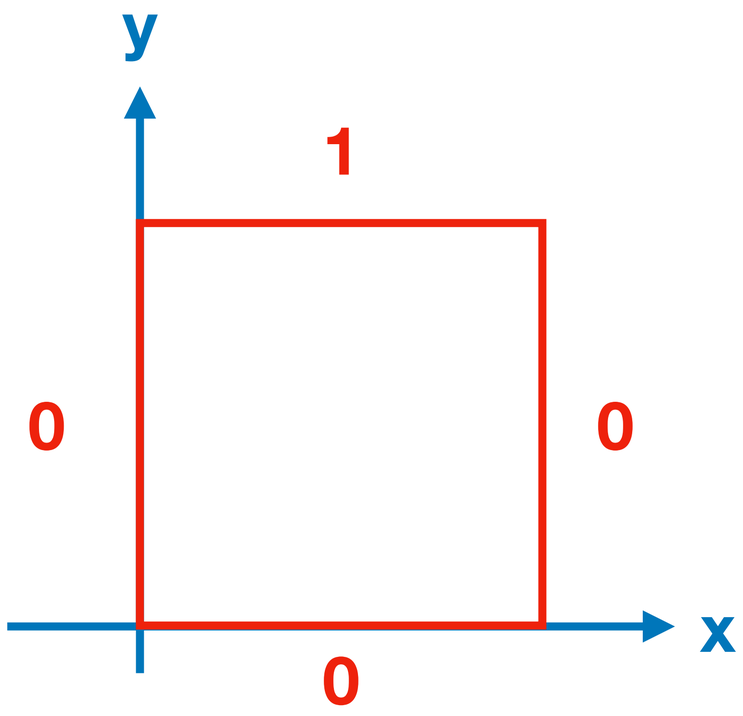
The basic idea is straightforward — write down the solution at each gridpoint as a finite difference, and solve the resulting system of equations.

Example: Laplace equation

$$u_{xx} + u_{yy} = 0$$

Graphics from Kuo-Chuan Pan (after Heath)

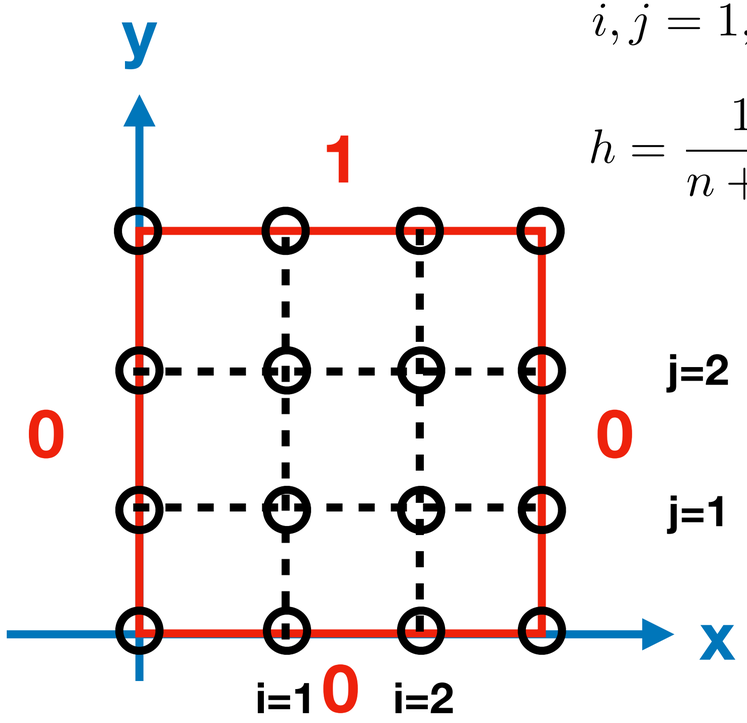
Boundary conditions:



Solution grid: $(x_i, y_j) = (ih, jh)$

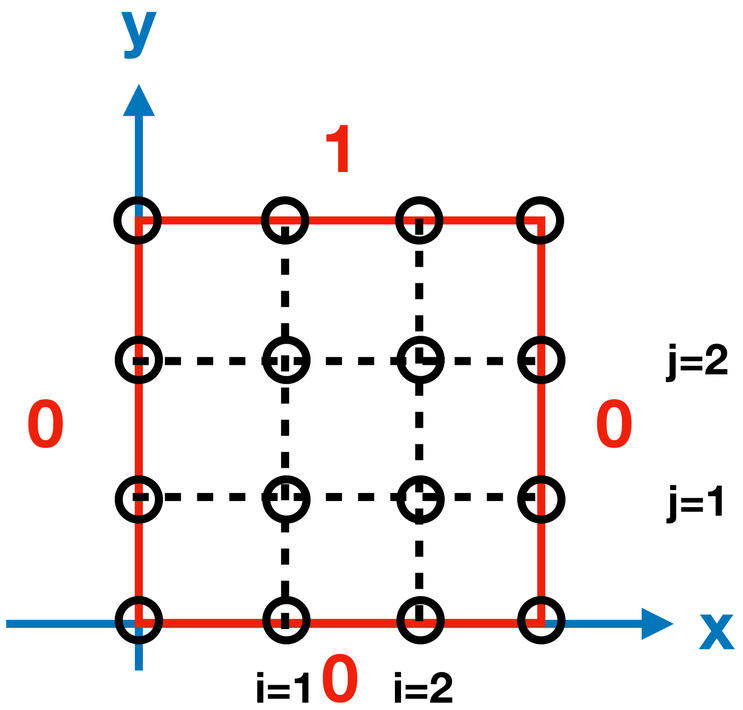
$i, j = 1, 2, ...n = 2$

$h = \frac{1}{n+1} = \frac{1}{3}$



Example: Laplace equation

Graphics from Kuo-Chuan Pan (after Heath)



$$u_{xx} + u_{yy} = 0$$

At each interior gridpoint, we use the FD second derivative approximation:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 0$$

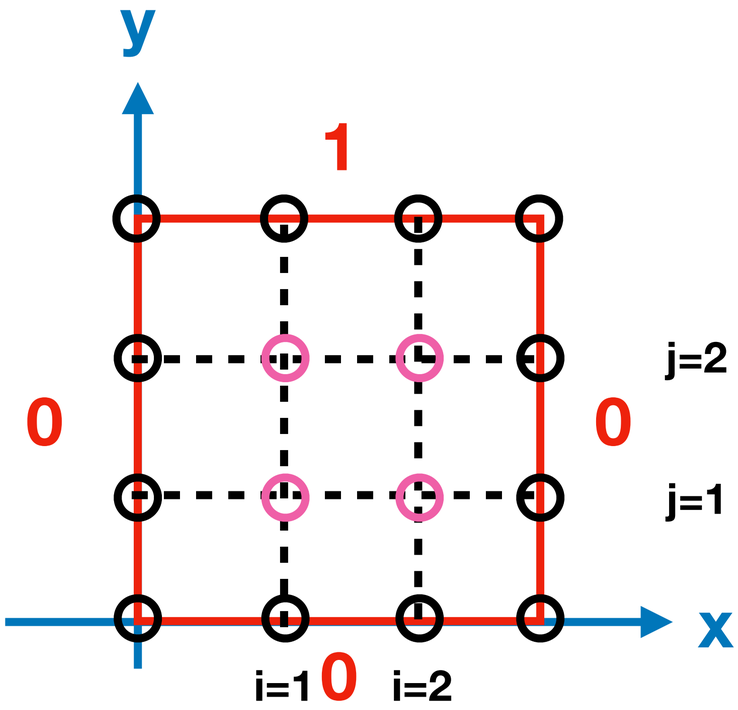
Example: Laplace equation

Graphics from Kuo-Chuan Pan (after Heath)

The solutions at each interior point are connected through the finite differencing scheme. Moreover, we know the solution on the boundary.

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 0$$

$$4u_{1,1} - u_{0,1} - u_{2,1} - u_{1,0} - u_{1,2} = 0$$
$$4u_{2,1} - u_{1,1} - u_{3,1} - u_{2,0} - u_{2,2} = 0$$
$$4u_{1,2} - u_{0,2} - u_{2,2} - u_{1,1} - u_{1,3} = 0$$
$$4u_{2,2} - u_{1,2} - u_{3,2} - u_{2,1} - u_{2,3} = 0$$



Example: Laplace equation

Graphics from Kuo-Chuan Pan (after Heath)

The solutions at each interior point are connected through the finite differencing scheme. Moreover, we know the solution on the boundary.

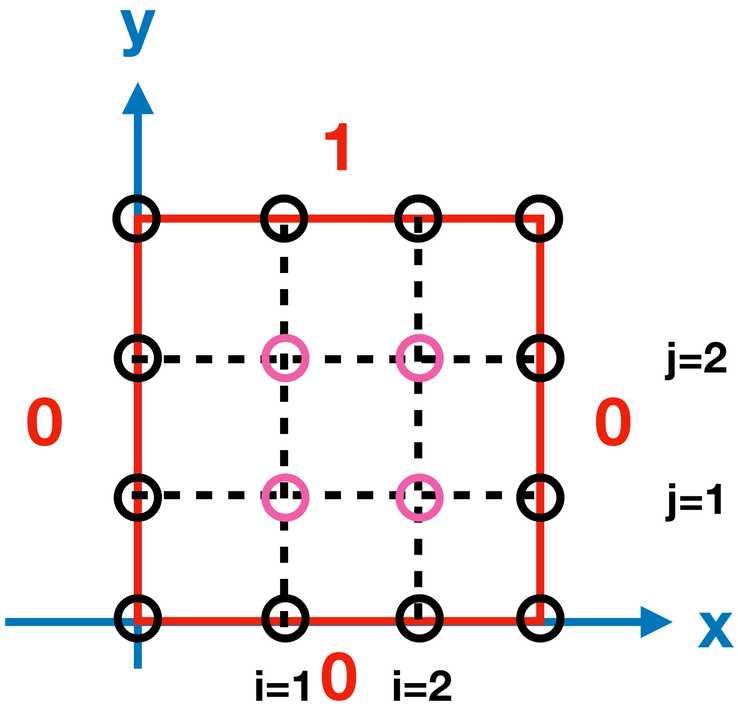
$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 0$$

$$4u_{1,1} - \cancel{u_{0,1}}^{0} - u_{2,1} - \cancel{u_{1,0}}^{0} - u_{1,2} = 0$$

$$4u_{2,1} - u_{1,1} - \cancel{u_{3,1}}^{0} - \cancel{u_{2,0}}^{0} - u_{2,2} = 0$$

$$4u_{1,2} - \cancel{u_{0,2}}^{0} - u_{2,2} - u_{1,1} - u_{1,3} = 0$$

$$4u_{2,2} - u_{1,2} - \cancel{u_{3,2}}^{0} - u_{2,1} - \underline{u_{2,3}}^{1} = 0$$



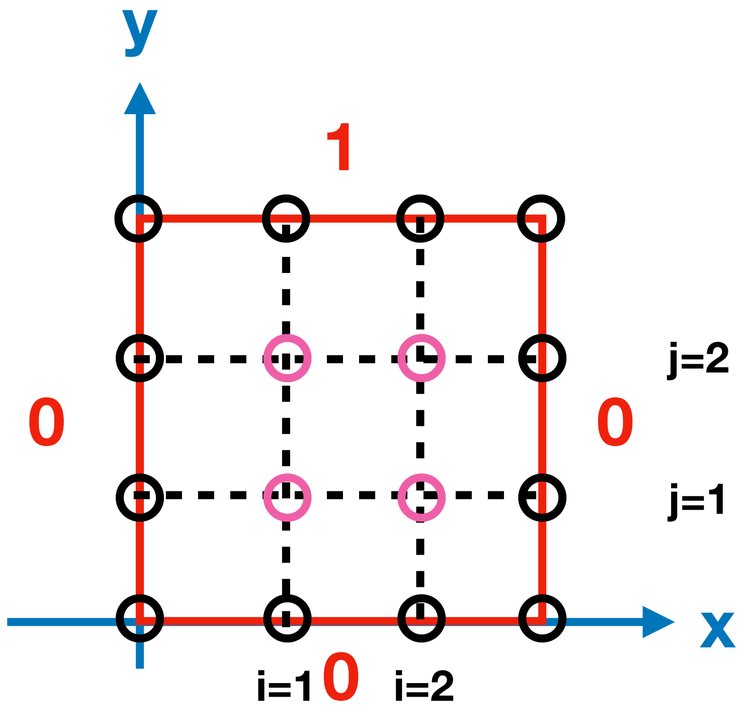
Example: Laplace equation

Graphics from Kuo-Chuan Pan (after Heath)

The solutions at each interior point are connected through the finite differencing scheme. Moreover, we know the solution on the boundary.

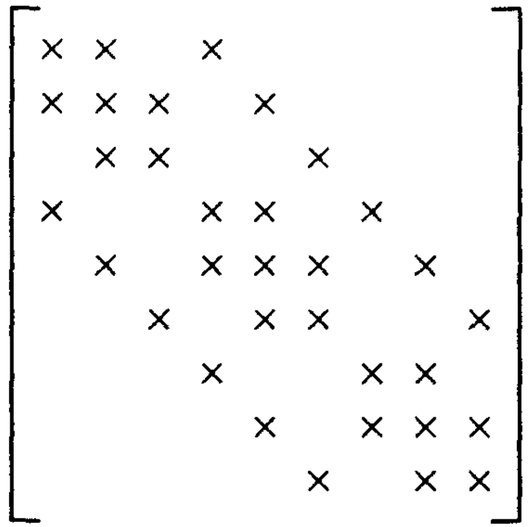
$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 0$$

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_{11} \\ u_{21} \\ u_{12} \\ u_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$



Sparse matrices

The matrices arising from these kinds of problem tend to be **sparse**, i.e. they are mostly zeros. For very large grids it becomes prohibitive to store all those zeros explicitly. It also becomes slow and error-prone to solve such systems with direct methods.



A *Heath*

Iterative methods

Explanation and notation following NR

The general idea here is to guess a solution and then refine the guess until it converges.

If we have an equation $\mathscr{L}u = \rho$, where $\mathscr{L}$ is an operator (e.g. ∇^2) that makes the equation elliptic, then we can consider a "guess" at the answer, u, to be given by the PDE

$$\frac{\partial u}{\partial t} = \mathscr{L}u - \rho$$

This is a "diffusion"-like equation. Diffusion tends to equilibrium. Over time, the left hand sight goes to zero, at which point u satisfies the original equation.

Iterative methods

Explanation and notation mostly following NR

As an example, taking the Laplace/Poisson equation,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - \rho$$

Using the finite differencing representation from above (slightly more generally), we have a relationship between some initial value of the solution and the next update.

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\Delta t}{h^2}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j+1}^{n} - 4u_{i,j}^{n}\right) - \rho_{i,j}\Delta t$$

Iterative methods

Explanation and notation mostly following NR

By an argument similar to the Courant condition, this method (in 2-d) is stable if $\Delta t \leq h^2/4$. Thus taking the biggest Δt possible (and noting the cancellation of the $u_{i,j}$ terms):

$$u_{i,j}^{n+1} = \frac{1}{4}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j+1}^{n}\right) - \frac{h^2}{4}\rho_{i,j}$$

Given a starting guess for the solution at a gridpoint, we can correct it based on the values at its four neighbouring points (and the source term, if there is one).

Iterative methods

Let's try this out… the matrix is:

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_{11} \\ u_{21} \\ u_{12} \\ u_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

which has the solution $\vec{u} = \left[\frac{1}{8}, \quad \frac{1}{8}, \quad \frac{3}{8}, \quad \frac{3}{8}\right]$.

```
A = np.array([[4,-1,-1,0],[-1,4,0,-1],[-1,0,4,-1],[0,-1,-1,4]])
B = np.array([0,0,1,1])
print(np.linalg.solve(A,B))
```

```
array([0.125, 0.125, 0.375, 0.375])
```

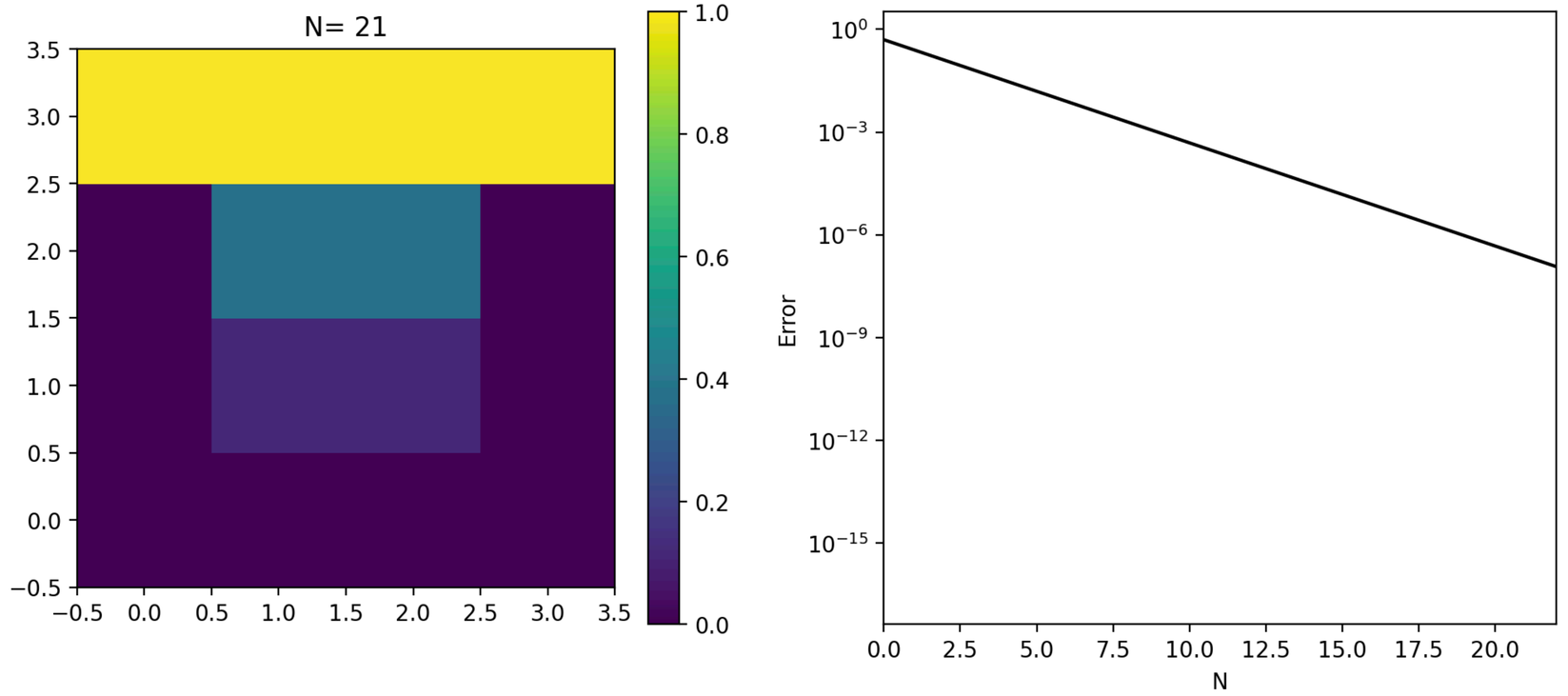
```python
def jacobi_step(u):
    """
    One iteration of the Jacobi method, loop version.
    """
    u0 = u.copy()
    for i in range(1,u.shape[0]-1):
        for j in range(1,u.shape[1]-1):
            u[i,j] =  (1.0/4.0)*(u0[i+1,j] + u0[i-1,j] + u0[i,j+1] + u0[i,j-1])
    return u

def iterate(u,stepper,atol=1e-6,nmax=100):
    """
    Interate until the absolute difference between
    steps is less than abstol, or nmax is reached.
    """
    for n in range(0,nmax):
        u0 = u.copy()
        u  = stepper(u)

        if np.all(np.abs(u-u0) < atol):
            break

    if n < nmax-1:
        print('Converged in {:d} steps'.format(n))
        print('Max absolute error: {:g}'.format(np.max(u-u0)))
    else:
        print('Failed to converge, n > nmax = {:d}'.format(nmax))
        print('Max absolute error: {:g}'.format(np.max(u-u0)))

    return u
```

```python
nx = 4
ny = 4
u  = np.zeros((nx,ny))

# Boundary conditions
u[0,:]  = 0
u[-1,:] = 0
u[:,0]  = 0
u[:,-1] = 1
```

```python
iterate(u, jacobi_step, atol=1e-7, nmax=100)
```

```
Converged in 21 steps
Max absolute error: 5.96046e-08

array([[0.        , 0.        , 0.        , 1.        ],
       [0.        , 0.12499994, 0.37499994, 1.        ],
       [0.        , 0.12499994, 0.37499994, 1.        ],
       [0.        , 0.        , 0.        , 1.        ]])
```

Converged in 21 steps
Max absolute error: 5.96046e-08



Iterative methods

Explanation and notation mostly following NR

This is all good, but extremely slow to converge for large problems.

```
u0 = setup(40,40)
%time u,n = iterate(u0, jacobi_step, atol=1e-7, nmax=10000)
```

```
Converged in 2920 steps
Max absolute error: 9.96834e-08
CPU times: user 1.71 s, sys: 18 ms, total: 1.73 s
Wall time: 1.73 s
```

```
u0 = setup(100,100)
%time u,n = iterate(u0, jacobi_step, atol=1e-7, nmax=100000)
```

```
Converged in 15133 steps
Max absolute error: 9.99758e-08
CPU times: user 57.6 s, sys: 755 ms, total: 58.4 s
Wall time: 58.6 s
```

Faster iterative methods

Explanation and notation mostly following NR

Of course, various improvements can be made to the algorithm to speed things up, but they are all fundamentally quite “slow”.

The Gauss-Seidel method and the Successive Over-Relaxation methods are easy to implement. However, they are only faster for a limited range of problems (like this one).

In general the approach to a fast differencing algorithm for elliptic equations proceeds to “multigrid methods” (see NR) which are beyond the scope of this course.

Faster iterative methods: example

Explanation and notation mostly following NR

The Gauss-Seidel method uses the "improved" estimate for grid cells as soon as they are computed:

```
def gs_step(u):
    """
    One iteration of the Gauss-Seidel method, loop version.
    """
    u0 = u.copy()
    for i in range(1,u.shape[0]-1):
        for j in range(1,u.shape[1]-1):
            u[i,j] =  (1.0/4.0)*(u0[i+1,j] + u[i-1,j] + u0[i,j+1] + u[i,j-1])

    max_abs_err = np.max(np.abs(u-u0))
    return u,max_abs_err
```

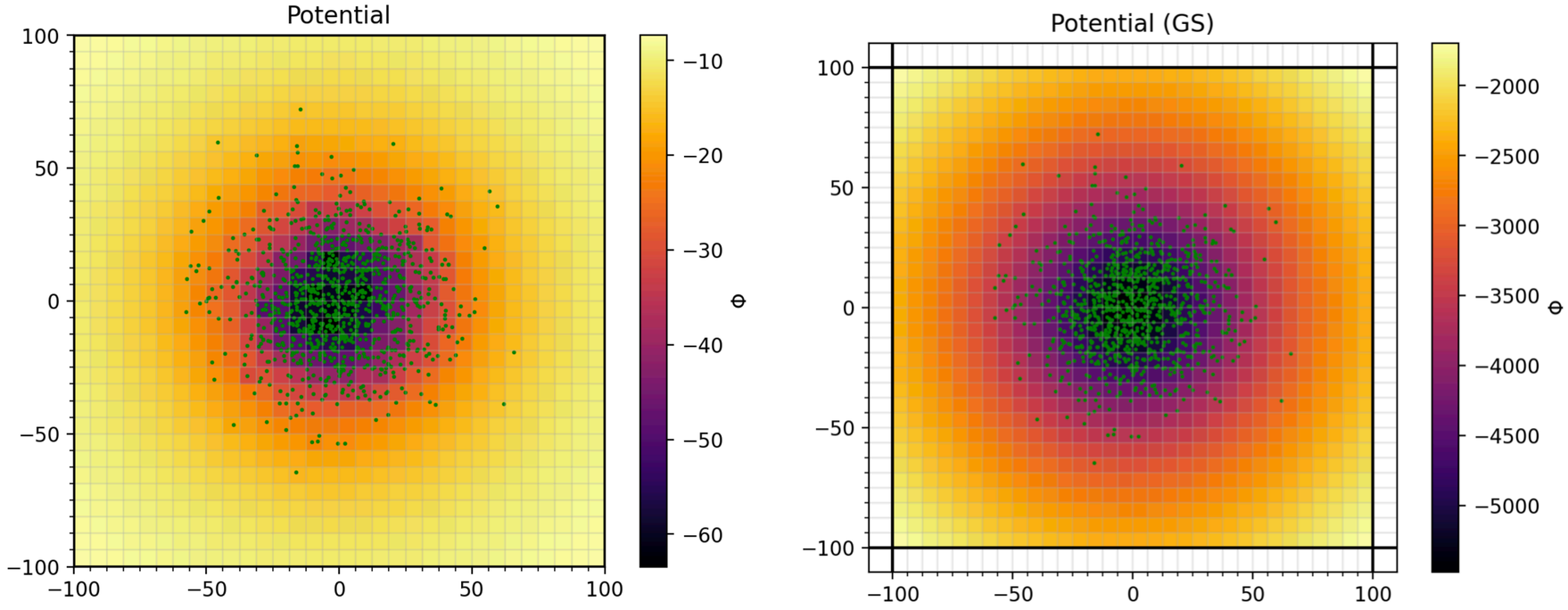
```
u0 = setup(100,100)
%time u,n = iterate(u0, gs_step, atol=1e-7, nmax=100000)
```

```
Converged in 8280 steps
Max absolute error: 9.99275e-08
CPU times: user 31.8 s, sys: 322 ms, total: 32.1 s
Wall time: 32.2 s
```

Applications to Poisson's equation

Again, this is not quite as easy as it sounds, probably because of the boundary conditions.

See `examples/elliptical/elliptical.ipynb`



Sources for today's slides

These slides build on and adapt previous lecture notes for this course, by Kuo-Chuan Pan, Karen Yang, and Hsi-Yu Schive (NTU). Much of the material on iterative solvers is taken from the textbook by Heath, and the material on SPH from various papers by Volker Springel and the SPH notes by Daniel Price (links on Wiki).