Title: Lecture - Numerical Methods, PHYS 777

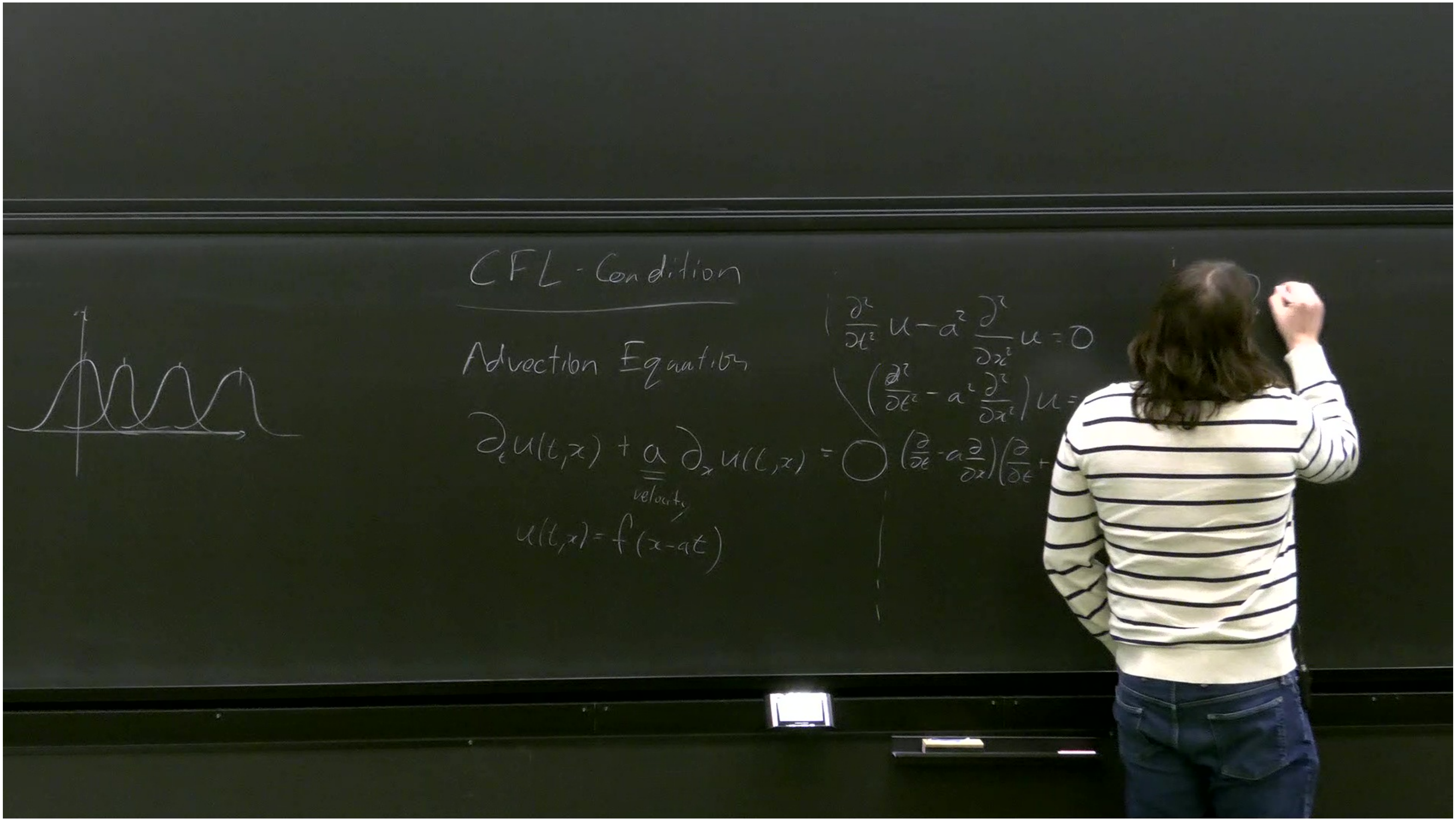
Speakers: Erik Schnetter, Dustin Lang

Collection/Series: Numerical Methods (Core), PHYS 777-, January 6 - February 5, 2025

Subject: Other

Date: January 23, 2025 - 10:15 AM

URL: https://pirsa.org/25010060


CFL - Condition
Advection Equation
$\partial_t u(t,x) + a \partial_x u(t,x) = 0$
velocity
$u(t,x) = f(x-at)$
$\frac{\partial^2}{\partial t^2} u - a^2 \frac{\partial^2}{\partial x^2} u = 0$
$\left(\frac{\partial^2}{\partial t^2} - a^2 \frac{\partial^2}{\partial x^2}\right) u =$
$\left(\frac{\partial}{\partial t} - a \frac{\partial}{\partial x}\right)\left(\frac{\partial}{\partial t} +\right.$

…on

…tion

$\partial_x u(t,x) = 0$

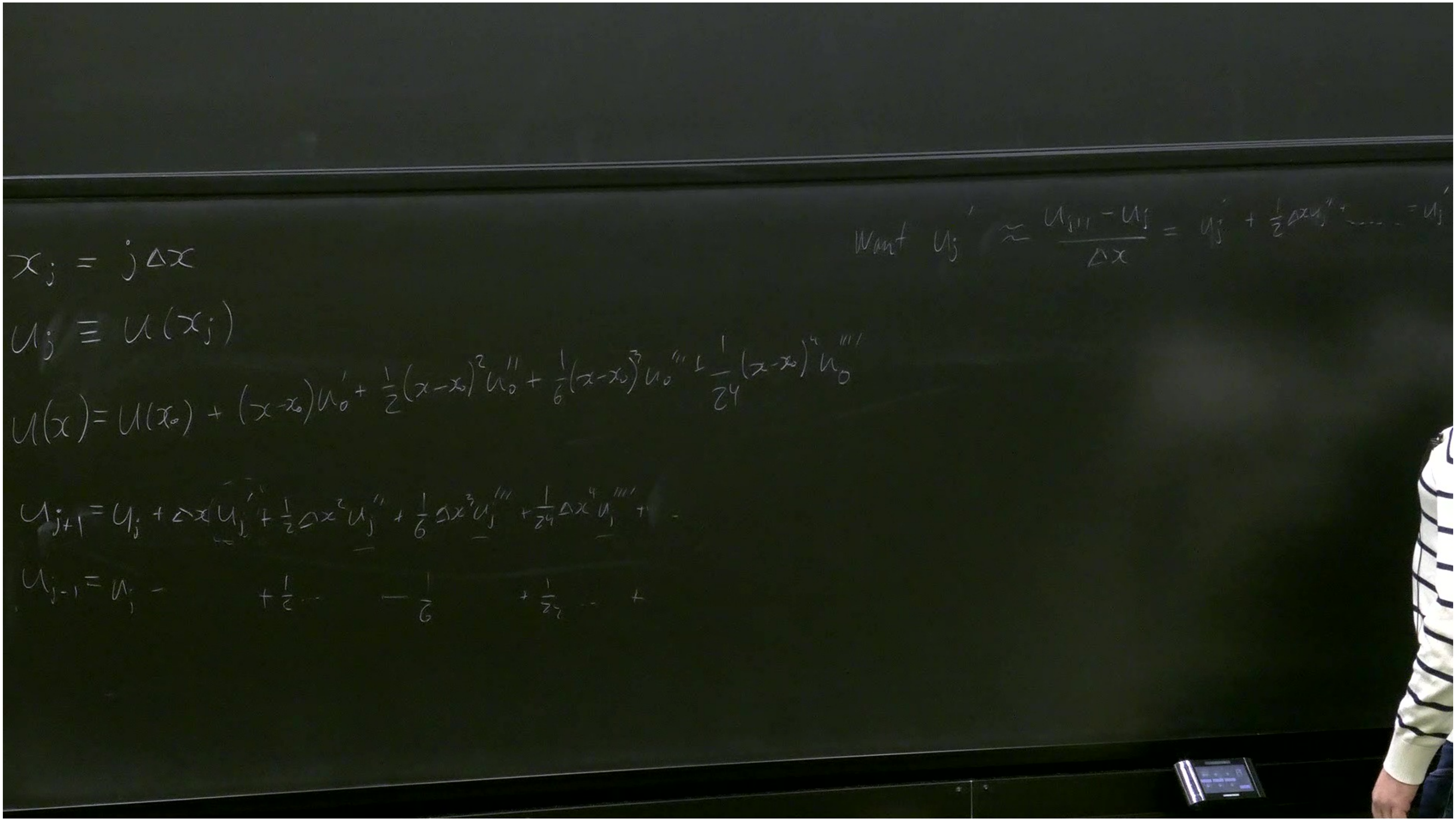
…t)

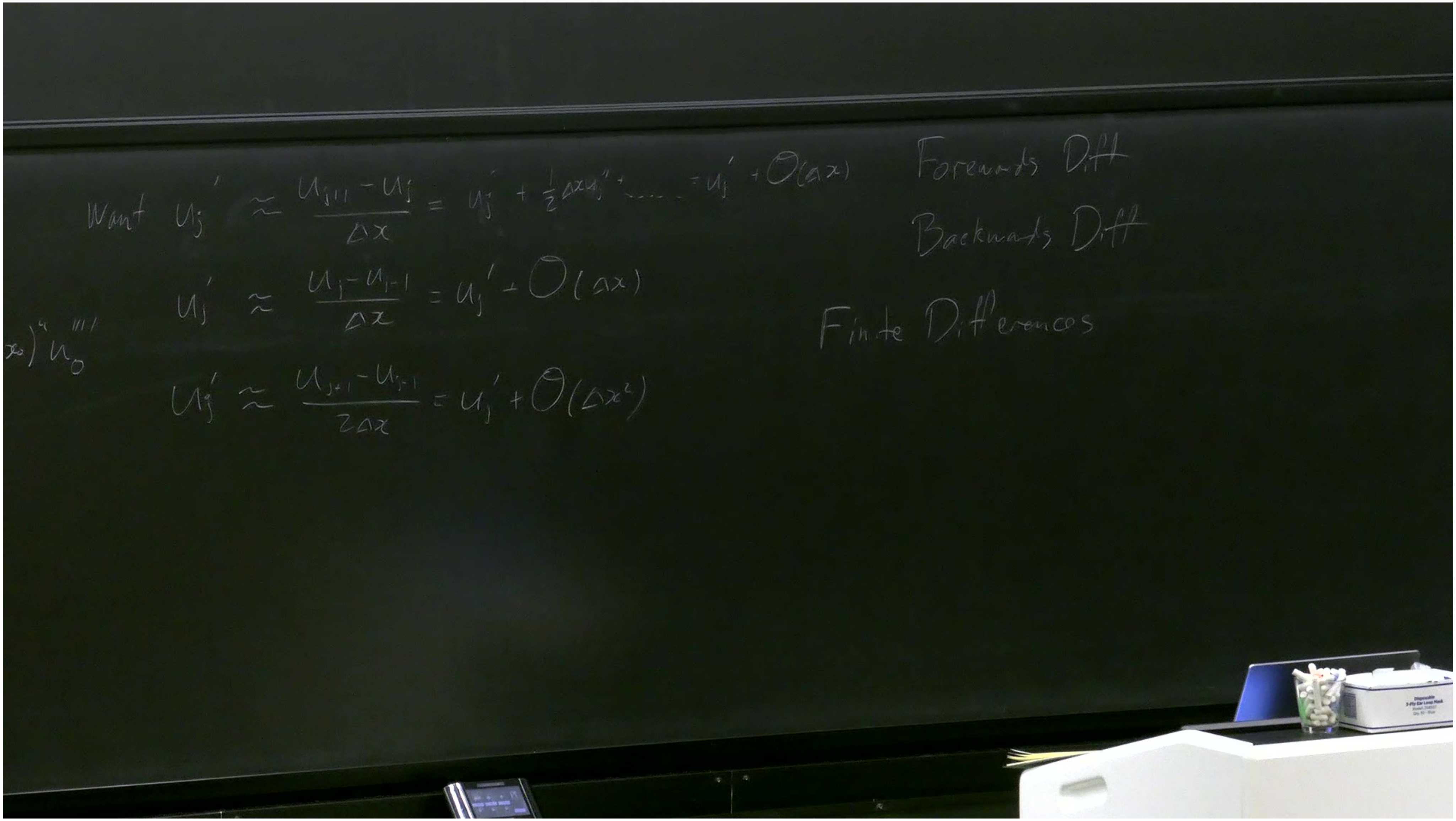
$$\frac{\partial^2}{\partial t^2} u - a^2 \frac{\partial^2}{\partial x^2} u = 0$$

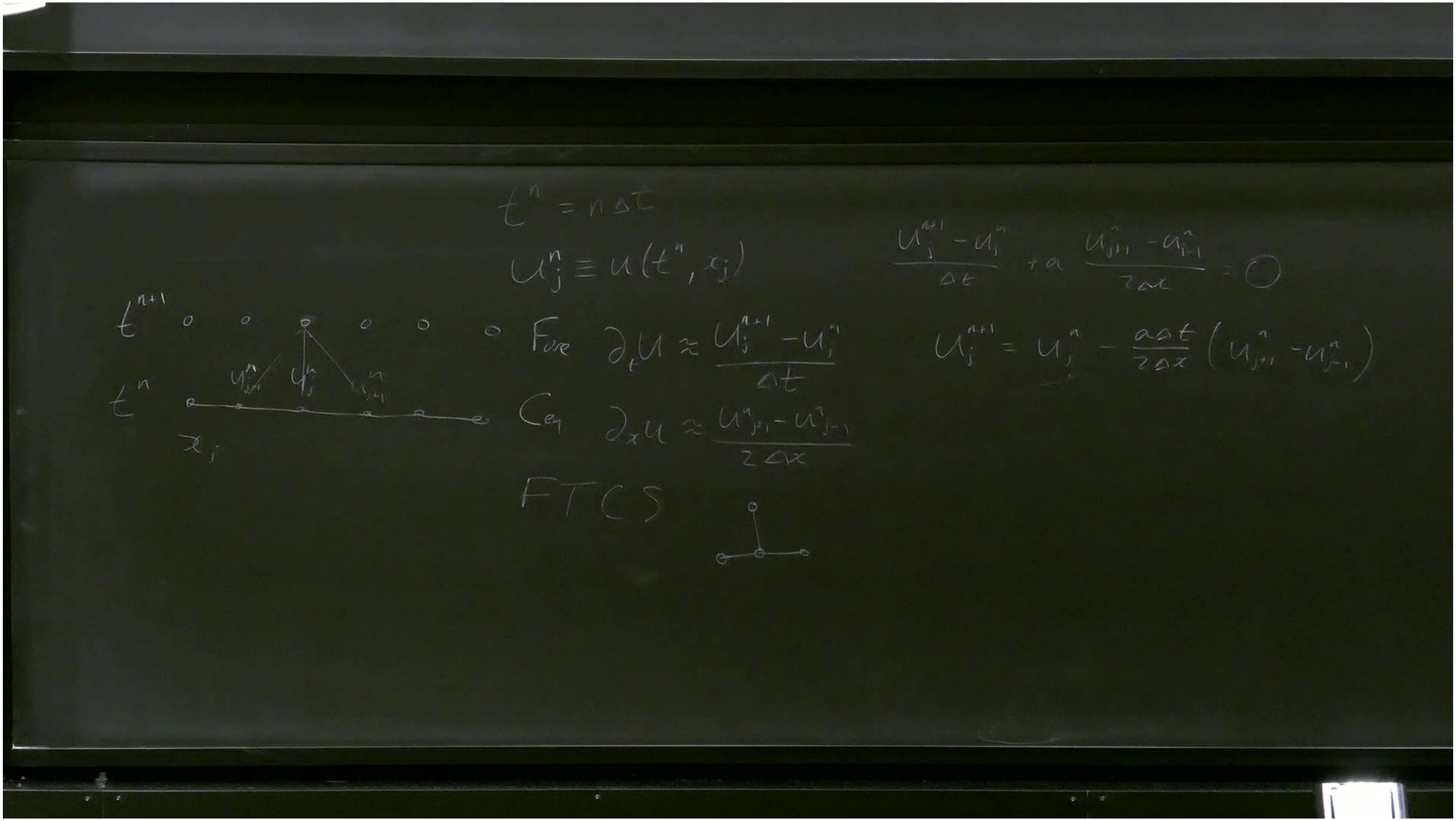
$$\left(\frac{\partial^2}{\partial t^2} - a^2 \frac{\partial^2}{\partial x^2}\right) u = 0$$

$$\left(\frac{\partial}{\partial t} - a\frac{\partial}{\partial x}\right)\left(\frac{\partial}{\partial t} + a\frac{\partial}{\partial x}\right) u = 0$$

$$0 = \nabla_\mu J^\mu$$
$$= \nabla_\mu (\rho_0 u^\mu)$$
$$= \partial_t(\rho_0 \gamma) + \partial_x(\rho_0 \gamma v)$$
$$\overset{v=\text{const}}{\Longrightarrow} \; = \partial_t \rho_0 + v\, \partial_x \rho_0$$


Want $u_j' \approx \frac{u_{j+1}-u_j}{\Delta x} = u_j' + \frac{1}{2}\Delta x u_j'' + \dots = u_j'$
$x_j = j\Delta x$
$u_j \equiv u(x_j)$
$u(x) = u(x_0) + (x-x_0)u_0' + \frac{1}{2}(x-x_0)^2 u_0'' + \frac{1}{6}(x-x_0)^3 u_0''' + \frac{1}{24}(x-x_0)^4 u_0''''$
$u_{j+1} = u_j + \Delta x u_j' + \frac{1}{2}\Delta x^2 u_j'' + \frac{1}{6}\Delta x^3 u_j''' + \frac{1}{24}\Delta x^4 u_j'''' + \dots$
$u_{j-1} = u_j - \quad + \frac{1}{2}\dots \quad - \frac{1}{6} \quad + \frac{1}{24}\dots +$


Forewards Diff
Backwards Diff
Finite Differences
Want $u_j' \approx \frac{u_{j+1} - u_j}{\Delta x} = u_j' + \frac{1}{2}\Delta x u_j'' + \ldots = u_j' + O(\Delta x)$
$u_j' \approx \frac{u_j - u_{j-1}}{\Delta x} = u_j' + O(\Delta x)$
$u_j' \approx \frac{u_{j+1} - u_{j-1}}{2\Delta x} = u_j' + O(\Delta x^2)$


$t^n = n\Delta t$
$u_j^n \equiv u(t^n, x_j)$
t^{n+1}
t^n
u_{j-1}^n u_j^n u_{j+1}^n
x_j
Fwe $\partial_t u \approx \frac{u_j^{n+1} - u_j^n}{\Delta t}$
Cen $\partial_x u \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$
FTCS
$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\,\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} = 0$
$u_j^{n+1} = u_j^n - \frac{a\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right)$

advection_howto

localhost:8888/notebooks/advection_howto.ipynb

Jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help

Trusted

Markdown

JupyterLab Julia 1.11.1

algorithms. Before you can use your algorithm for hydrodynamics, you have to show it can deal with advection.

Here we build a simple method to solve this equation to 1st order in space and time. More sophisticated methods can go to substantially higher order.

Spatial Grid

First we need to define our discretization of space.

This is a "cell-centered" grid that breaks up the interval $[a, b]$ into N_x cells of width $\Delta x = (b-a)/N_x$. There are $N_x + 1$ *faces* in between the cells: $x_{e,i} = a + i\Delta x$ and N_x cell centers at $x_i = a + (i + 1/2)\Delta x$. The `make_grid` function also allows for N_g *ghost zones* on each side of the boundary, which are useful for applying boundary conditions.

[17]:
```
function make_grid(a, b, Nx, Ng)

    # The desired cell-width
    dx = (b - a) / Nx

    # First get the edges of all the cells, adding ghost zones to the left and right
    xe = LinRange(a - Ng*dx, b + Ng*dx, Nx + 2*Ng + 1)

    # Now the cell centers are just the average of the left & right cell faces
    xL = xe[begin:end-1]
    xR = xe[begin+1:end]
    x = 0.5*(xL + xR)

    return dx, x, xe
end
```

[17]:
```
make_grid (generic function with 1 method)
```

Plotting Utilities

Just a couple useful functions for plotting our data.

You may have to change GLMakie to WGLMakie

[2]:
```
using GLMakie
GLMakie.activate!()

function make_figure()
    fig = Figure()
    ax = Axis(fig[1,1])
    return fig, ax
end

function add_to_plot(ax, x, f)
```

advection_howto

localhost:8888/notebooks/advection_howto.ipynb

jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help Trusted

Markdown JupyterLab Julia 1.11.1

This is a "cell-centered" grid that breaks up the interval $[a, b]$ into N_x cells of width $\Delta x = (b - a)/N_x$. There are $N_x + 1$ *faces* in between the cells: $x_{e,i} = a + i\Delta x$ and N_x cell centers at $x_i = a + (i + 1/2)\Delta x$. The `make_grid` function also allows for N_g *ghost zones* on each side of the boundary, which are useful for applying boundary conditions.

[17]:
```
function make_grid(a, b, Nx, Ng)

    # The desired cell-width
    dx = (b - a) / Nx

    # First get the edges of all the cells, adding ghost zones to the left and right
    xe = LinRange(a - Ng*dx, b + Ng*dx, Nx + 2*Ng + 1)

    # Now the cell centers are just the average of the left & right cell faces
    xL = xe[begin:end-1]
    xR = xe[begin+1:end]
    x = 0.5*(xL + xR)

    return dx, x, xe
end
```

[17]:
```
make_grid (generic function with 1 method)
```

Plotting Utilities

Just a couple useful functions for plotting our data.

advection_howto

localhost:8888/notebooks/advection_howto.ipynb

jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help Trusted

Code JupyterLab Julia 1.11.1

[5]: evolve (generic function with 1 method)

Per-Cell update function -- where the magic happens!

This function gets called on each cell to compute the time-derivative: $\partial_t u = -a\partial_x u$

[21]:
```
function udot_ftcs(uL, uC, uR, a, dx)
    # Write a function that computes du/dt for the advection equation.
    # uR = u_{j+1}
    # uC = u_j
    # uL = u_{j-1}
    return -a * (uR - uL) / (2*dx)
end

function udot_ftls(uL, uC, uR, a, dx)
    # Write a function that computes du/dt for the advection equation.
    # uR = u_{j+1}
    # uC = u_j
    # uL = u_{j-1}
    return -a * (uC - uL) / (dx)
end
```

[21]: udot_ftls (generic function with 1 method)

advection_howto

localhost:8888/notebooks/advection_howto.ipynb

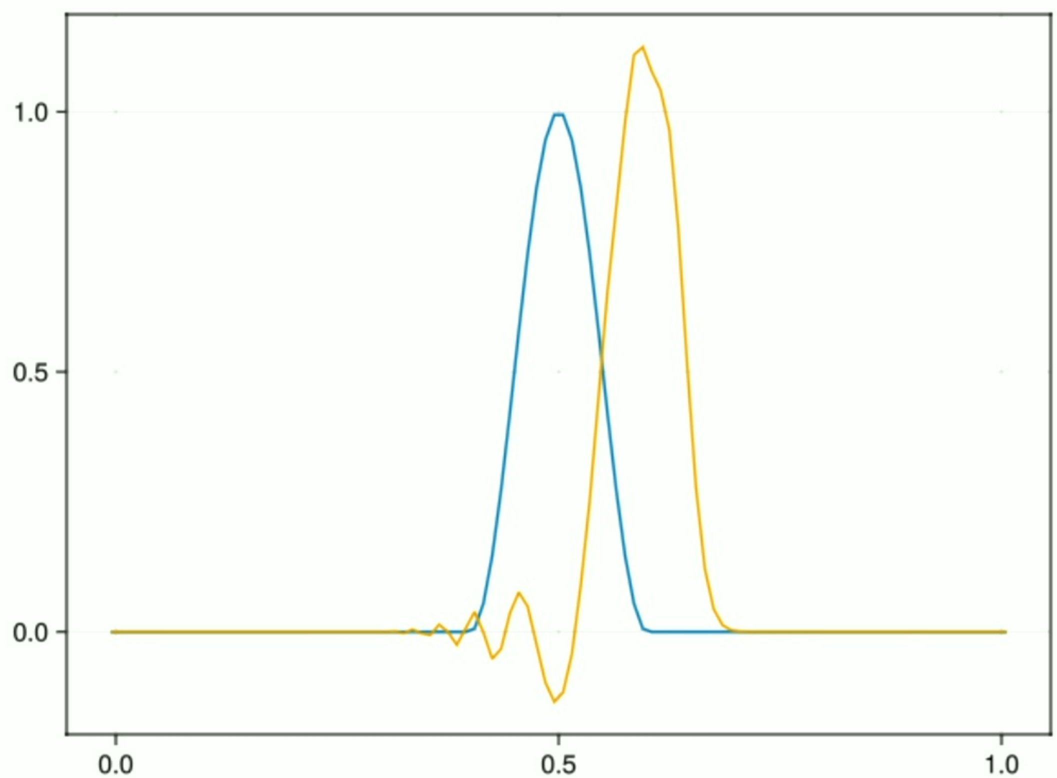
jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help Trusted

Markdown JupyterLab Julia 1.11.1

```
fig
```

[24]:



advection_howto

localhost:8888/notebooks/advection_howto.ipynb

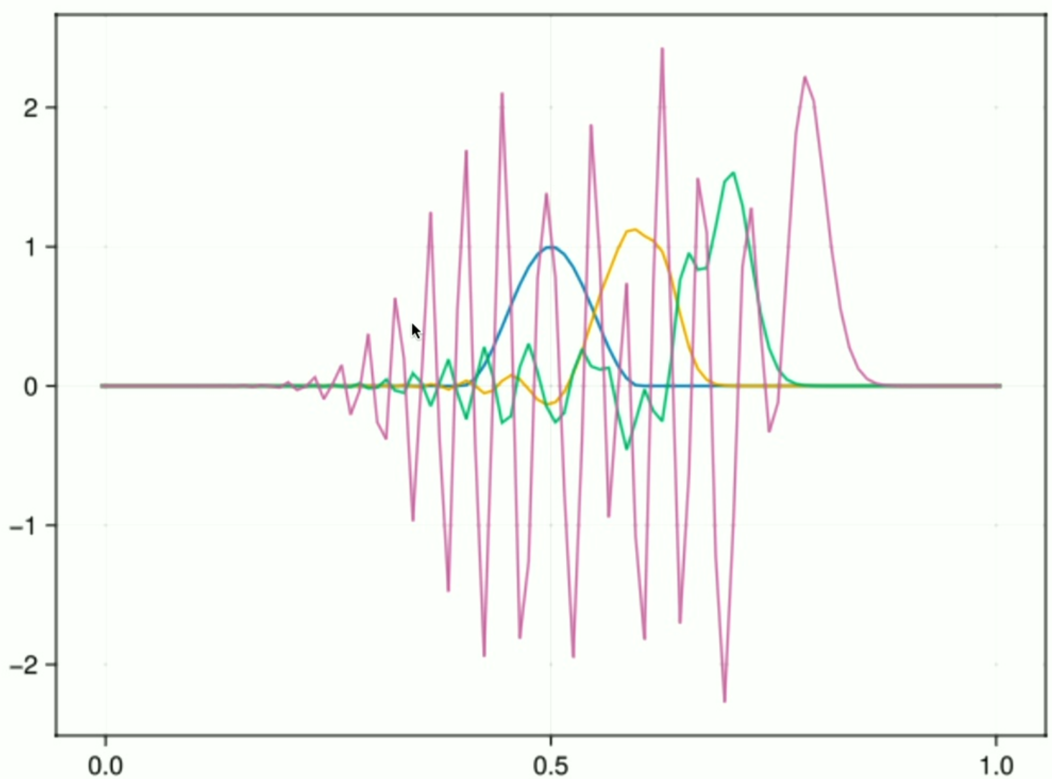
jupyter advection_howto Last Checkpoint: 1 hour ago

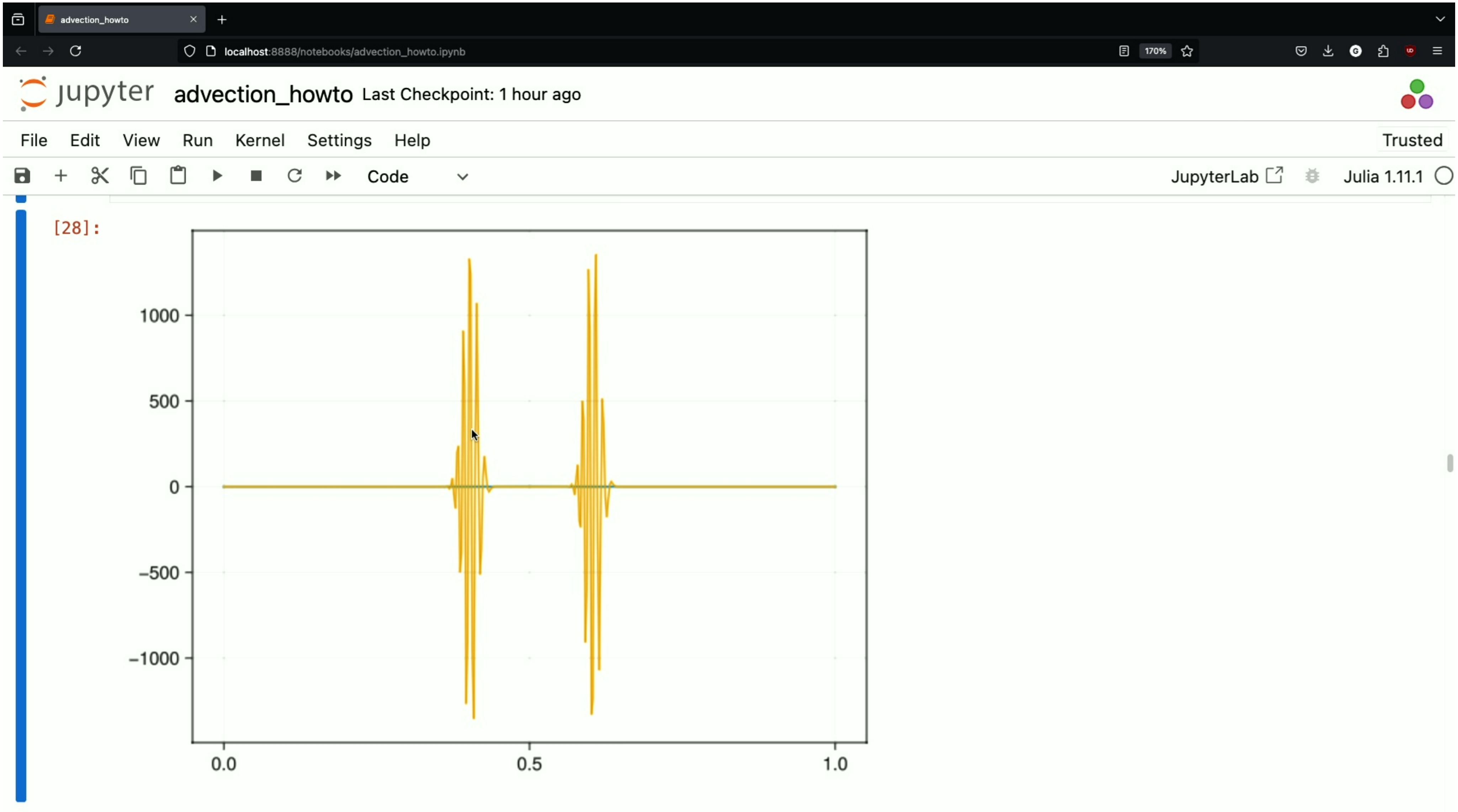
File Edit View Run Kernel Settings Help Trusted

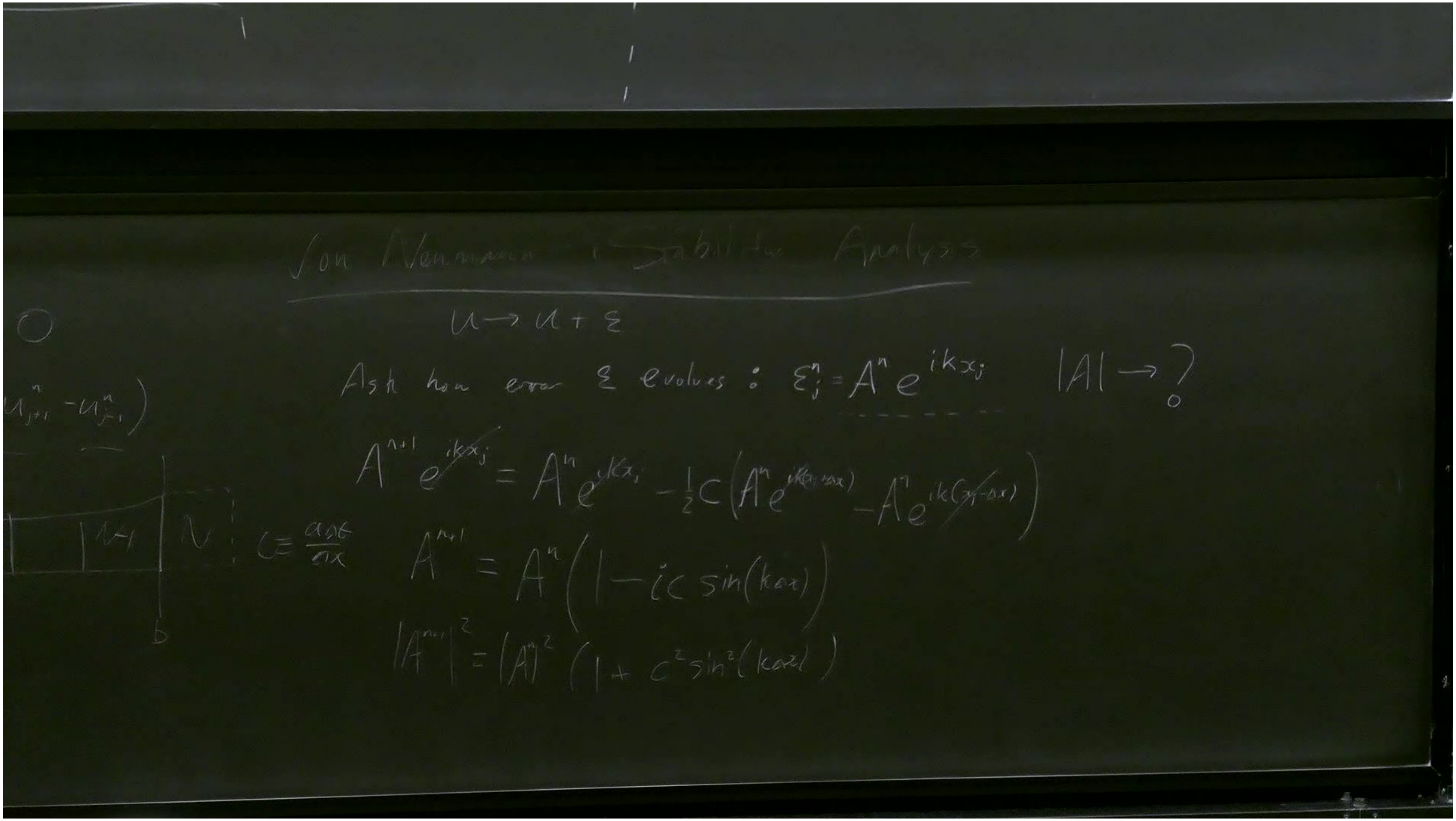
Markdown JupyterLab Julia 1.11.1

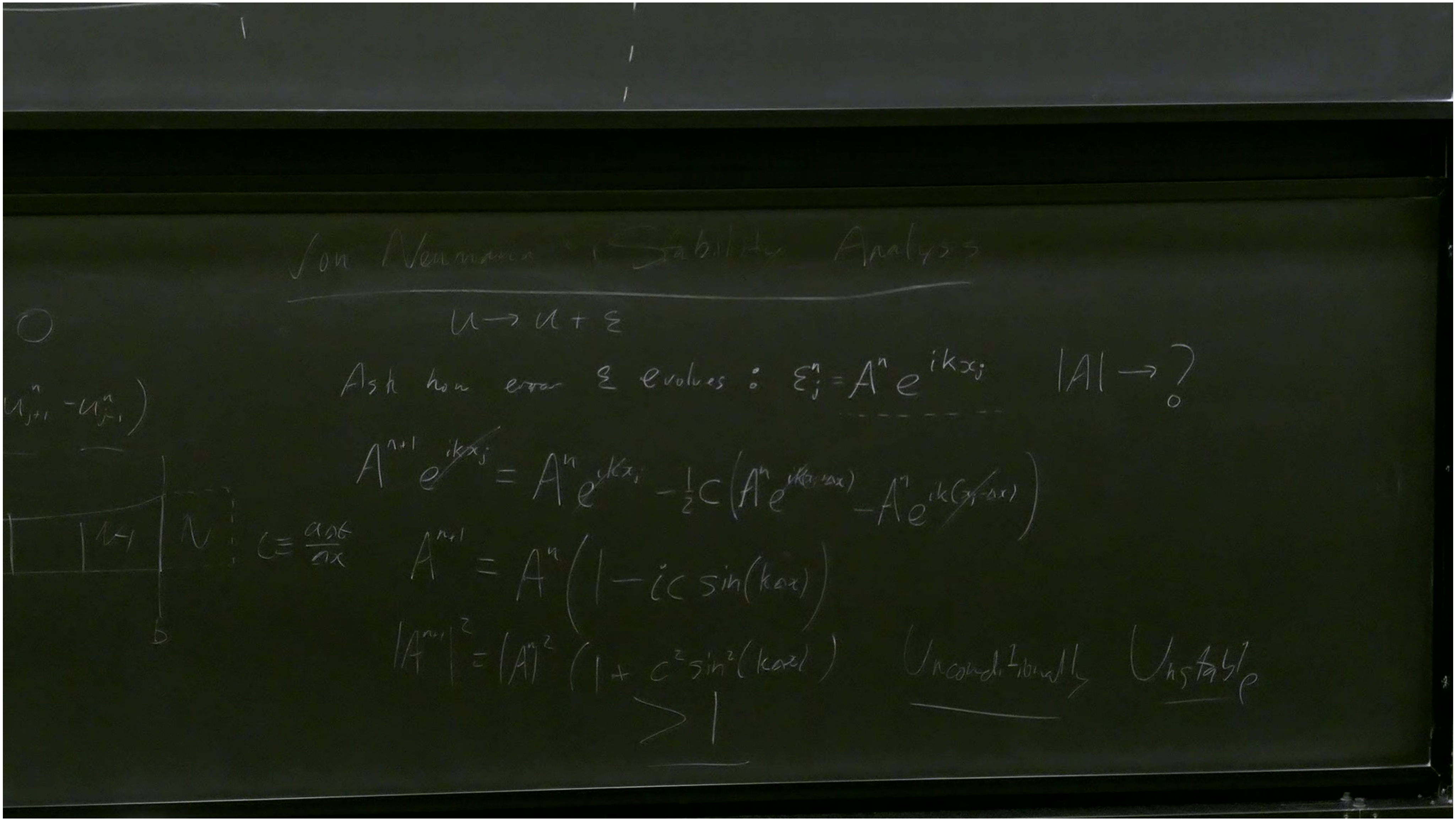
```
fig
```

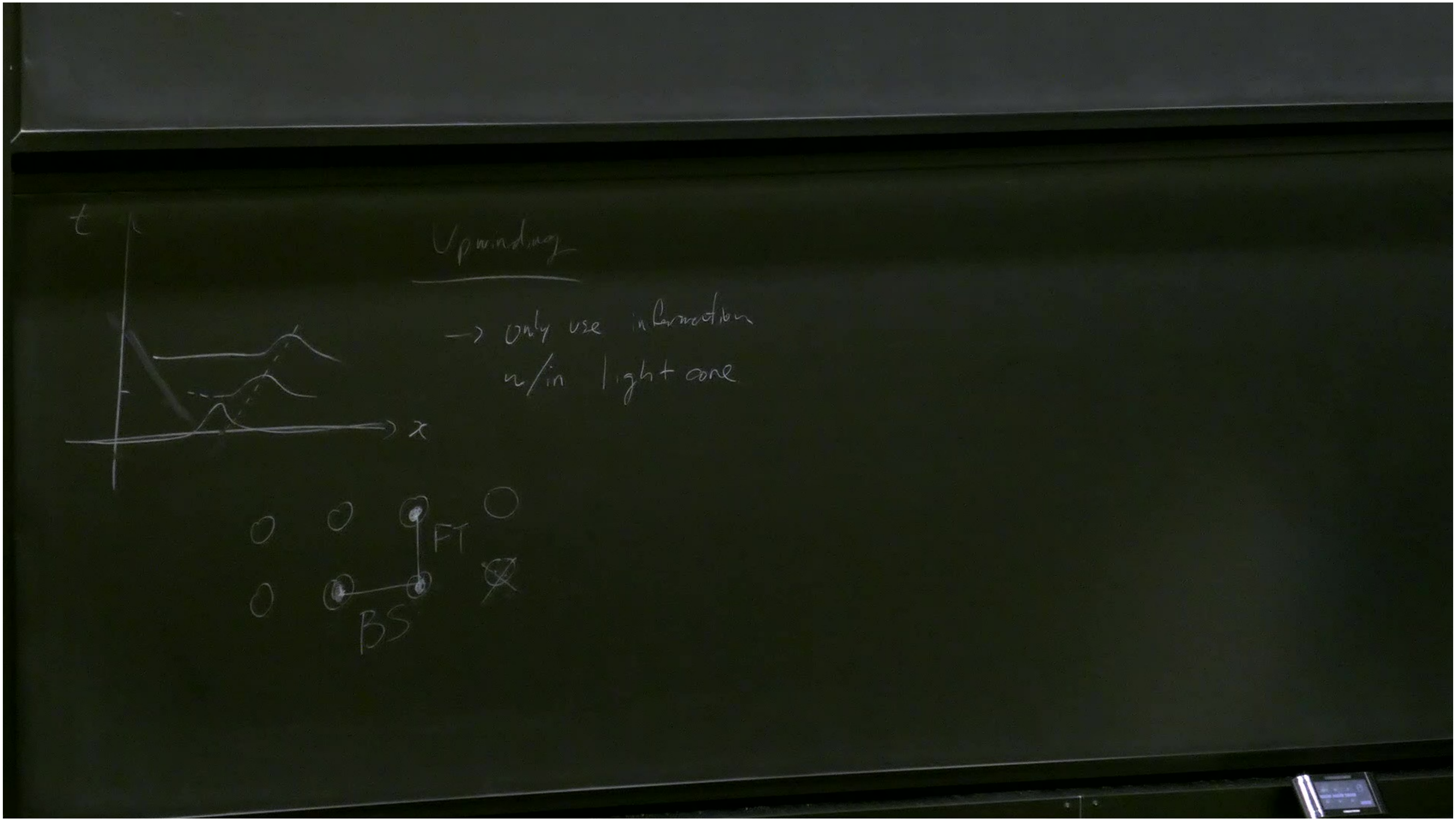
[27]:

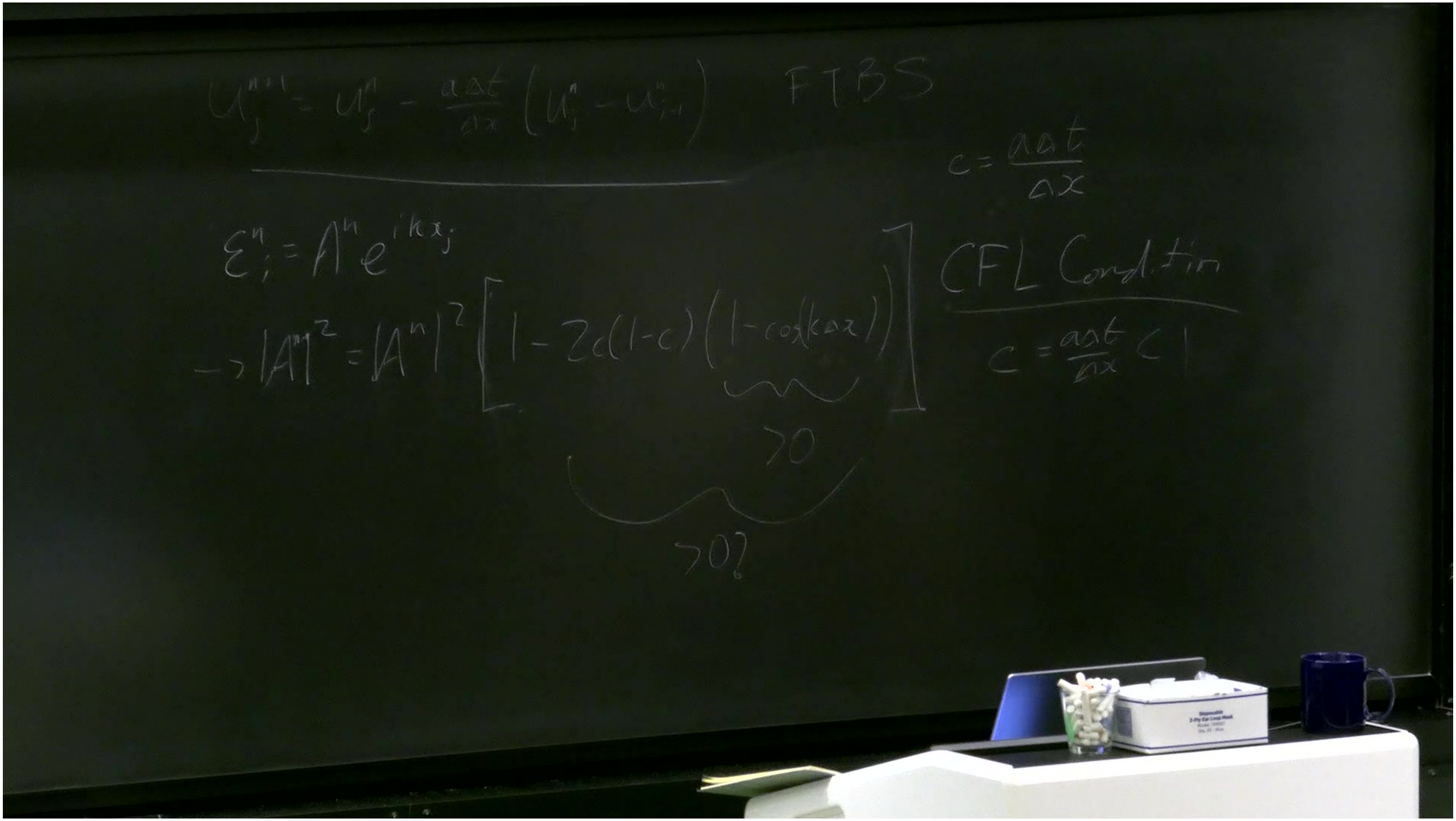



advection_howto
localhost:8888/notebooks/advection_howto.ipynb
170%
jupyter advection_howto Last Checkpoint: 1 hour ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Julia 1.11.1
[28]:
1000
500
0
−500
−1000
0.0
0.5
1.0


Von Neumann Stability Analysis
$u \to u + \varepsilon$
Ask how error ε evolves: $\varepsilon_j^n = A^n e^{ikx_j}$
$|A| \to ?$
$A^{n+1} e^{ikx_j} = A^n e^{ikx_j} - \frac{1}{2}c\left(A^n e^{ik(x_j+\Delta x)} - A^n e^{ik(x_j-\Delta x)}\right)$
$c \equiv \frac{a\Delta t}{\Delta x}$
$A^{n+1} = A^n\left(1 - ic\sin(k\Delta x)\right)$
$|A^{n+1}|^2 = |A^n|^2\left(1 + c^2\sin^2(k\Delta x)\right)$


Von Neumann Stability Analysis
u → u + ε
Ask how error ε evolves : ε_j^n = A^n e^{ikx_j} |A| → ?
A^{n+1} e^{ikx_j} = A^n e^{ikx_j} − ½c(A^n e^{ik(x_j+Δx)} − A^n e^{ik(x_j−Δx)})
c ≡ aΔt/Δx
A^{n+1} = A^n(1 − ic sin(kΔx))
|A^{n+1}|^2 = |A^n|^2 (1 + c^2 sin^2(kΔx))
> 1
Unconditionally Unstable


Upwinding
→ only use information
w/in light cone
t
x
FT
BS


$u_j^{n+1} = u_j^n - \frac{a\Delta t}{\Delta x}(u_j^n - u_{j-1}^n)$
FTBS
$c = \frac{a\Delta t}{\Delta x}$
$\varepsilon_j^n = A^n e^{ikx_j}$
$\rightarrow |A^{n+1}|^2 = |A^n|^2 \left[1 - 2c(1-c)(1-\cos(k\Delta x))\right]$
CFL Condition
$c = \frac{a\Delta t}{\Delta x} < 1$
>0
$>0?$

advection_howto

localhost:8888/notebooks/advection_howto.ipynb

jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help Trusted

Code JupyterLab Julia 1.11.1

We can *upwind* our solution by using a biased estimate for $\partial_x u$, only using u^n_{j-1} and u^n_j. This gives us the Forward-Time-Left-Space algorithm or FTLS.

FTLS is stable if $\Delta t < \Delta x/a$, the Courant-Freidrichs-Lewy (CFL) condition. We can enforce this by setting Δt with the equation:

$$\Delta t = \mathtt{CFL} \times \Delta x/a$$

Where $\mathtt{CFL} < 1.0$ is a constant.

Let's try FTLS with $CFL = 0.9$, $N_x = 100$, and $T = 0.3$

[10]:

```
dx, x, xe = make_grid(0, 1, 100, 1)

u = init_bump(x, 0.5, 0.1, 1)

fig, ax = make_figure()
add_to_plot(ax, x, u)

a = 1.0

CFL = 0.9
dt = CFL * dx / a

t_run = 0.1

# run to 0.1 & save
evolve(u, t_run, dx, dt, a, udot_ftls, periodic_BC)
```

advection_howto

localhost:8888/notebooks/advection_howto.ipynb

jupyter advection_howto Last Checkpoint: 1 hour ago

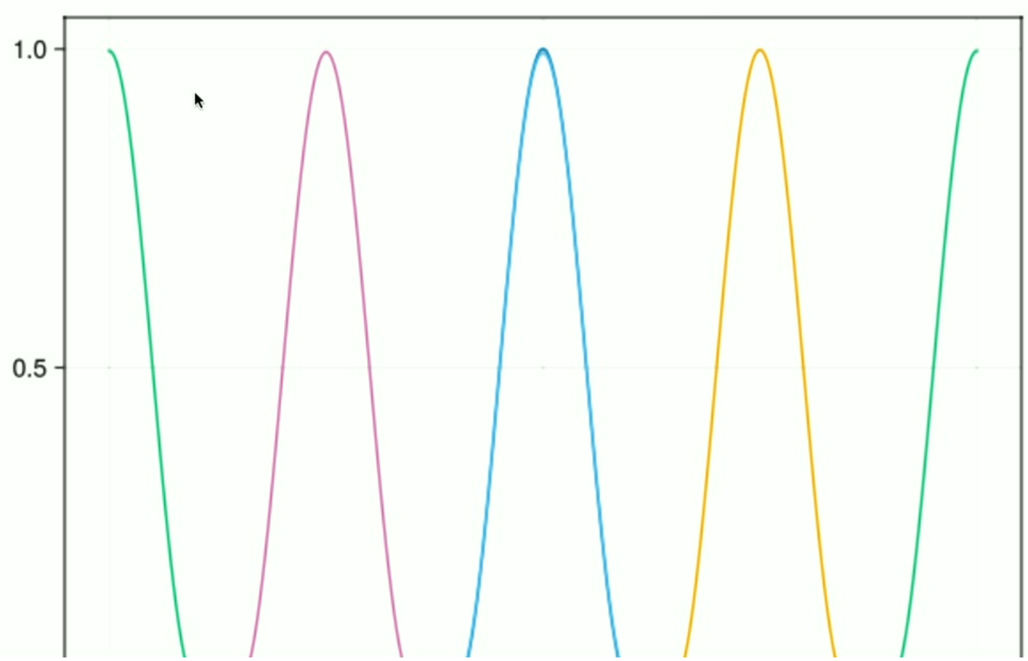
File Edit View Run Kernel Settings Help Trusted

Code JupyterLab Julia 1.11.1

```
# run for 0.25 more (1.0 total) & save
evolve(u, t_run, dx, dt, a, udot_ftls, periodic_BC)
add_to_plot(ax, x, u)

fig
```

[30]:



jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help — Trusted

Markdown — JupyterLab — Julia 1.11.1

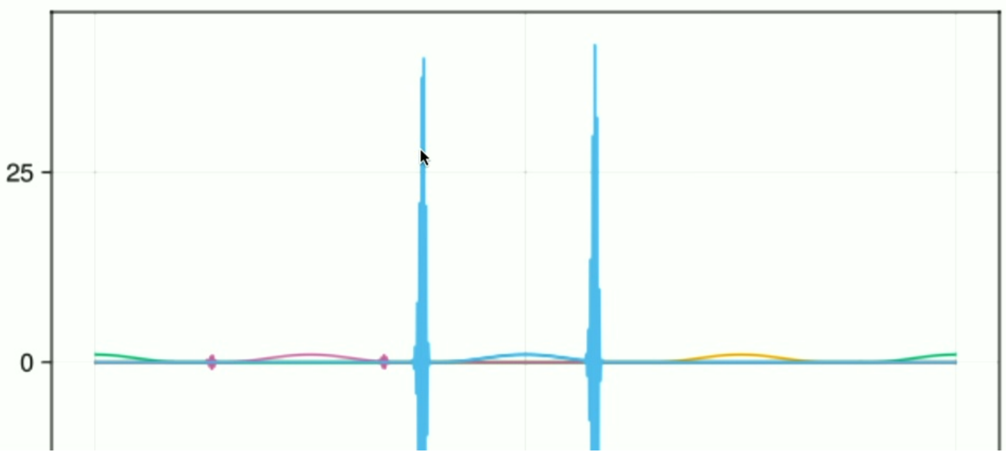
```
evolve(u, t_run, dx, dt, a, udot_ftls, periodic_BC)
add_to_plot(ax, x, u)

# run for 0.25 more (0.75 total) & save
evolve(u, t_run, dx, dt, a, udot_ftls, periodic_BC)
add_to_plot(ax, x, u)

# run for 0.25 more (1.0 total) & save
evolve(u, t_run, dx, dt, a, udot_ftls, periodic_BC)
add_to_plot(ax, x, u)

fig
```

[32]:



advection_howto

localhost:8888/notebooks/advection_howto.ipynb

jupyter advection_howto Last Checkpoint: 1 hour ago

File Edit View Run Kernel Settings Help

Trusted

Code

JupyterLab

Julia 1.11.1

```
fig
```

[34]: