

***Engineering***



***& Physics***

# **PHYS 351**

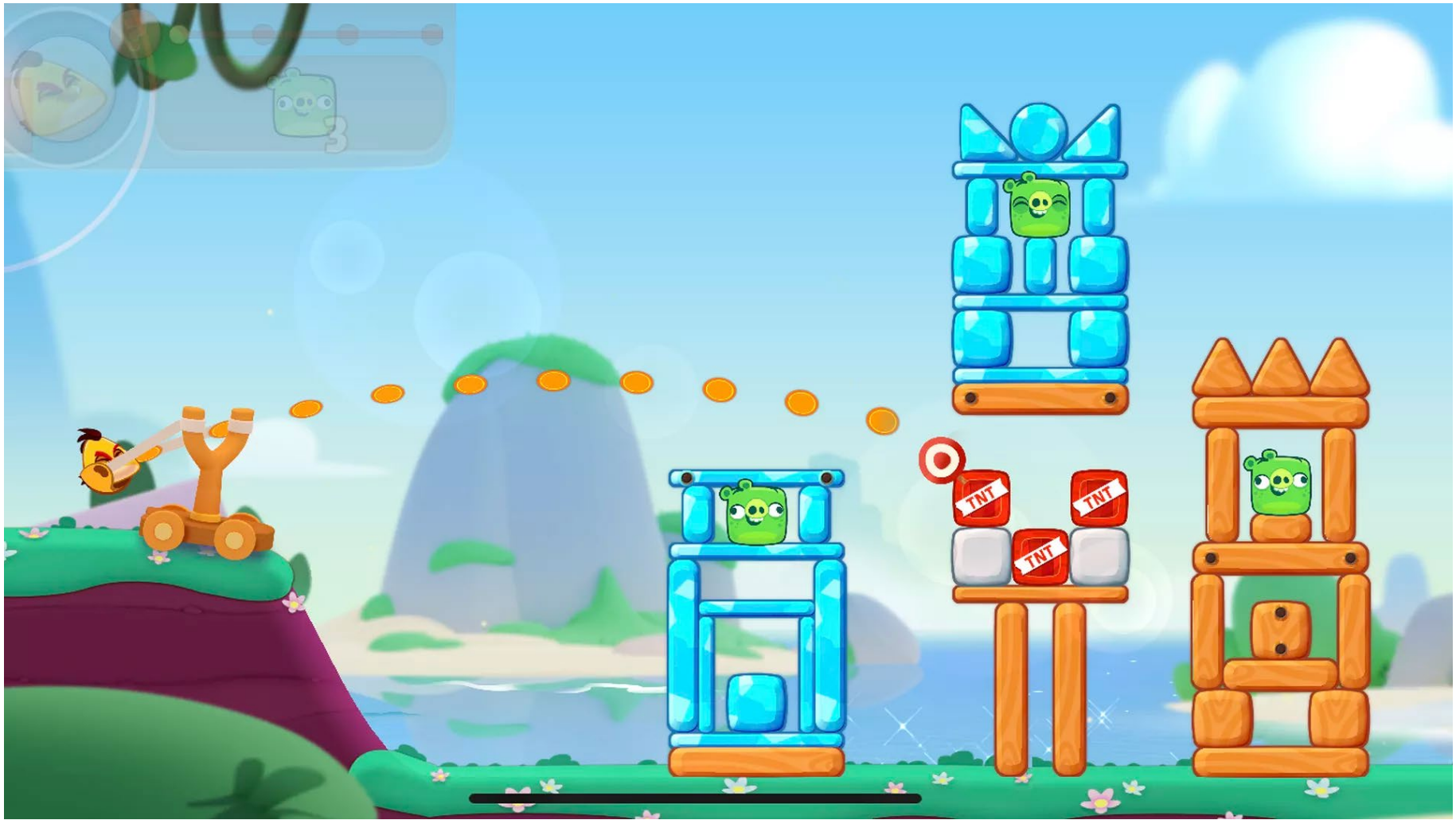
## **PDE and BVP**

Dr. Daugherty  
Abilene Christian University

# Problem Statement

**IVP:** Find an unknown function  $y(t)$  given information about its derivatives and initial values  $y(t_0), y'(t_0)$  at the same time.

**BVP:** Find an unknown function  $y(t)$  given information about its derivatives and values  $y(t_a), y(t_b)$  at different times.



BVP Practice

# SHOOTING METHOD

# Shooting Method

- We have to find an unknown parameter by starting with an initial guess and adjusting until we get the correct values at the boundaries
- Shooting for BVP = IVP + root finding!
- Continuous – we will use `solve_ivp` with its adaptive step sizes

# Shooting Method

**Example from Github: projectile motion.**

Given  $a = -g$  and boundaries  $y(0) = 0$  and  $y(2) = 10$ .

Find value of initial speed that gives us correct boundaries

```
1 # define a function to match BC
2 # want y(2)=10
3 def bc(v0):
4     y0 = [0, v0] # Initial Values
5     ivp = solve_ivp(f, [0,TMAX], y0)
6     y = ivp.y[0]
7     yf = y[-1]
8     v = ivp.y[1]
9     vf = v[-1]
10    return yf - 10
```

Define a function that takes guess for  $v_0$  and calls `solve_ivp` to return difference between actual boundary value vs BC. This function will return zero when we are right.

```
1 # root is a little less than 15. Use initial guesses x0,x1
2 sol = root_scalar(bc, x0=15, x1 = 14.9)
3 print(sol)
```

```
converged: True
flag: 'converged'
function_calls: 3
iterations: 2
root: 14.799999999999995
```

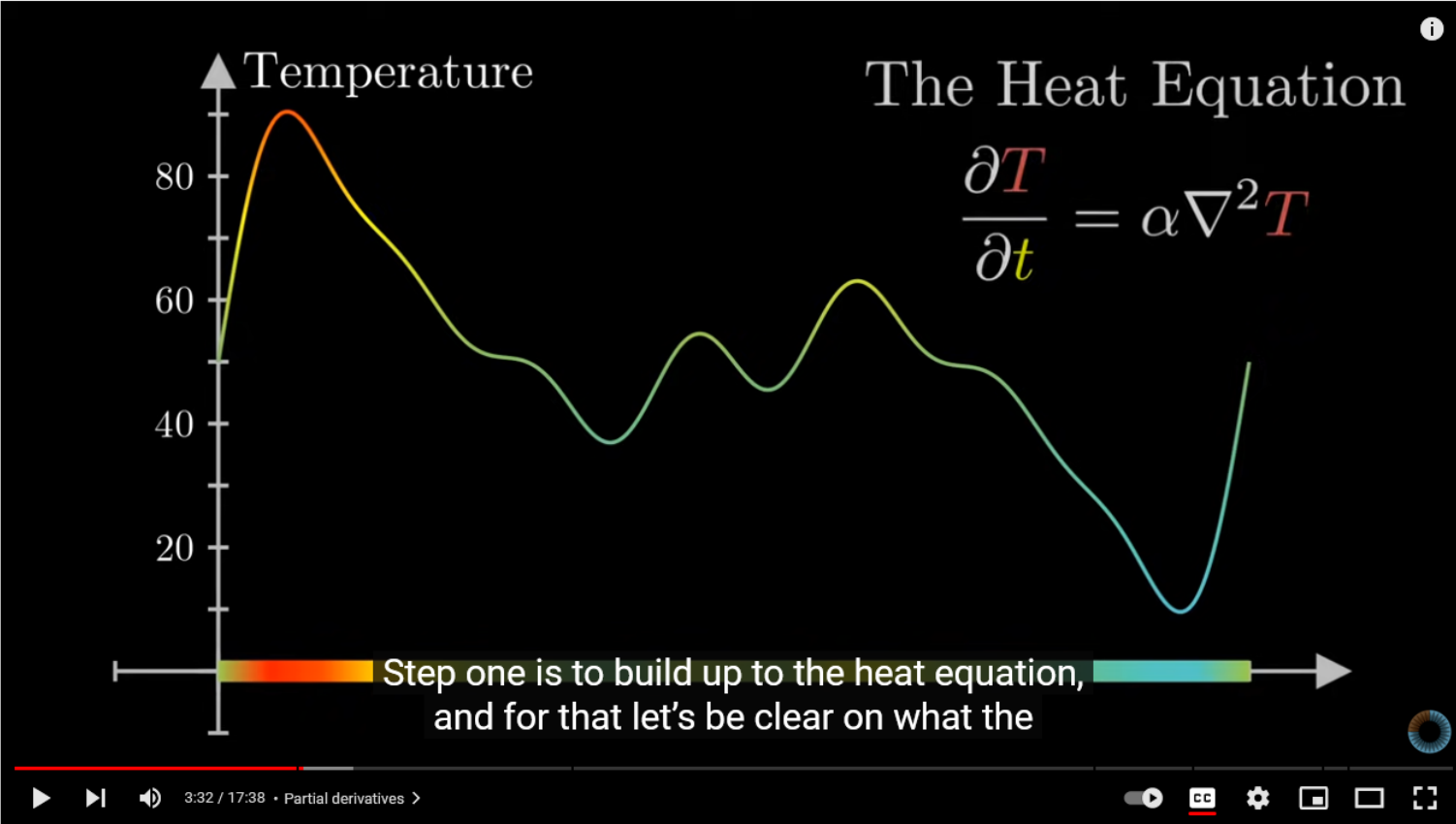
Use `root_scalar` (or `root` for vectors) to hunt for correct value of unknown parameter. Then you can run `solve_ivp` one final time with correct value.

Relax

# FINITE DIFFERENCE METHOD

<https://www.youtube.com/watch?v=ly4S0oi3Yz8>

Play 3:32 – 13:12



The video player displays a graph titled "Temperature" on the y-axis, which ranges from 0 to 80. The x-axis represents time. A curve starts at approximately (0, 50), rises to a peak of about 90, and then oscillates with decreasing amplitude, ending at approximately (10, 50). The curve is colored with a gradient from orange to green. To the right of the graph, the text "The Heat Equation" is displayed above the partial differential equation  $\frac{\partial T}{\partial t} = \alpha \nabla^2 T$ . Below the graph, a horizontal bar with a color gradient from orange to green is shown, with the text "Step one is to build up to the heat equation, and for that let's be clear on what the" overlaid on it. The video player interface includes a progress bar at the bottom, showing the video is at 3:32 / 17:38. The video title is "But what is a partial differential equation? | DE2" and the channel is "3Blue1Brown" with 4.82M subscribers. The video has 57K likes and various sharing options are visible.

Temperature

The Heat Equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

Step one is to build up to the heat equation, and for that let's be clear on what the

3Blue1Brown series S4 E2  
But what is a partial differential equation? | DE2

3Blue1Brown  
4.82M subscribers

57K

Share

Thanks

Clip

Save

Full DE Playlist:

<https://www.youtube.com/playlist?list=PLZHQObOWTQDNPOjrT6KVlfJuKtYTftqH6>

# BVP: Finite Difference

The shooting method doesn't extend well to gross problems...

## **New plan:**

1. Set up “mesh” to discretize problem -> data points instead of continuous functions. Remember error is  $O(h^2)$
2. Go back and use centered-difference derivatives (as much as possible)
3. Start with some initial condition (or guess) and enforce boundary conditions



# Implicit vs Explicit

Depending on problem we can have 2 general cases:

- **EXPLICIT:** we have enough information to directly calculate the function at a point from known values
- **IMPLICIT:** we can't find the function at a point, so we have to solve a system of equations to get the function across many points from boundary to boundary

# Relaxation

- Simplest way to solve **implicit** problems is **method of relaxation**. Many other methods exist.
- We get a system of equations from boundary to boundary that must be met simultaneously for a valid solution
- Relaxation: start with initial guess, calculate system over and over again until it stops changing
- Over relaxation: can often speed up convergence by taking larger steps

Suppose we found  $y_{NEW}$  from  $y_{OLD}$ . Instead of just using  $y_{NEW}$  we:

1) Define  $\Delta y = y_{NEW} - y_{OLD}$

2) For new values we use  $y_{OLD} + \omega \Delta y$  with  $1 \leq \omega < 2$

Simulation can get unstable if  $\omega$  is too big...

# Derivatives

**FORWARD**

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

**BACKWARD**

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

only use these at edges

**CENTERED 1st**

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

**CENTERED 2nd**

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

# Example: Free Fall

Given  $y'' = -g$  and information and initial and final times

$$y''(t) \approx \frac{y(t+h) - 2y(t) + y(t-h)}{h^2} \rightarrow y(t) \approx \frac{1}{2} \left[ \underbrace{y(t+h) + y(t-h)}_{\text{average}} - h^2 y''(t) \right]$$

Note that each point depends on average of neighboring points. If we run enough iterations we will relax into correct answer

```

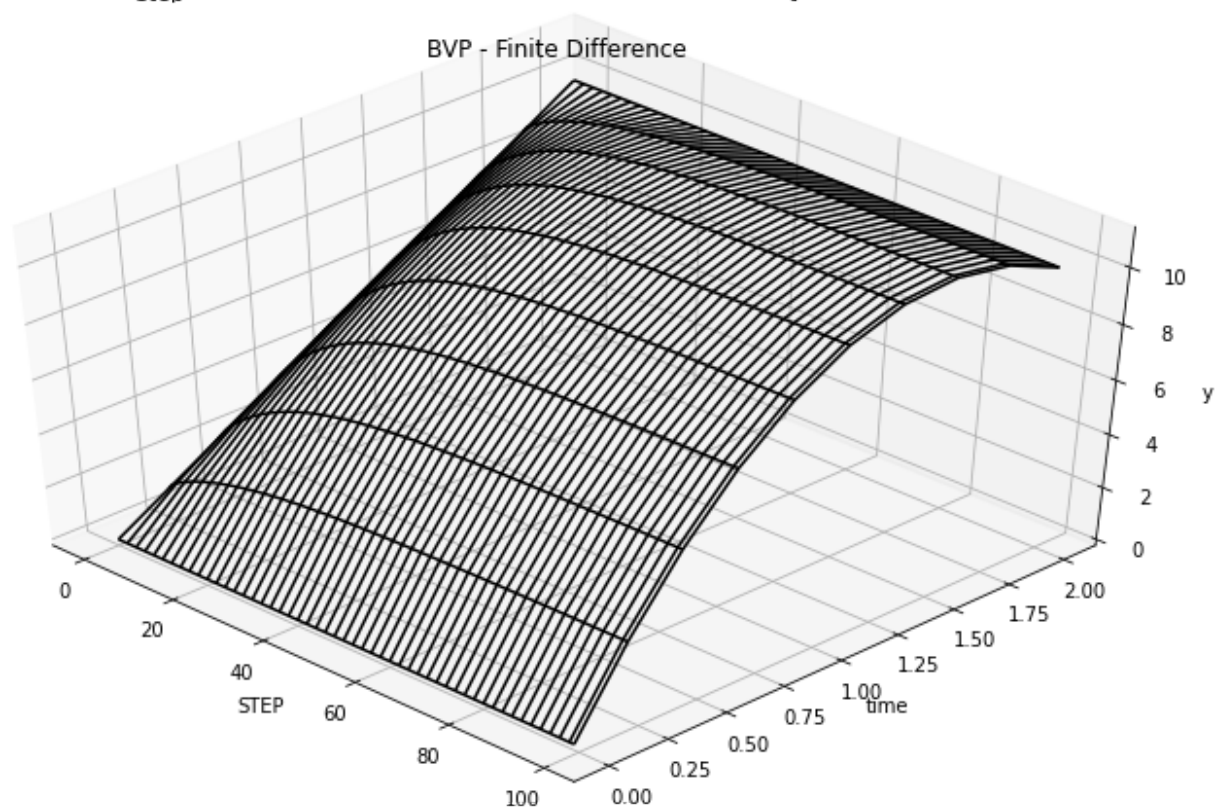
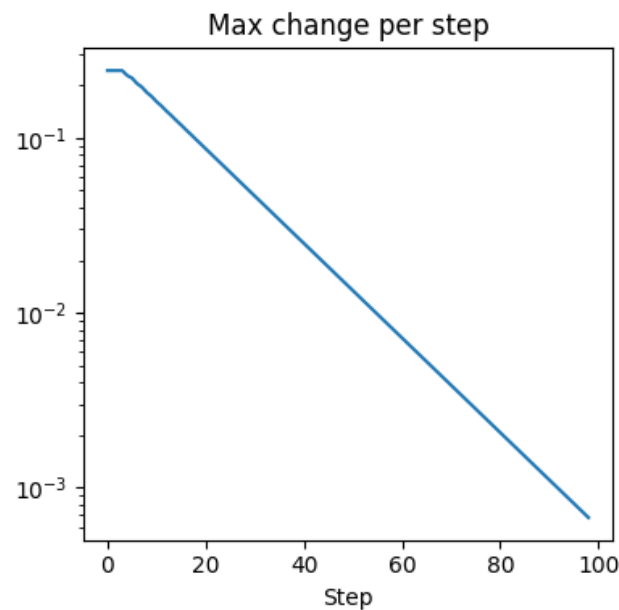
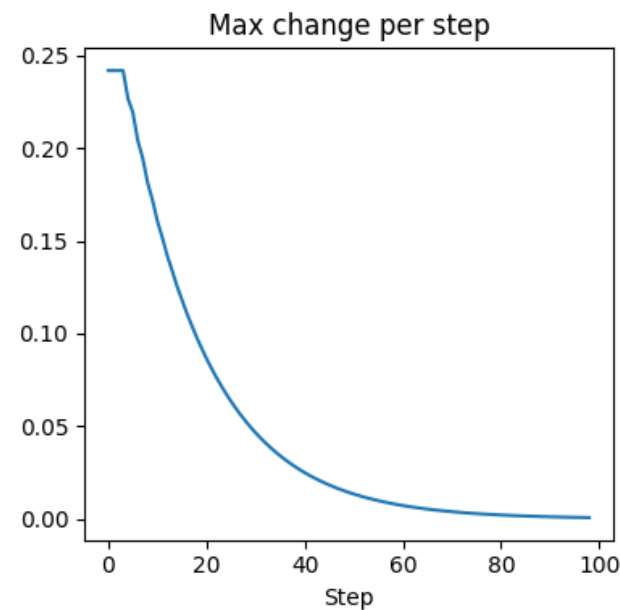
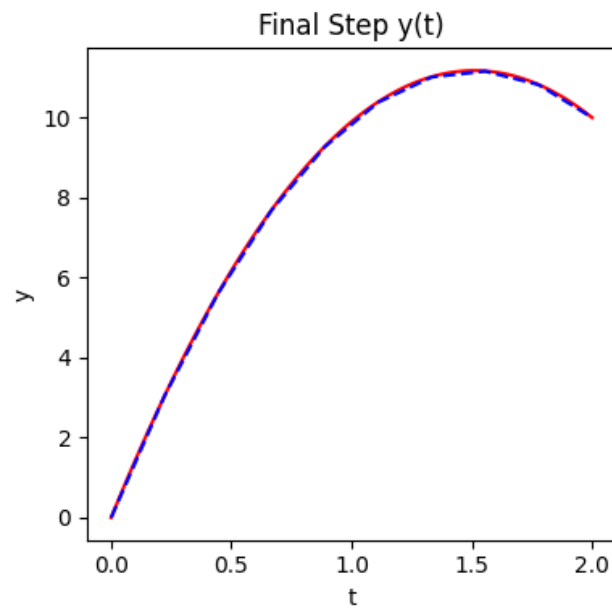
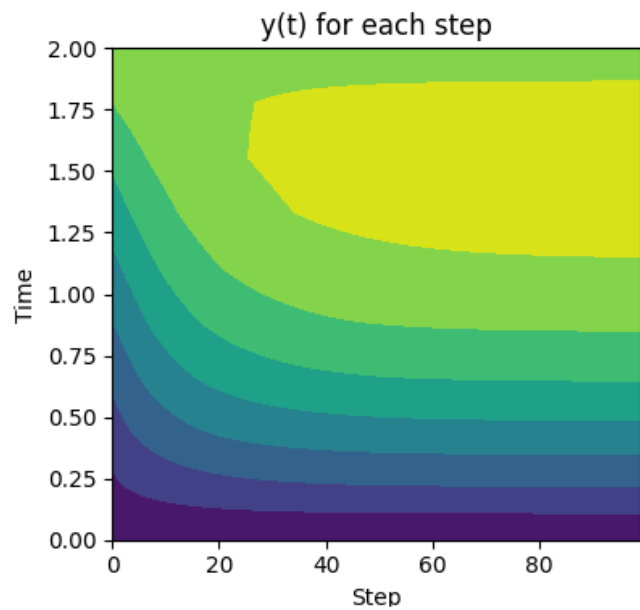
1 g = 9.8
2 a = 0 # time span
3 b = 2
4 ya = 0 # BC
5 yb = 10
6
7 t = np.linspace(a,b,10)
8 h = t[1] - t[0] # step size
9
10 y = np.zeros_like(t)
11 y[0] = ya # these don't change
12 y[-1] = yb
13
14 MAX_STEP = 100 # exit conditions
15 TOL = 1e-3
16 DONE = 0
17 STEP = 0
18 while not DONE:
19     yold = y.copy() # IMPORTANT to use copy
20
21     for i in range(1, len(y)-1):
22         y[i] = 0.5*(y[i-1]+y[i+1]+h**2*g)
23
24     #plt.plot(t,y,'-')
25     r = y-yold
26     rmax = np.abs(r).max() # largest element of r
27     #print(STEP, rmax)
28     if rmax<TOL:
29         DONE = 1
30
31     STEP+=1
32     if STEP>=MAX_STEP:
33         DONE = 1
34 print('NUM STEPS =',STEP)
35 print('TOL = ',rmax)

```

Technical note on line 22: you would expect that the right-hand side of this equation to use yold instead of y. As we move across from left to right the code as written will be using the new value of y on the left and the old value on the right. It turns out that writing it this way converges faster.

$$\rightarrow y(t) \approx \frac{1}{2} [ y(t+h) + y(t-h) - h^2 y''(t) ]$$

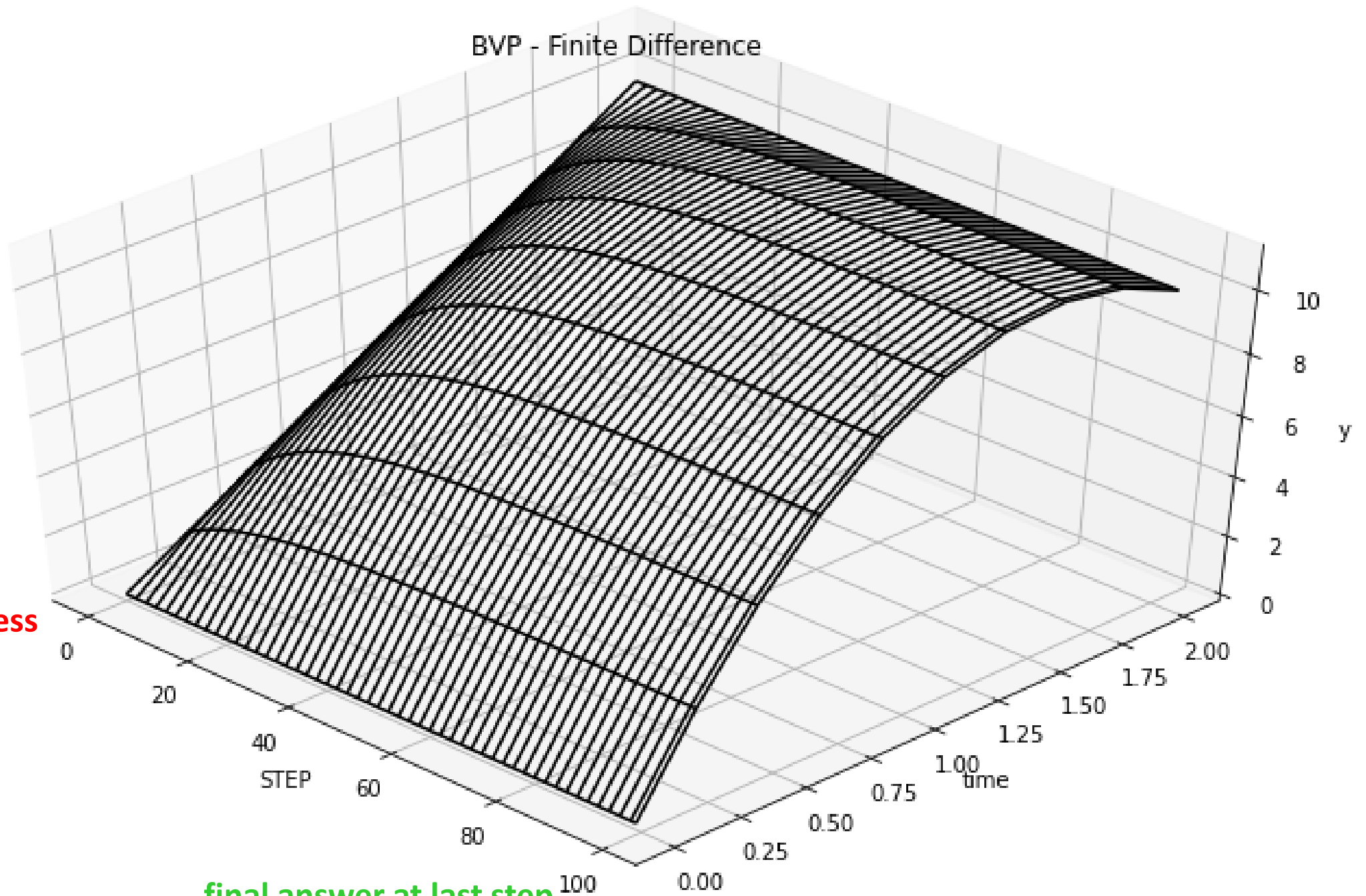
Convergence: run until the change between iterations is small



BVP - Finite Difference

initial guess  
at step 0

final answer at last step



Finally time for PDE

# **PARTIAL DIFFERENTIAL EQUATIONS**



# Problem Statement

**ODE:** unknown function  $y(t)$  depends only on one variable

**PDE:** unknown function  $y(x, t)$  depends on more than one variable

# Partial Differential Equations

Same plan with Finite Difference Method as before, but now derivatives will be different (e.g. some for space and others for time)

- Write PDE in discrete form using derivative equations
- Solve equation for next time step in terms of current time
- Enforce boundary conditions at each step (either by fixing endpoints or calculating endpoints to get correct derivatives)

# Heat Flow

For temperature  $T(x, t)$  we get:

$$\frac{\partial T}{\partial t} = \frac{K}{C\rho} \nabla^2 T$$

Constants:

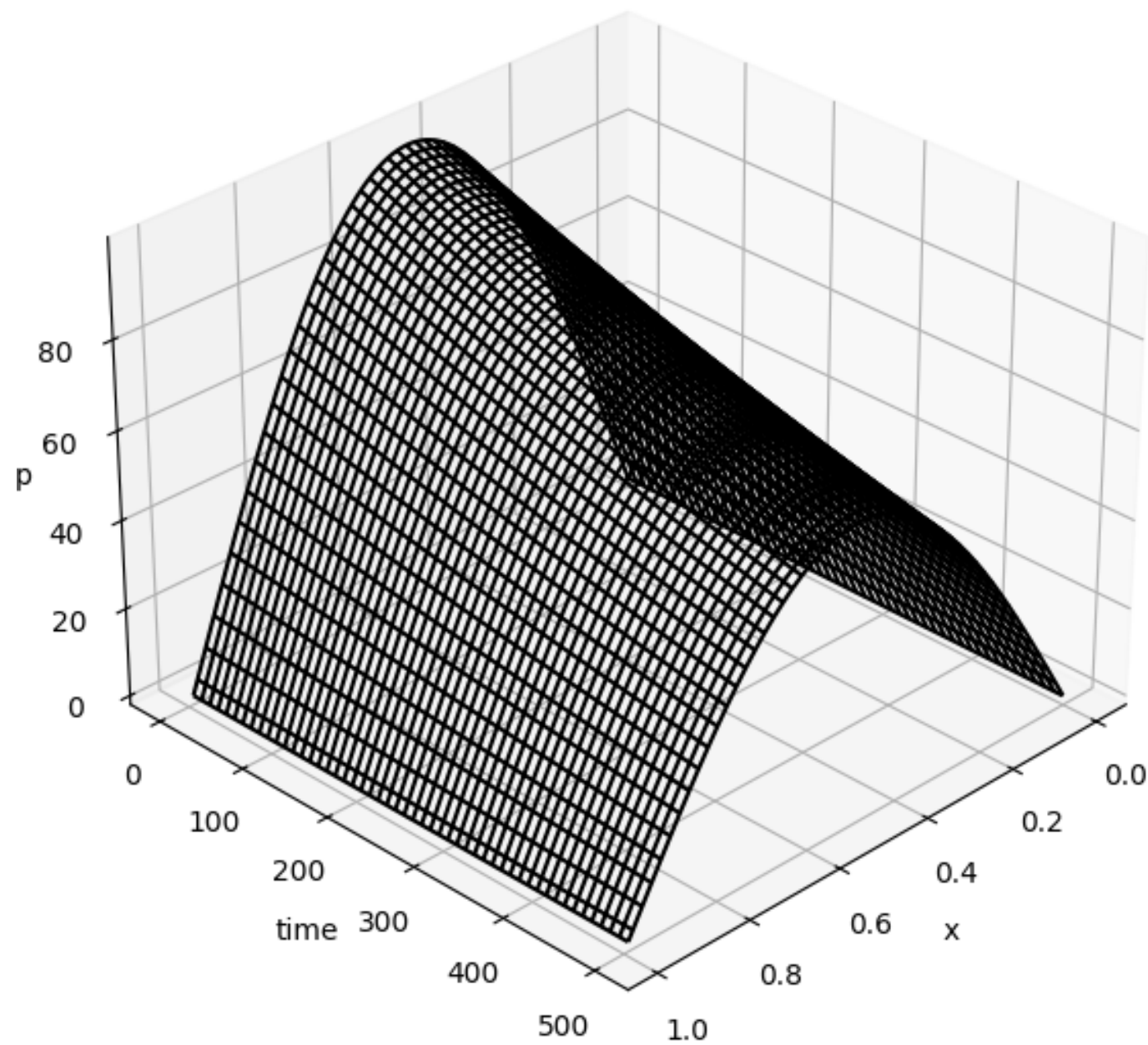
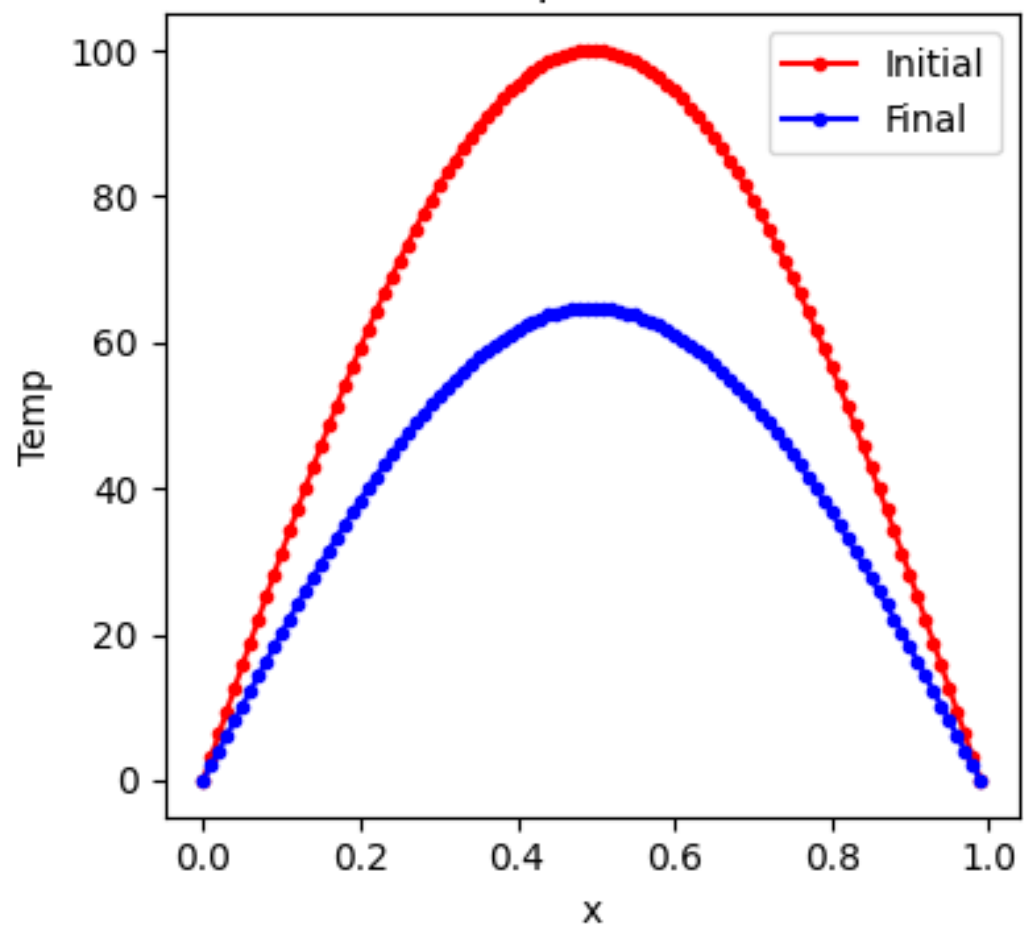
$K$  = thermal conductivity (gives rate of heat flow)

$C$  = specific heat (relates heat to temperature per unit mass)

$\rho$  = density (mass per unit length)

No relaxation, just simulating forwards in time.

Temp Profiles



# Electrostatic Potentials

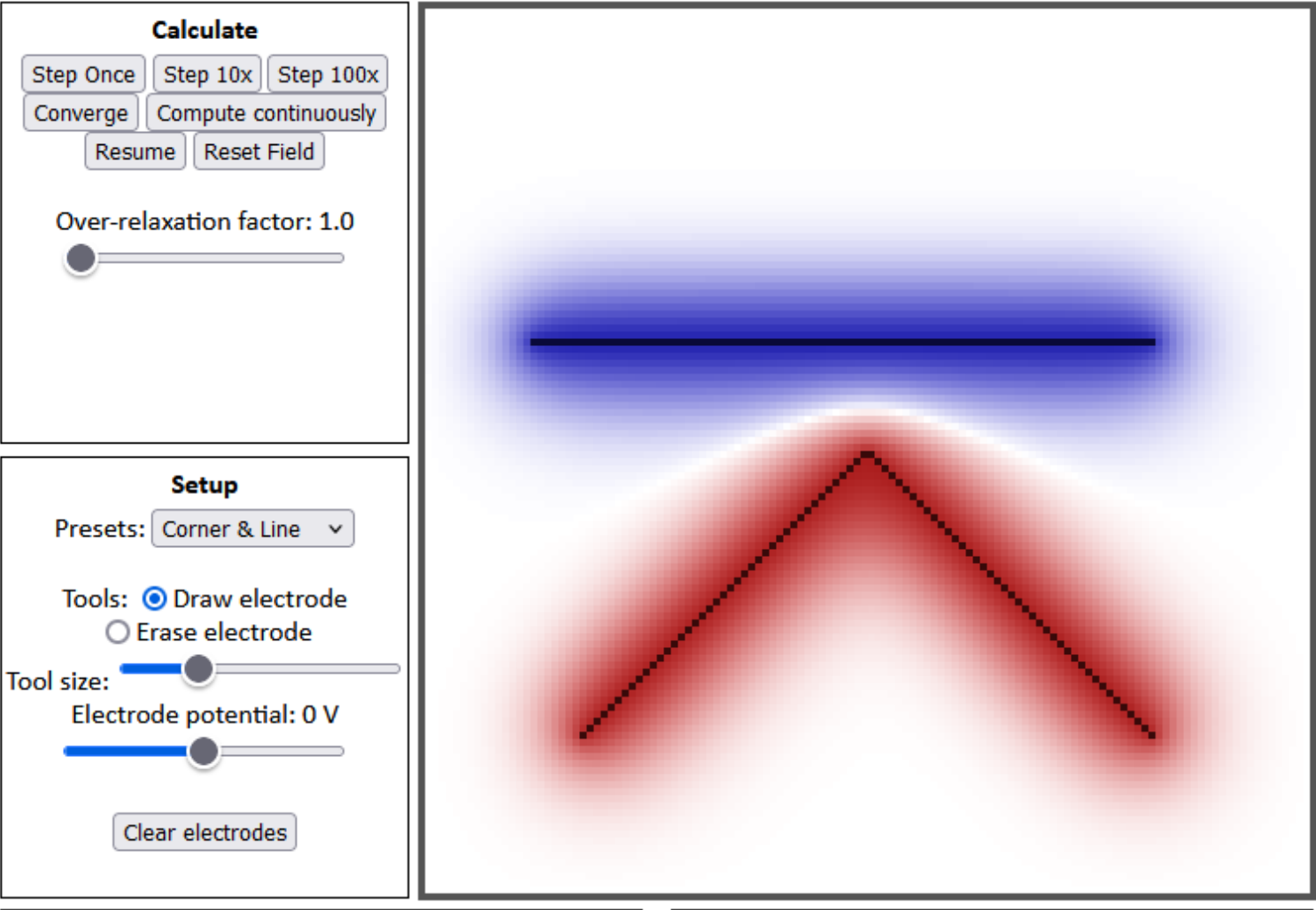
Super simple example of PDE. In 2D free space:

$$\nabla^2 V = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

Follow same plan:

- define a mesh
- write centered-difference for both derivatives
- solve for  $V_{i,j}$  in terms of other points
- use method of relaxation

# Electric Potentials and Fields



## From page source:

```
//uses the Gauss-Seidel relaxation method
function calculate() {
    convergence = 0; //reset the degree of convergence in each calculation step

    for (var r = 0; r < numberOfCells; r++) {
        for (var c = 0; c < numberOfCells; c++) {
            if (!electrodes[r][c]) {
                var vOld = vCells[r][c],
                    change = overRelaxationFactor*((vCells[r][c + 1] + vCells[r - 1][c] + vCells[r][c - 1] + vCells[r + 1][c])/4 - vOld);
                //the potential at any cell is the average of the potentials at each neighboring cell
                vCells[r][c] += change;
                if (change > convergence) { //the largest change in the calculation step is the degree of convergence
                    convergence = change;
                }
            }
        }
    }
}
```

BVP Practice

**SOLVE\_BVP**



# scipy.integrate.solve\_ivp

`scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **options)` [\[source\]](#)

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

$$\begin{aligned} \frac{dy}{dt} &= f(t, y) \\ y(t_0) &= y_0 \end{aligned}$$

Here  $t$  is a 1-D independent variable (time),  $y(t)$  is an N-D vector-valued function (state), and an N-D vector-valued function  $f(t, y)$  determines the differential equations. The goal is to find  $y(t)$  approximately satisfying the differential equations, given an initial value  $y(t_0)=y_0$ .

Some of the solvers support integration in the complex domain, but note that for stiff ODE solvers, the right-hand side must be complex-differentiable (satisfy Cauchy-Riemann equations [\[11\]](#)). To solve a problem in the complex domain, pass  $y_0$  with a complex data type. Another option always available is to rewrite your problem for real and imaginary parts separately.

**Parameters:** **fun** : *callable*

Right-hand side of the system. The calling signature is `fun(t, y)`. Here  $t$  is a scalar, and there are two options for the ndarray  $y$ : It can either have shape  $(n,)$ ; then  $fun$  must return array\_like with shape  $(n,)$ . Alternatively, it can have shape  $(n, k)$ ; then  $fun$  must return an array\_like with shape  $(n, k)$ , i.e., each column corresponds to a single column in  $y$ . The choice between the two options is determined by *vectorized* argument (see below). The vectorized implementation allows a faster approximation of the Jacobian by finite differences (required for stiff solvers).

# scipy.integrate.solve\_bvp

`scipy.integrate.solve_bvp(fun, bc, x, y, p=None, S=None, fun_jac=None, bc_jac=None, tol=0.001, max_nodes=1000, verbose=0, bc_tol=None)` [\[source\]](#)

Solve a boundary value problem for a system of ODEs.

This function numerically solves a first order system of ODEs subject to two-point boundary conditions:

$$\begin{aligned} \frac{dy}{dx} &= f(x, y, p) + S * y / (x - a), \quad a \leq x \leq b \\ bc(y(a), y(b), p) &= 0 \end{aligned}$$

Here  $x$  is a 1-D independent variable,  $y(x)$  is an N-D vector-valued function and  $p$  is a k-D vector of unknown parameters which is to be found along with  $y(x)$ . For the problem to be determined, there must be  $n + k$  boundary conditions, i.e.,  $bc$  must be an  $(n + k)$ -D function.

**Parameters:** **fun** : *callable*

Right-hand side of the system. The calling signature is `fun(x, y)`, or `fun(x, y, p)` if parameters are present. All arguments are ndarray:  $x$  with shape  $(m,)$ ,  $y$  with shape  $(n, m)$ , meaning that  $y[:, i]$  corresponds to  $x[i]$ , and  $p$  with shape  $(k,)$ . The return value must be an array with shape  $(n, m)$  and with the same layout as  $y$ .

**bc** : *callable*

Function evaluating residuals of the boundary conditions. The calling signature is `bc(ya, yb)`, or `bc(ya, yb, p)` if parameters are present. All arguments are ndarray:  $ya$  and  $yb$  with shape  $(n,)$ , and  $p$  with shape  $(k,)$ . The return value must be an array with shape  $(n + k,)$ .

**x** : *array\_like, shape (m,)*

Initial mesh. Must be a strictly increasing sequence of real numbers with  $x[0]=a$  and  $x[-1]=b$ .

**y** : *array\_like, shape (n, m)*

Initial guess for the function values at the mesh nodes,  $i$ th column corresponds to  $x[i]$ . For problems in a complex domain pass  $y$  with a complex data type (even if the initial guess is purely real).

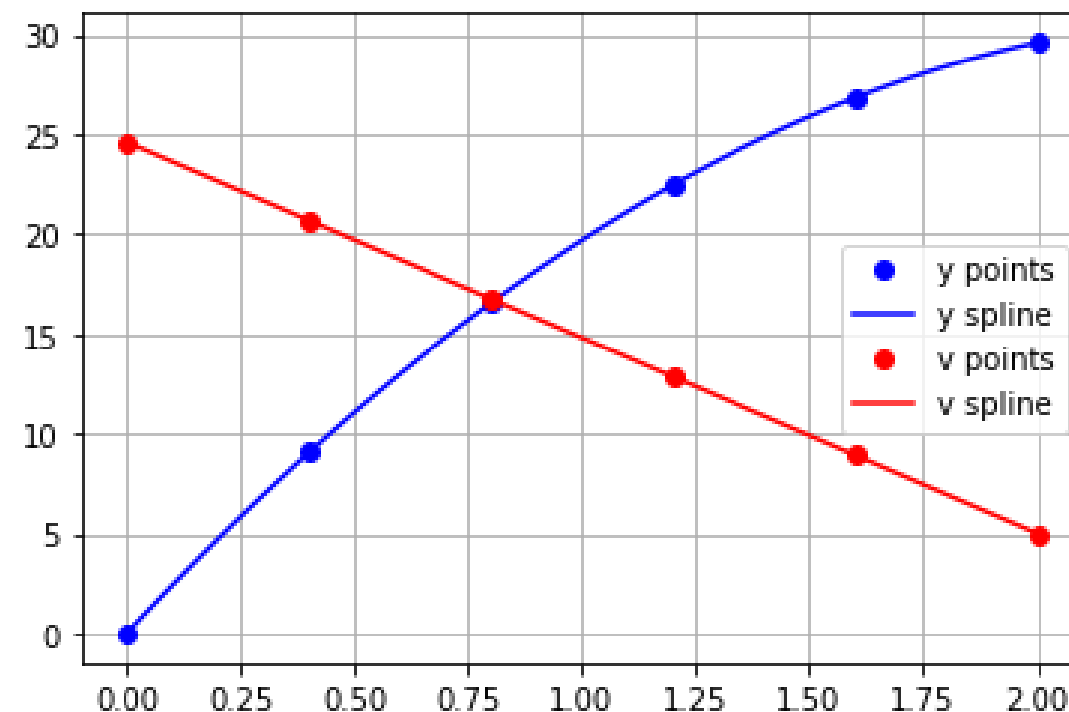
**p** : *array\_like with shape (k,) or None, optional*

Initial guess for the unknown parameters. If `None` (default), it is assumed that the problem doesn't depend on any parameters.

```
1 # Definitions:
2 # x = 1D array with shape (m)
3 # y = 2D array with shape (n,m) for n equations
4
5 # Define y=[y0, y1] so n=2 with y0=y(t), y1=y0'
6 def fun(x,y): # derivative function, returns y' with shape (n,m)
7     return np.vstack((y[1], -9.8*np.ones_like(x)))
8
9 def bc(ya,yb): # BC residuals for y at both endpoints, returns shape (n)
10     # BC: y[0]=0, y'[2]=5 (ya is y at t=0, yb is y at tf)
11     return np.array([ya[0], yb[1]-5])
12
13 # Initial values
14 TMAX = 2
15 x = np.linspace(0,2,6)
16 y0 = np.zeros((2,x.size)) # initial guess: y is shape (n,m)
17
18 sol = solve_bvp(fun, bc, x, y0)
19 print(sol.message)
20
21 # Look at points
22 t = sol.x
23 y = sol.y[0]
24 v = sol.y[1]
25
26 # Spline Interpolation
27 ts = np.linspace(0,TMAX, 50) # points to interpolate
28 spl = sol.sol(ts) # splines with shape (n,m)
29 ys = spl[0]
30 vs = spl[1]
31
32 plt.plot(t,y,'bo', label='y points')
33 plt.plot(ts,ys,'b-', label='y spline')
34 plt.plot(t,v,'ro',label='v points')
35 plt.plot(ts,vs,'r-',label='v spline')
36 plt.legend()
37 plt.grid()
38 plt.show()
```

## Free fall with $y(0)=0$ , $y'(2)=5$

The algorithm converged to the desired accuracy.



# Practice Problems

Solve  $y'' = -3y y'$  with  $y(0) = 0$  and  $y(2) = 1$

Kiusalaas pg 311

# Things to Know

- Terminology: ODE vs PDE, IVP vs BVP
- Shooting method = more accurate, best for easy BVPs
- Finite Difference method = best for PDEs