Computational practicum: Lecture 7 Boundary value problems (BVPs): Shooting method, solve_bvp function

Zoë J.G. Gromotka, Artur M. Schweidtmann, Ferdinand Grozema, Tanuj Karia

With support from Lukas S. Balhorn and Monica I. Lacatus

Computational Practicum
Dept. Chemical Engineering
Delft University of Technology

TU Delft
Process Intelligence Research
Delft Institute of Applied Mathematics

Wednesday, 15th October 2024

Recap last lecture

Recap

Boundary value problems (BVPs)

- BVPs have side constraints at more than one point.
- Boundary conditions define side constraints, e.g.,
  - Dirichlet boundary condition $y = f$
  - Neumann boundary condition $\frac{\mathrm{d}y}{dx} = f$
  - …

Finite difference method

- Equation, e.g.,

$$\frac{d^2y}{dx^2} + \alpha\frac{dy}{dx} + y\beta = f(x)$$

- Finite difference equations for all nodes:

$$y_{i+1}\left(1 + \frac{h}{2}\alpha\right) + y_i(h^2\beta - 2) + y_{i-1}\left(1 - \frac{h}{2}\alpha\right) = h^2 f(x_i)$$

$$\forall\, i \in [1, N-1]$$

$$y_0 = y_a, \qquad y_N = y_b$$

Learning goals of this lecture

After successfully completing this lecture, you are able to…

- explain numerical solution methods for boundary value problems (BVPs), namely:
  - finite difference method,
  - shooting method.
- implement the shooting method for boundary value problems (BVPs) from scratch.
- use Python libraries' built-in functions for numerical solution approaches to BVPs.

Agenda

- **Boundary value problems (BVPs)**
  - Shooting method
    a. Linear interpolation
    b. Root-finding with secant method
  - *Solve_bvp* function

Agenda

- **Boundary value problems (BVPs)**
  - Shooting method
    a. Linear interpolation
    b. Root-finding with secant method
  - *Solve_bvp* function

An example boundary value problem (BVP)

- ODE

$$\frac{d^2y}{dx^2} = 4(y - x)$$

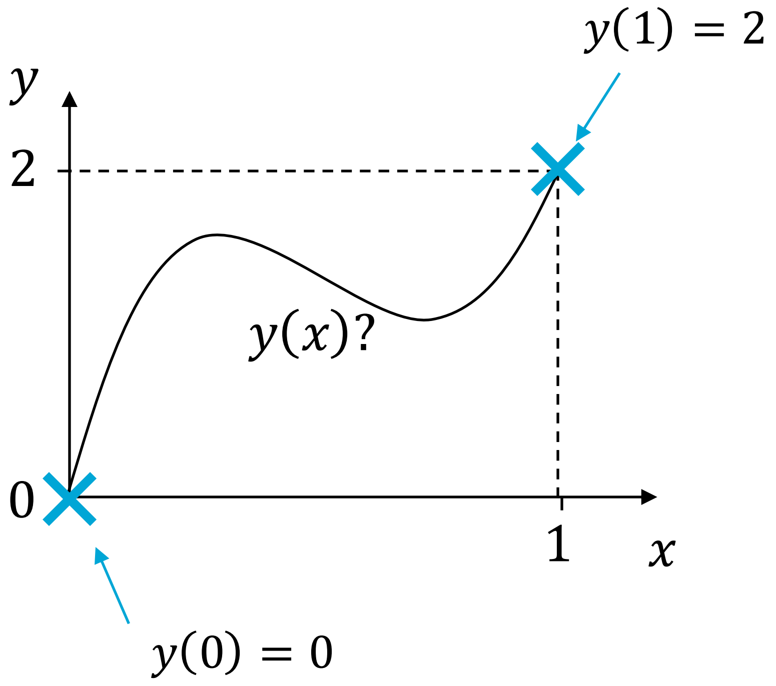
- Domain

$$x \in [0,1]$$

- Boundary conditions

$$y(0) = 0, \qquad y(1) = 2$$

- Let's try to interpret the BVP as an initial value problem (IVP)!
  - The ODE is not 1st order!
  - The ODE is non-autonomous but for the methods presented here, this is not a problem.



RECAP

To solve an IVP, the ODE should be in a 1st order form, cf. lecture 5.

Rewrite BVP in 1st order

- ODE

$$\frac{d^2y}{dx^2} = 4(y - x)$$

- Variable substitution

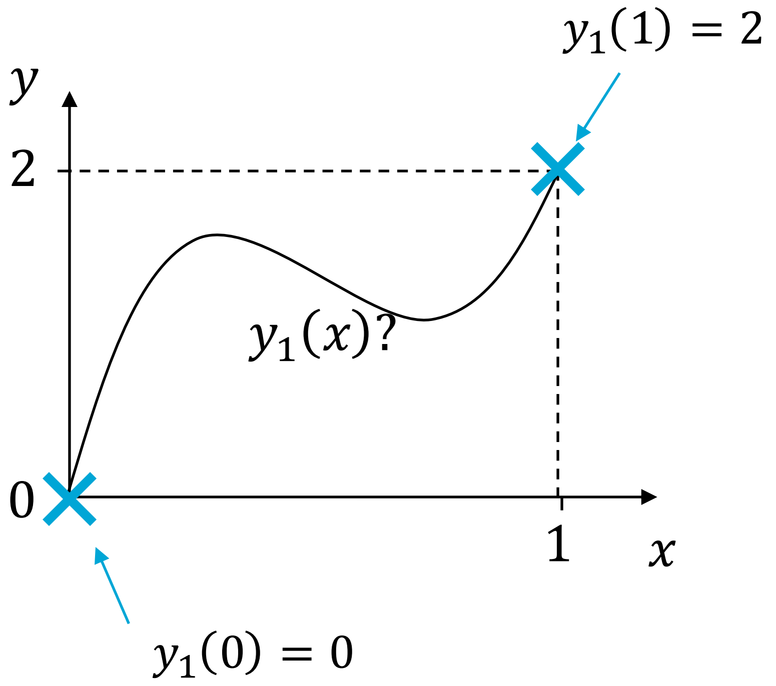
$$y_1 \equiv y, \qquad y_2 \equiv \frac{dy_1}{dx}$$

- 1st order ODE system

$$\frac{dy_1}{dx} = y_2, \qquad \frac{dy_2}{dx} = 4(y_1 - x)$$

- Initial conditions?

$$y_1(0) = 0, \qquad y_2(0) = ?$$



There is no initial condition defined for y_2. We need the initial condition to apply the IVP methods.

The shooting method

- 1st order ODE system

$$\frac{dy_1}{dx} = y_2, \qquad \frac{dy_2}{dx} = 4(y_1 - x)$$

- Boundary conditions

$$y_1(0) = 0, \qquad y_1(1) = 2$$

For now, ignore when solving the IVP

1. Make a (random) guess for the initial condition of y_2.

$$y_2(0) = \gamma_0 = 1$$

2. Solve the ODE system with the IVP methods from lecture 5, e.g., forward Euler method.
3. Evaluate the solution and update the guess γ for $y_2(0)$ until $y_1(1; \gamma) = 2$.

HINT

For an IVP, all side constraints ("initial conditions") need to be defined in one point. This point does not necessarily need to be the left domain boundary.

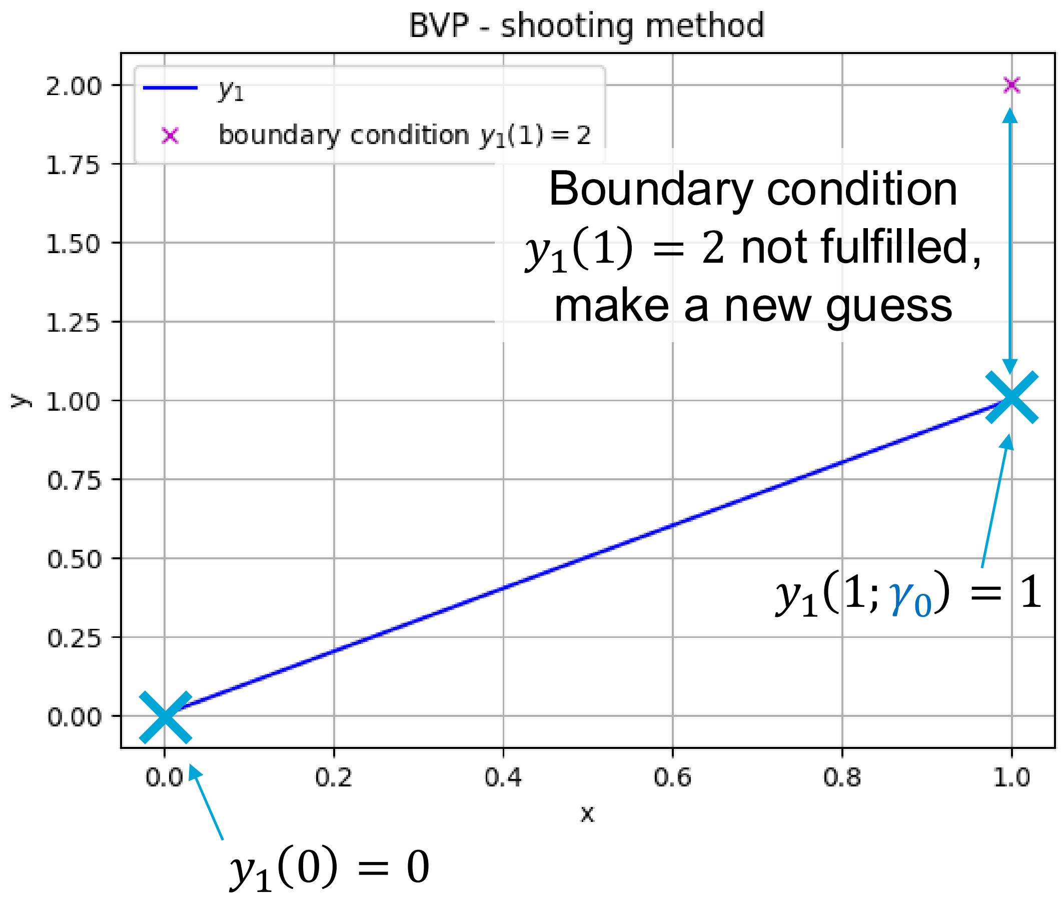
Greek letter gamma γ, not to be confused with Latin letter y.

The shooting method

$$\frac{dy_1}{dx} = y_2, \qquad \frac{dy_2}{dx} = 4(y_1 - x), \qquad y_1(0) = 0$$

- First guess γ_0

$$y_2(0) = \gamma_0 = 1$$



The shooting method

$$\frac{dy_1}{dx} = y_2, \qquad \frac{dy_2}{dx} = 4(y_1 - x), \qquad y_1(0) = 0$$

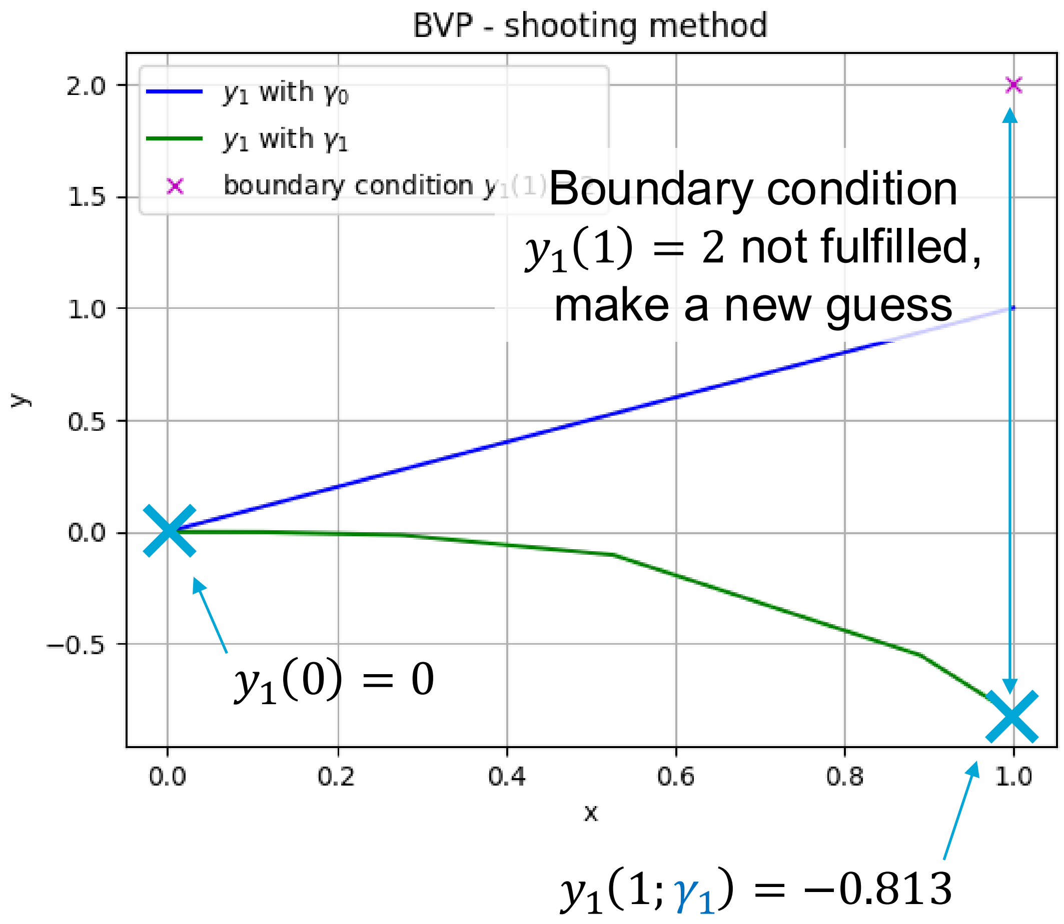
- First guess γ_0

$$y_2(0) = \gamma_0 = 1$$

- Second guess γ_1

$$y_2(0) = \gamma_1 = 0$$

We "shoot" from the start of the domain toward target. Then we repeat the shooting until we hit the target. Therefore, the method is called shooting method. :D



The shooting method

$$\frac{dy_1}{dx} = y_2, \qquad \frac{dy_2}{dx} = 4(y_1 - x), \qquad y_1(0) = 0$$

- First guess γ_0

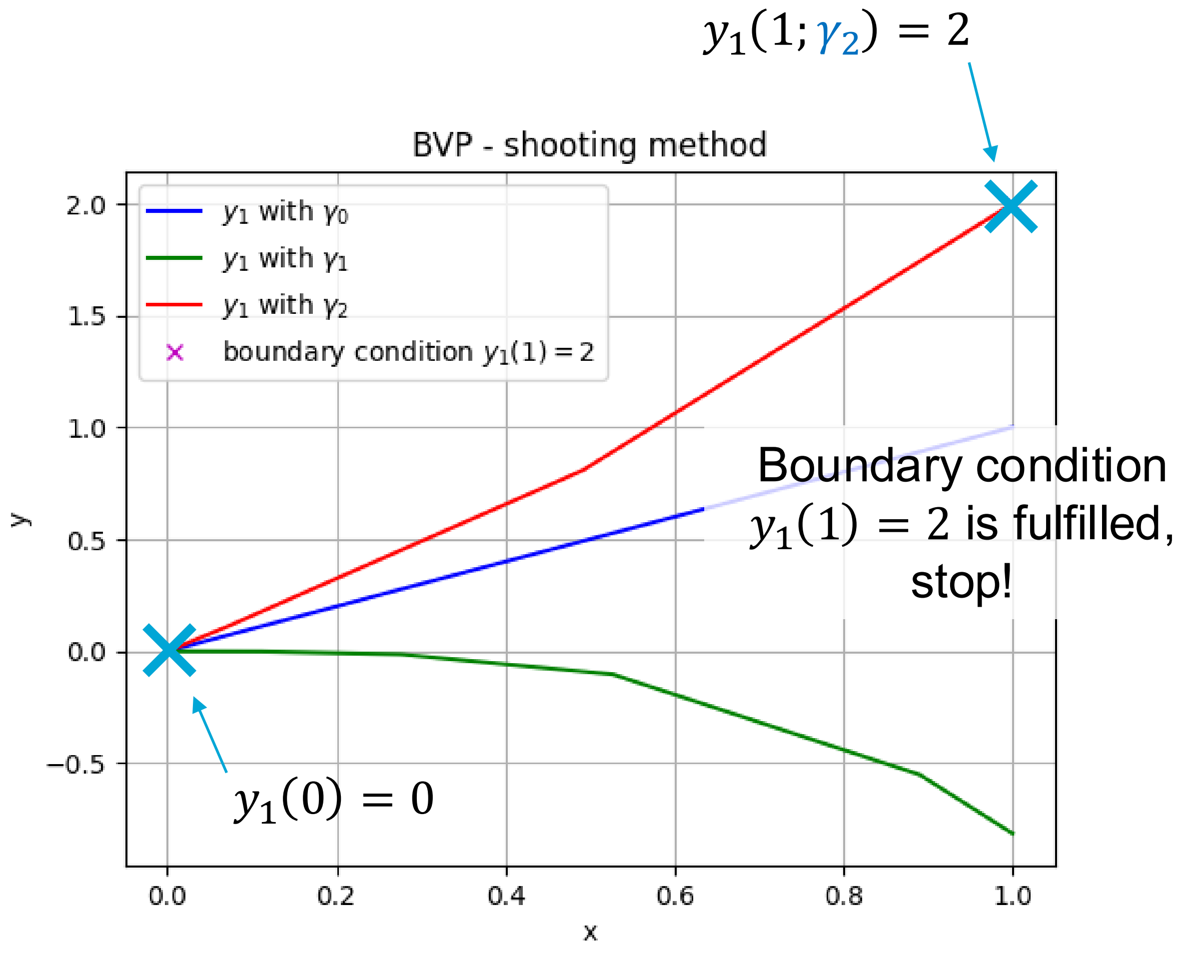
$$y_2(0) = \gamma_0 = 1$$

- Second guess γ_1

$$y_2(0) = \gamma_1 = 0$$

- Third guess γ_2

$$y_2(0) = \gamma_2 = 1.55$$



Live coding: Shooting method

- Open Colab: [Shooting method]

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

The shooting method

$$\frac{dy_1}{dx} = y_2, \qquad \frac{dy_2}{dx} = 4(y_1 - x), \qquad y_1(0) = 0$$

- First guess γ_0

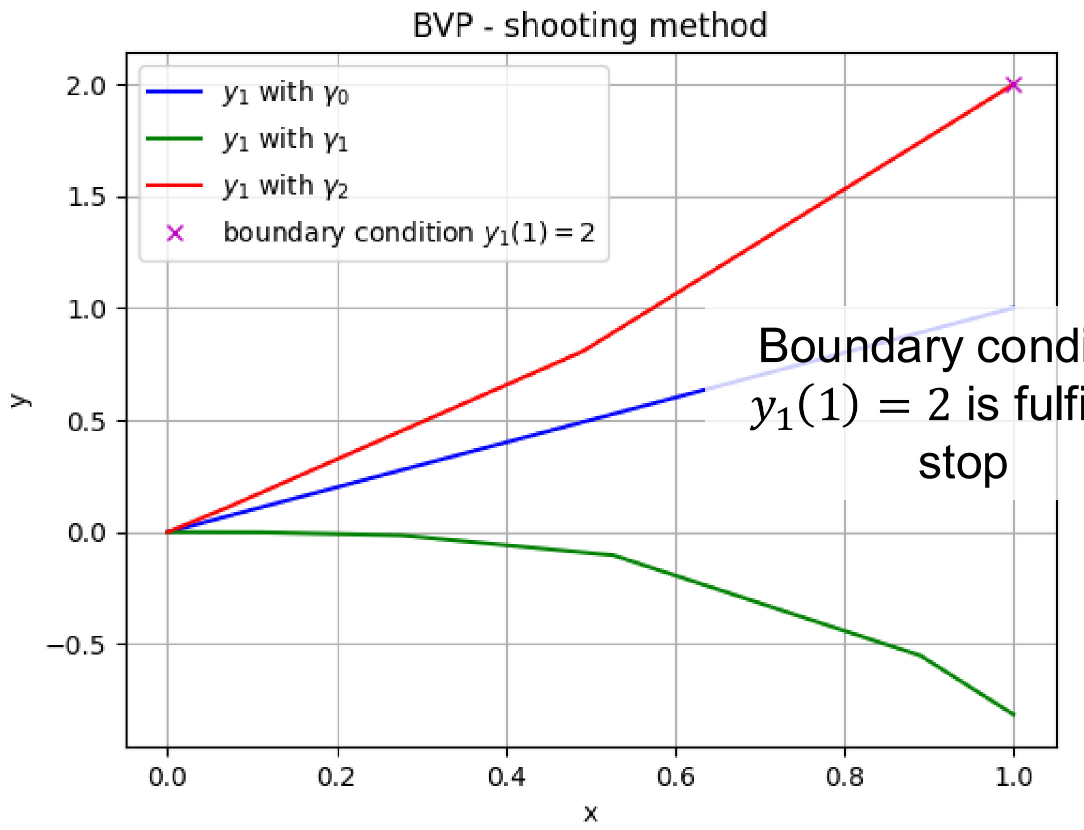
$$y_2(0) = \gamma_0 = 1$$

- Second guess γ_1

$$y_2(0) = \gamma_1 = 0$$

- Third guess γ_2

$$y_2(0) = \gamma_2 = 1.55$$



Boundary condition $y_1(1) = 2$ is fulfilled, stop

So far, we were randomly guessing the initial conditions. However, we want to make clever, strategic decisions on the initial conditions.

Agenda

- **Boundary value problems (BVPs)**
  - Shooting method
    a. Linear interpolation
    b. Root-finding with secant method
  - *Solve_bvp* function

Updating guess: Linear interpolation

- Take two random guesses γ_0, γ_1
- Compute next guess using linear interpolation between the previous two results. :

$$\frac{\Delta y_{1_{0\to1}}}{\Delta\gamma_{0\to1}} = \frac{\Delta y_{1_{1\to2}}}{\Delta\gamma_{1\to2}}$$

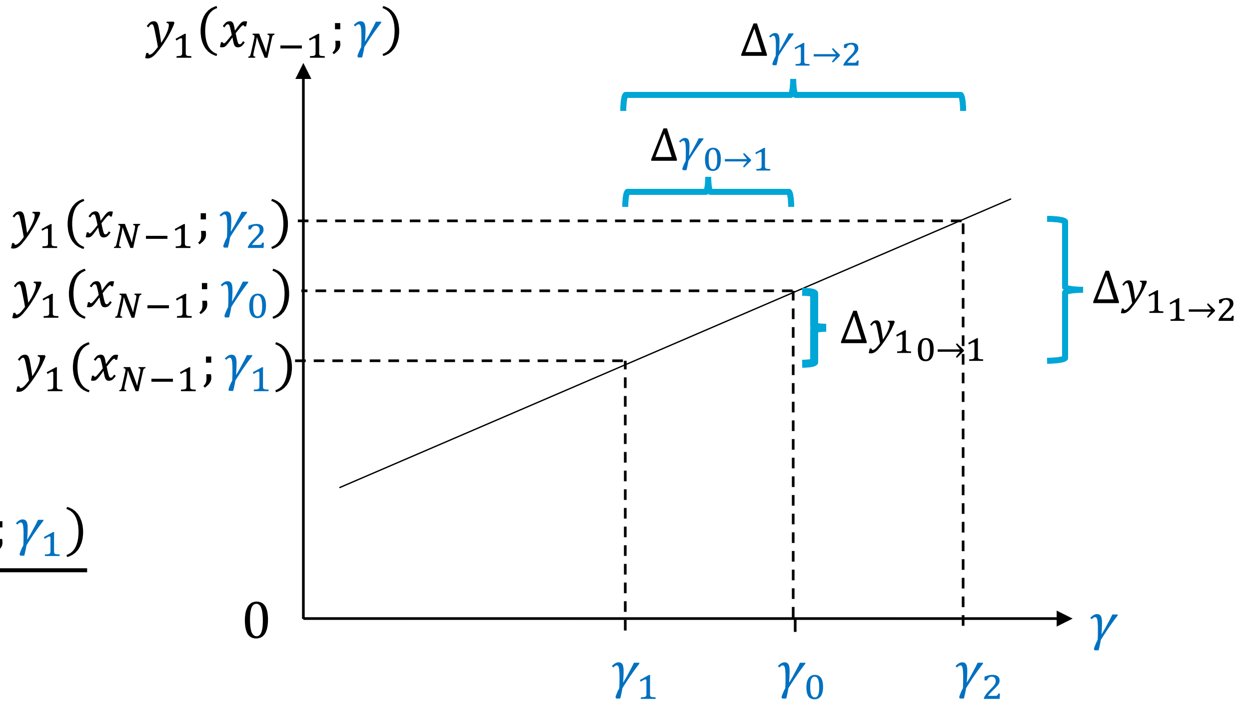
Solution of IVP at right bounday x_{N-1}using guess γ_1

Boundary condition at right bounday x_{N-1}.

$$\frac{y_1(x_{N-1};\gamma_1) - y_1(x_{N-1};\gamma_0)}{\gamma_1 - \gamma_0} = \frac{y_1(x_{N-1}) - y_1(x_{N-1};\gamma_1)}{\gamma_2 - \gamma_1}$$

Guess

$$\gamma_2 = \gamma_1 + (\gamma_1 - \gamma_0)\frac{y_1(x_{N-1}) - y_1(x_{N-1};\gamma_1)}{y_1(x_{N-1};\gamma_1) - y_1(x_{N-1};\gamma_0)}$$



NOTE

This is the general formula, in our example x_{N-1} is the right boundary index located at $x = 1$.

Updating guess: Linear interpolation

- Take two random guesses γ_0, γ_1
- Next guess,

$$\gamma_2 = \gamma_1 + (\gamma_1 - \gamma_0)\frac{y_1(x_{N-1}) - y_1(x_{N-1}; \gamma_1)}{y_1(x_{N-1}; \gamma_1) - y_1(x_{N-1}; \gamma_0)}$$

- For our example from the previous slides:

$$\gamma_2 = 0 + (0 - 1)\frac{2 - (-0.831)}{-0.831 - 1} = 1.55$$

- Linear interpolation directly yields correct guess for linear ODEs
  - In practice, the solution from the IVP solver $y(1; \gamma_i)$ is not exact
    → For bad guesses γ_0, γ_1, the next guess γ_2 might be off
  - → Ideally, always use the root-finding using secant method.

Agenda

- **Boundary value problems (BVPs)**
  - Shooting method
    a. Linear interpolation
    b. Root-finding with secant method
  - *Solve_bvp* function

Updating guess: Root-finding with secant method

$$y_1(1; \gamma_0) = 1$$

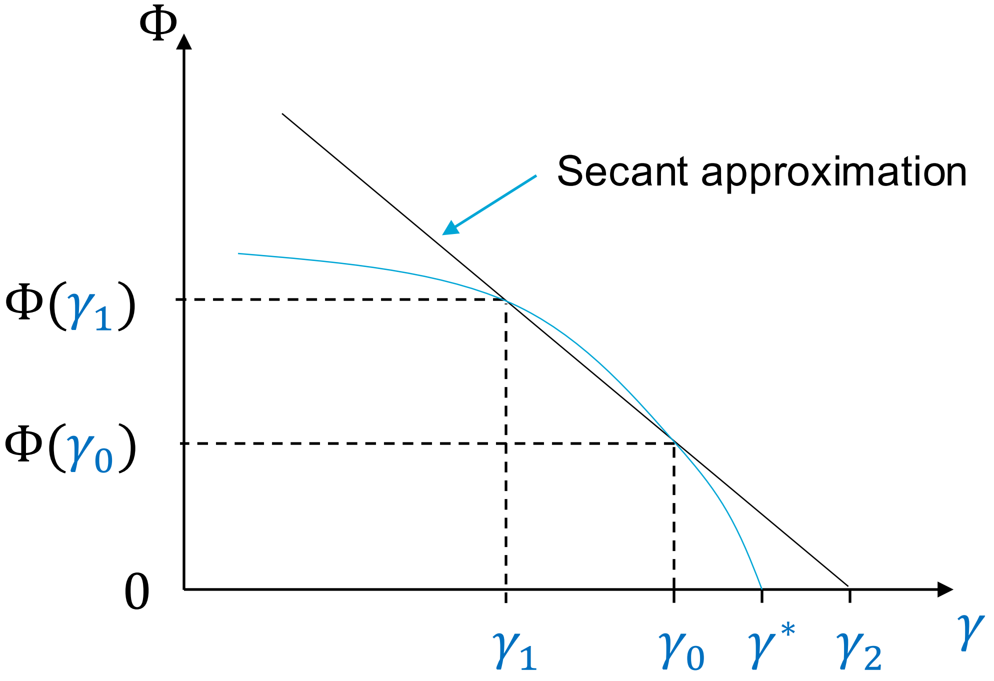
$$y_1(1; \gamma_1) = -0.813$$

- Define a residual function measuring how far the boundary condition is from being satisfied:

$$\Phi(\gamma) = y_1(x_{N-1}; \gamma) - y_1(x_{N-1})$$

Boundary value we reach with our guess γ

Actual boundary value, here = 2



- Objective: $\min |\Phi(\gamma)|$
- Optimal solution: $\Phi(\gamma^*) = 0$

RECAP

Use a root finding algorithm to find the optimal solution (c.f., lecture 2)

Optimal solution of the residual function

- Residual function

$$\Phi(\gamma_{i+1}) = y_1(x_{N-1}; \gamma_{i+1}) - y_1(x_{N-1})$$

- Taylor expansion

$$\Phi(\gamma_{i+1}) = \Phi(\gamma_i + \Delta\gamma) = \Phi(\gamma_i) + \left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i} \Delta\gamma + \mathcal{O}(\Delta\gamma^2)$$

Current guess

Changing the current guess

Optimal solution of the residual function

- Residual function

$$\Phi(\gamma_{i+1}) = y_1(x_{N-1};\gamma_{i+1}) - y_1(x_{N-1})$$

- Taylor expansion

$$\Phi(\gamma_{i+1}) = \Phi(\gamma_i + \Delta\gamma) = \Phi(\gamma_i) + \left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i} \Delta\gamma + \mathcal{O}(\Delta\gamma^2)$$

- The correct initial guess is obtained if $\Phi(\gamma) = 0$

$$0 = \Phi(\gamma_i) + \left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i} \Delta\gamma + \mathcal{O}(\Delta\gamma^2)$$

Optimal solution of the residual function

- Residual function

$$\Phi(\gamma_{i+1}) = y_1(x_{N-1}; \gamma_{i+1}) - y_1(x_{N-1})$$

- Taylor expansion

$$\Phi(\gamma_{i+1}) = \Phi(\gamma_i + \Delta\gamma) = \Phi(\gamma_i) + \left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i} \Delta\gamma + \mathcal{O}(\Delta\gamma^2)$$

- The optimal value for γ satisfies $\Phi(\gamma) = 0$:

$$0 = \Phi(\gamma_i) + \left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i} \Delta\gamma + \mathcal{O}(\Delta\gamma^2) \qquad \left| \begin{array}{l} rearrange, \\ neglect\ \mathcal{O}(\Delta\gamma^2) \end{array} \right.$$

$$\Delta\gamma = -\frac{\Phi(\gamma_i)}{\left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i}}$$

This is how we should change the current guess

Optimal solution of the residual function

$$\Delta\gamma = -\frac{\Phi(\gamma_i)}{\left.\frac{d\Phi}{d\gamma}\right|_{\gamma_i}}$$

$$\Phi(\gamma) = y_1(x_{N-1};\gamma) - y_1(x_{N-1})$$

$$\frac{d\Phi}{d\gamma} = \frac{dy_1(x_{N-1};\gamma_i)}{d\gamma}$$

Known values

$$\Delta\gamma = -\frac{y_1(x_{N-1};\gamma_i) - y_1(x_{N-1})}{\frac{dy_1(x_{N-1};\gamma_i)}{d\gamma}}$$

Use previous guesses to estimate gradient

- Approximate the derivative numerically (backward difference)

$$\frac{dy_1(x_{N-1};\gamma_i)}{d\gamma} = \frac{y_1(x_{N-1};\gamma_i) - y_1(x_{N-1};\gamma_{i-1})}{\gamma_i - \gamma_{i-1}}$$

$$\Rightarrow \quad \Delta\gamma = -\big(y_1(x_{N-1};\gamma_i) - y_1(x_{N-1})\big) \cdot \frac{\gamma_i - \gamma_{i-1}}{y_1(x_{N-1};\gamma_i) - y_1(x_{N-1};\gamma_{i-1})}$$

Optimal solution of the residual function

$$\Delta\gamma = \gamma_{i+1} - \gamma_i \rightarrow \gamma_{i+1} = \gamma_i - \left(y_1(x_{N-1};\gamma_i) - y_1(x_{N-1})\right) \cdot \frac{\gamma_i - \gamma_{i-1}}{y_1(x_{N-1};\gamma_i) - y_1(x_{N-1};\gamma_{i-1})}$$

Yes, this is the same equation as for the linear interpolation. But it is derived differently, and we use it in an iterative approach.

- With

$$y_1(x_{N-1};\gamma_0) = 1, \qquad y_1(x_{N-1};\gamma_1) = -0.813, \qquad y_1(x_{N-1}) = 2, \qquad \gamma_0 = 1, \qquad \gamma_1 = 0$$

- Improve the guess

$$\gamma_2 = \gamma_1 + \Delta\gamma_1$$

$$\gamma_2 = 0 + 1 \cdot 1.55 = 1.55$$

The shooting method residual function

Loop intuition:

Start:

- System of 1st order ODEs $\frac{dy}{dx} = f(x, y)$
- Initial guesses γ_0, γ_1
- Solve IVP with $\gamma_0, \gamma_1 \rightarrow y_1(x_{N-1}; \gamma_0), y_1(x_{N-1}; \gamma_1)$

Check:

- $|\Phi| = |y_1(x_{N-1}; \gamma_i) - y_1(x_{N-1})| \leq \epsilon$

Then:

- Compute $\gamma_{i+1} = \gamma_i - \left(y_1(x_{N-1}; \gamma_i) - y_1(x_{N-1})\right) \cdot \frac{\gamma_i - \gamma_{i-1}}{y_1(x_{N-1}; \gamma_i) - y_1(x_{N-1}; \gamma_{i-1})}$
- Solve IVP with $\gamma_{i+1} \rightarrow y_1(x_{N-1}; \gamma_{i+1})$
- Set $\gamma_i = \gamma_{i+1}, \gamma_{i-1} = \gamma_i$

HINT: connect the dots

The shooting method solves IVPs. Hence, it inherits the stability (or instability) of the associated IVP solver (cf., lecture 5).

Live coding: Optimal shooting method

- Open Colab: Optimal shooting

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

The shooting method for M equations

- Equations

$$\frac{dy_i}{dx} = f_i(x, y_1, y_2, \dots, y_M), \qquad x \in [x_0, x_{N-1}], \qquad i = 1, \dots, M$$

- Boundary conditions at x_{N-1} need to be replaced by boundary conditions at x_0

$$y_j(x_{N-1}) = y_{j,N-1}, \qquad j \subset i$$

- Initial guesses

$$y_j(x_0) = \gamma_0^j, \qquad j \subset i$$

The shooting method for M equations

- Equations

$$\frac{dy_i}{dx} = f_i(x, y_1, y_2, \dots, y_M), \qquad x \in [x_0, x_{N-1}], \qquad i = 1, \dots, M$$

- Improve the guesses

$$\Delta\boldsymbol{\gamma} = [\boldsymbol{J}(x_{N-1}, \boldsymbol{\gamma})]^{-1} \begin{pmatrix} y_{j[0]}(x_{N-1}; \gamma_{j[0]}) \\ \dots \\ y_{j[-1]}(x_{N-1}; \gamma_{j[-1]}) \end{pmatrix}$$

RECAP

Jacobian known from Newton-Raphson method (c.f., lecture 2).

- With $\boldsymbol{J}(x_{N-1}; \boldsymbol{\gamma})$ being the Jacobian

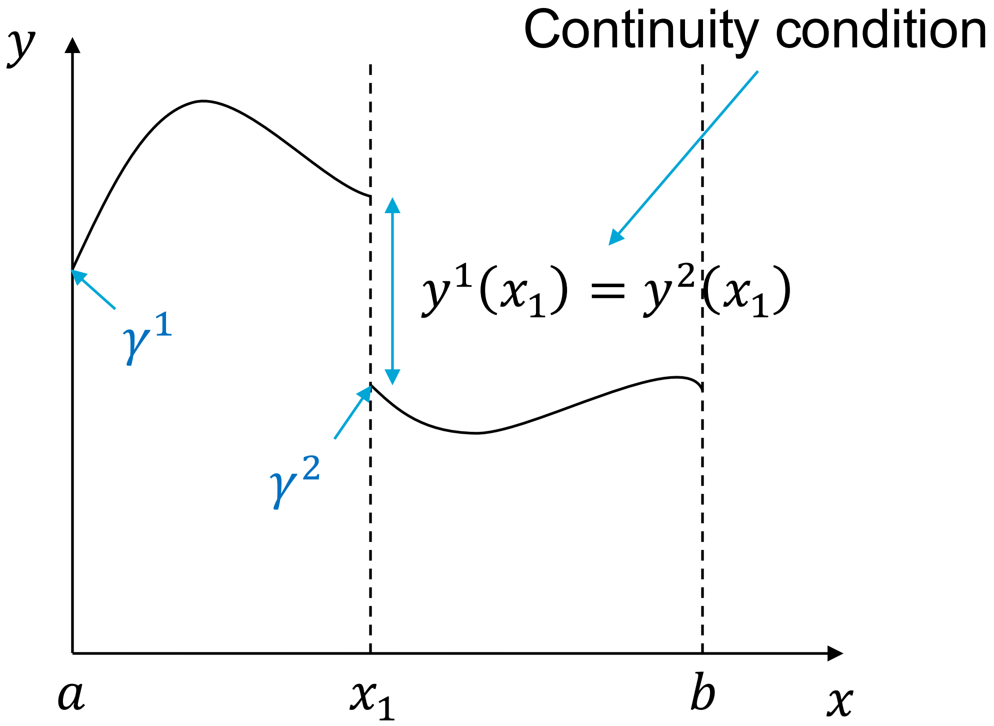
$$\boldsymbol{J}(x_{N-1}, \boldsymbol{\gamma}) = \begin{pmatrix} \frac{\partial y_{j[0]}}{\partial \gamma_{j[0]}} & \cdots & \frac{\partial y_{j[0]}}{\partial \gamma_{j[-1]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{j[-1]}}{\partial \gamma_{j[0]}} & \cdots & \frac{\partial y_{j[-1]}}{\partial \gamma_{j[-1]}} \end{pmatrix}$$

Approximated with previous guess, only exact for linear ODEs

$$\frac{dy}{d\gamma} = \frac{y(x_{N-1}; \gamma_i) - y(x_{N-1}; \gamma_{i-1})}{\gamma_i - \gamma_{i-1}}$$

The multiple shooting method

- Advanced shooting method.
- Breaks domain into sub-domains and solves IVP in each sub-domain.
- Add continuity and boundary conditions for each sub-domain.
- Advantages:
  - + Improved stability
  - + Parallelization
  - + More flexibility for initial guesses



Finite differences vs. shooting method

Finite difference method

- + ODE is solved only once.
- - Nonlinear ODEs require to solve a system of nonlinear equations.

Shooting method

- + Solution of nonlinear equations is fairly straightforward.
- - ODE needs to be solved several times.

➡ Choice depends on the problem.

Amos Gilat, Vish Subramaniam (2013) - Numerical Methods for Engineers and Scientists

Agenda

- **Boundary value problems (BVPs)**
  - Shooting method
    a. Linear interpolation
    b. Secant method
  - *Solve_bvp* function

SciPy's *solve_bvp*

- Uses the collocation method:
  - modification of the Multiple Shooting Method and the Finite Difference Method.
- Function arguments:
  - *fun*: Function containing the ODE system.
  - *bc*: Boundary conditions defined as a function.
  - *x:* Domain, must include domain boundaries.
  - *y*: Initial guess for function values.

Read the complete description of *solve_bvp* (link) at home.

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:

```
dy / dx = f(x, y, p) + S * y / (x - a), a <= x <= b
bc(y(a), y(b), p) = 0
```

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.

The last singular term on the right-hand side of the system is optional. It is defined by an n-by-n matrix S, such that the solution must satisfy S y(a) = 0. This condition will be forced during iterations, so it must not contradict boundary conditions. See [2] for the explanation how this term is handled when solving BVPs numerically.

Problems in a complex domain can be solved as well. In this case, y and p are considered to be complex, and f and bc are assumed to be complex-valued functions, but x stays real. Note that f and bc must be complex differentiable (satisfy Cauchy-Riemann equations [4]), otherwise you should rewrite your problem for real and imaginary parts separately. To solve a problem in a complex domain, pass an initial guess for y with a complex data type (see below).

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,).

SciPy's *solve_bvp*

- *fun*: Function containing the ODE system.

$$\underbrace{\frac{dy_1}{dx}}_{dydx[0]} = \underbrace{y_2}_{y[1]}, \qquad \frac{dy_2}{dx} = 4(y_1 - x)$$

```python
import numpy as np

def dy(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    """First order ODEs from d2y/dx2=4(y-x).

    Args:
        x (np.ndarray): Grid points.
        y (np.ndarray): Function values.
    Returns:
        np.ndarray: Right hand-side of ODEs.
    """
    dydx = np.zeros(y.shape)
    dydx[0] = y[1]
    dydx[1] = 4*(y[0]-x)
    return dydx
```

- *bc*: Boundary conditions defined as a function.

$$0 = \underbrace{y_1}_{[0]}\underbrace{(0)}_{ya}, \qquad 0 = \underbrace{y_1}_{[0]}\underbrace{(1)}_{yb} - 2$$

```python
import numpy as np

def my_bc(ya: np.ndarray, yb: np.ndarray) -> np.ndarray:
    """Residuals of the boundary conditions.

    Args:
        ya (np.ndarray): Function values at x0.
        yb (np.ndarray): Function values at x(N-1).
    Returns:
        np.ndarray: Residuals of boundary conditions.
    """
    return np.array([ya[0],yb[0]-2])
```

SciPy's *solve_bvp*

- Solve the BVP

```
import scipy.integrate

x = np.linspace(0,1,101)  # x grid points
y_guess = np.zeros((2,len(x)))  # first guess, shape: no. dependent variables, no. grid points
bvp_result = scipy.integrate.solve_bvp(dy, my_bc, x, y_guess)
```

- *solve_bvp* returns a tuple including, besides others:
  - *sol*: Polynomial approximation of the solution. This is a function that takes grid points as input.
  - *x*: Nodes of the final mesh (grid points inside solver).
  - *y*: Solution values at the mesh nodes.
- Note the difference between *sol* and *y*!

Live coding: SciPy's *solve_bvp()*

- Open Colab: solve_bvp function

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Learning goals of this lecture

After successfully completing this lecture, you are able to…

- explain numerical solution methods for boundary value problems (BVPs), namely,
  - finite difference method,
  - shooting method.
- implement the shooting method for boundary value problems (BVPs) from scratch.
- use Python libraries' built-in functions for numerical solution approaches to BVPs.

Thank you very much for your attention!

Computer exam

- Computer exam has two parts: (i) a theory part and (ii) a practical programming part
- (i) The theory part consists of questions in ANS (link)
- (ii) The programming part will be similar to the assignments
  - IDE: Anaconda Jupyter Notebook (link)
  - Points given for solution format, e.g.,

    - Functions need to have type hints and descriptive docstrings

    - Plots need to have title, axis labels, legend (if more than two lines in one plot)
  - No autochecks provided
- Computer exam is closed book, but we provide
  - a cheat sheet with common Python commands (see Brightspace/Resources)
  - documentation for important build-in functions, e.g., *solve_bvp* (link).

Computer exam

Monday 03-01-2025
09:00-12:00

CEG-Computer Room

Practice exam

- We will publish solutions to the practice exam.
- Solving the assignments or practice exam can help you test your skills and knowledge level; however, you should not expect the questions of the final exams to be a simple repetition of questions from the assignments or practice exam. We expect students to be able to apply the learned theory and skills to a problem that they have not seen before.