UNIVERSITY OF AMSTERDAM

Assignment #2

The Square That Sets the Pace

Effect of hot–square size on Jacobi convergence in a 2-D Laplace problem

Friday 7th November, 2025 16:55

Student:
Maarten Stork - 15761770
Lecturer:
Dr. Rob Belleman

Course:
Scientific Visualization and Virtual Reality
Course code:
5284SVVR6Y

Abstract

This report studies how the size of a central hot square affects the convergence of a Jacobi solver for a 2-D steady-state heat problem. Using the by-UvA-provided setup, we solve Laplace's equation on a 9×9 plate with Dirichlet boundaries (cool bottom, warm top, linear side ramps) and a fixed-temperature heater at the center. The solver is a standard Jacobi method with a five-point stencil and a max-norm δ stopping rule. We vary only the heater area (as a fraction of the domain) and compare both the iterations needed to reach the tolerance and the full δ–vs–iteration history. Results are shown in two ways: (i) ParaView outputs (`VTI` frames with a `PVD` time series) documented via a Haber&Mcnabb pipeline, and (ii) Matplotlib figures/animations for a lightweight web companion. Across the tested sizes, larger interior Dirichlet regions converge in fewer iterations.

1 Introduction

Laplace's equation sits at the heart of many steady-state phenomena, among which heat flow, electrostatics, and potential fluid models [2, 4]. Its solutions are entirely shaped by what we fix on the boundary (or inside the domain), so simple geometric changes can alter both the final field and how quickly an iterative solver gets there. Jacobi's method, while basic, is ideal for seeing this link between problem geometry and convergence behaviour, and scientific visualization makes that link easily detectable by showing the field evolve under consistent scales and annotations [1, 5, 6].

We adopt the UvA "cartoon" setup [1]: a square plate with a cool bath at one edge, a warm edge opposite, linear side ramps, and a fixed-temperature hot square embedded at the center (think of a small heater inside the material). Starting from a plain initial guess, the system relaxes to steady state. In this study we vary only one element (the area

[1]Access to the full cartoon through this Github repository: `https://github.com/MaartenStork/SVVR/blob/main/GeneralPlots/CartoonUVA.pdf`

of the embedded heater) and examine how that modelling choice affects solver effort, giving us the following research question:

How does the size of the central hot square affect the number of iterations a Jacobi solver needs to reach a given tolerance?

The aim is to provide a clear, visual account of convergence alongside a simple quantitative comparison across heater sizes, keeping all other modelling and visualization choices fixed.

2 Methods

Problem setup We model steady heat conduction on a square plate: the bottom edge is held cool (T=32°), the top edge warm (T=100°), and the left/right edges ramp linearly between them (T=32 to 100°); a hot square (T=212°) embedded at the center is fixed at a higher temperature (an interior Dirichlet region). In steady state the temperature field $T(x, y)$ satisfies Laplace's equation $\nabla^2 T = 0$ on the plate outside the hot square, with Dirichlet conditions on all outer edges and on the embedded square. This follows the computational science pipeline from physical model $\longrightarrow$ mathematical model $\longrightarrow$ numerical model.

We discretize the domain on a uniform lattice (i, j) and solve the 2-D Laplace problem with the standard five-point stencil, updating only non-Dirichlet nodes. The Jacobi update is

$$T_{i,j}^{(n+1)} = \tfrac{1}{4}\left(T_{i-1,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j-1}^{(n)} + T_{i,j+1}^{(n)}\right),$$

applied wherever the node is not fixed; boundary nodes and the hot-square mask are copied through each sweep. Convergence is monitored with the max-norm

$$\delta^{(n+1)} = \max_{i,j} \left| T_{i,j}^{(n+1)} - T_{i,j}^{(n)} \right|,$$

and the iteration stops when $\delta \le \varepsilon$ (tolerance) or a safety cap of 20,000 iterations is reached.

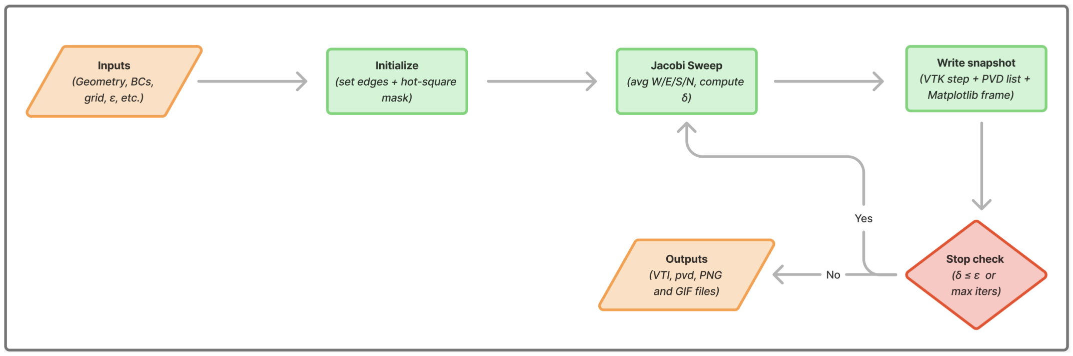
Inputs
(Geometry, BCs, grid, ε, etc.)
Initialize
(set edges + hot-square mask)
Jacobi Sweep
(avg W/E/S/N, compute δ)
Write snapshot
(VTK step + PVD list + Matplotlib frame)
Yes
Stop check
(δ ≤ ε or max iters)
No
Outputs
(VTI, pvd, PNG and GIF files)

Figure 1: *Flowchart (or pipeline) of the data gathering process.*

This implementation (almost completely) mirrors the Jacobi pseudocode provided by the University of Amsterdam[2] and is depicted in Figure 2. The steps are: (1) initialize the lattice and set Dirichlet values on the outer edges as well as on the central hot-square mask; (2) iterate Jacobi sweeps using west/east/north/south neighbours while tracking δ each sweep; (3) output a snapshot at the chosen cadence (for ParaView: `.vti` plus a `.pvd` time collection); and (4)

[2]Readable in the GitHub at `https://github.com/MaartenStork/SVVR/blob/main/GeneralPlots/slides.pdf`.

stop when δ falls below the tolerance. As stated, we follow the slides' Jacobi loop, but instead use Dirichlet edges (no wrapping) to match the fixed-temperature boundary cartoon better; inside the loop we "clamp" neighbor indices so they never go outside the grid. This doesn't change the result, because all boundary cells already have fixed temperatures (Dirichlet). So if a neighbor would fall off the edge, we just read the boundary value that's fixed anyway. The core code of this implementation is also shown in the Appendix.

Visualization pipeline We provide two complementary visualizations, with different levels of ease:

1. For the web companion[3], we render Matplotlib frames and combine these in a GIF with a fixed colormap range, axes/ticks, legend, and iteration labels; this lightweight view is meant for quick exploration and for embedding on the page, for the reviewer of this paper to run the simulations in a vacuum.

2. For the full pipeline in ParaView, the solver writes VTK-XML ImageData files (`.vti`) together with a `.pvd` collection so the sequence loads as a single time-aware source. In ParaView, the pipeline follows the Haber&McNabb model [3], also depicted in Figure 2: (1) the data source obtained through the code (flowchart described in Figure 2), (2) the mapping, as well as the additions of the contour and the annotating of time, and finally (3) rendering it all within Paraview.

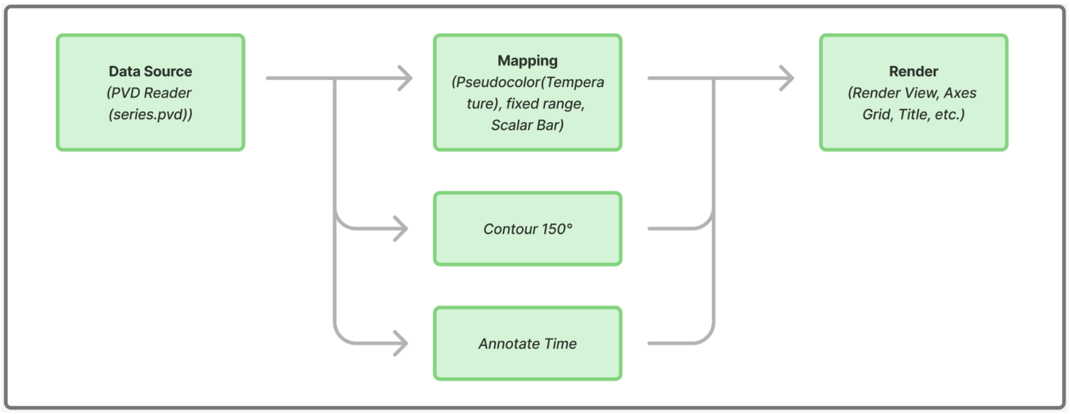
Data Source
(PVD Reader (series.pvd))
Mapping
(Pseudocolor(Temperature), fixed range, Scalar Bar)
Render
(Render View, Axes Grid, Title, etc.)
Contour 150°
Annotate Time

Figure 2: *Pipeline of the data visualization process.*

This reproduces the core visualization pipeline while offering both an accessible web version and a richer ParaView analysis. [3].

Visualization Choices Here we provide a few explanations for the visualization choices that went in to designing the plots presented in the Results section.

1. Firstly, the actual visualization. Both Matplotlib and ParaView use a physical $x, y \in [0, 9]$ domain to match the $9m \times 9m$ plate, with axes, ticks, and labels shown to give a sense of scale. We keep a single, fixed blue to red colormap range across both visualizations; a scalar bar (vertical, on the right) reinforces the mapping. In Matplotlib we use `turbo`; its hues differ slightly from the slide/ParaView palette (with a brighter yellow and green in there), and given the deadline on this particular assignment, we did not define a custom LUT. Time/iteration appears at the top of each frame for consistency. In ParaView we display it via *Annotate Time*; in Matplotlib it is rendered in the title - numerically they are the same (e.g., Matplotlib "iteration 220" corresponds to ParaView

[3]See the website here, feel free to play around with it: `https://jacobi-simulation-backend.onrender.com/`.

"time = 220"). For the Paraview figures we overlay a single *Contour* at a representative isotherm to indicate warm regions.

2. Secondly, the plots meant for more insight into convergence behavior. For plotted convergence speed, we decide to use blue points (like our main theme for the snapshots of the visualization), with no lines inbetween, as we do not want to imply some form of linearity. For the delta-versus-iteration plot, we plot them in different values, shifting the value from purple to yellow, following Mackinlay's effectiveness ordering for ordered data [5].

Answering the research question We run a controlled sweep over the hot–square size $f \in (0, 1]$ (fraction of plate side) in steps of $\Delta f = 0.05$, keeping all other settings fixed (geometry, grid, boundary conditions). For each f, we run the Jacobi solver to a max-norm tolerance of $\varepsilon = 10^{-4}$ and record (i) the iterations to convergence $N(f)$ and (ii) the full δ-versus-iteration history. Because the system is deterministic, a single run per f suffices. We then plot $N(f)$ to read how convergence time varies with heater size, and use the δ curves (together with consistent visualization settings) to confirm the settling behaviour across conditions.

3 Results

Visualizations Figure 3 shows the ParaView render for some of the first, middle and last frames. The contour described in the methods is visible. These are just single snapshots - a GIF of this visualization is available in the resources added to this report[4] for the inspection of the entire animation.

Figure 4 shows the Matplotlib rendering, which one can also play around with in our online environment referenced before.

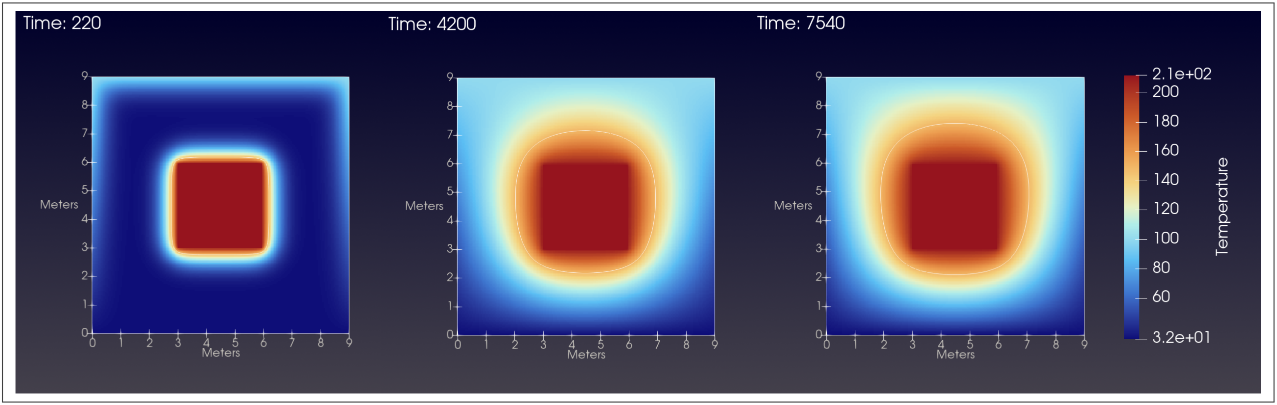

Figure 3: *Three snapshots of the ParaView visualization: early, intermediate, and late stages of the temperature evolution.*

Hot-square tests Figure 5 showcases several hot–square sizes to illustrate how the central heater actually changes the entire field. The side-by-side views are meant to give an intuitive feel for the experiment before we turn to the convergence results.

Figures 6 and 7 summarize solver behaviour. Figure 6 shows the speed (iterations required to reach the tolerance) so differences are visible at a glance. Figure 7 shows the history (the max-norm δ versus iteration), for each hot-square size.

[4]See the GIF in the Github here: `https://github.com/MaartenStork/SVVR/blob/main/GeneralPlots/ParaViewGIF.gif`, as well as the AVI file here: `https://github.com/MaartenStork/SVVR/blob/main/GeneralPlots/ParaViewVideo.avi`.

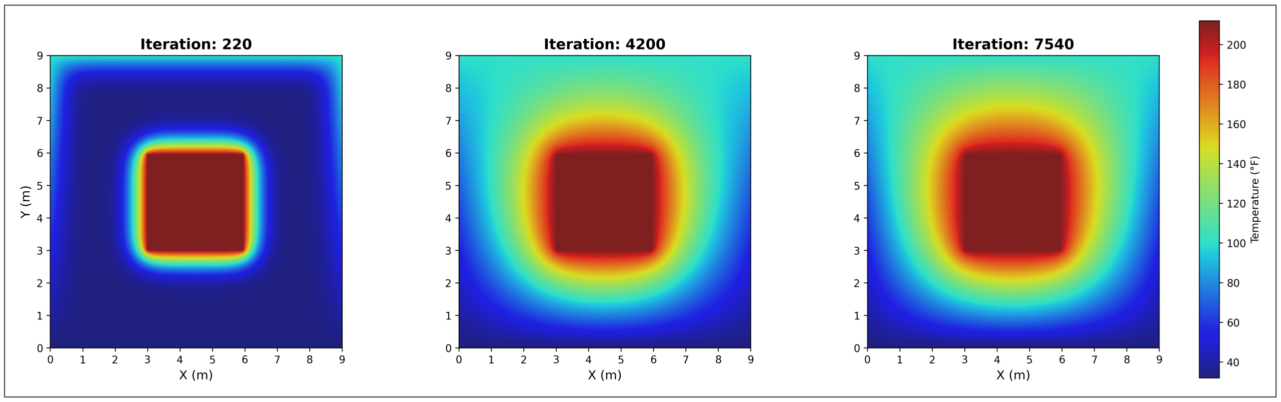

Figure 4: *Three snapshots of the Matplotlib visualization: early, intermediate, and late stages of the temperature evolution. A simpler version of the ParaView visualization, meant for easy experimentation online.*

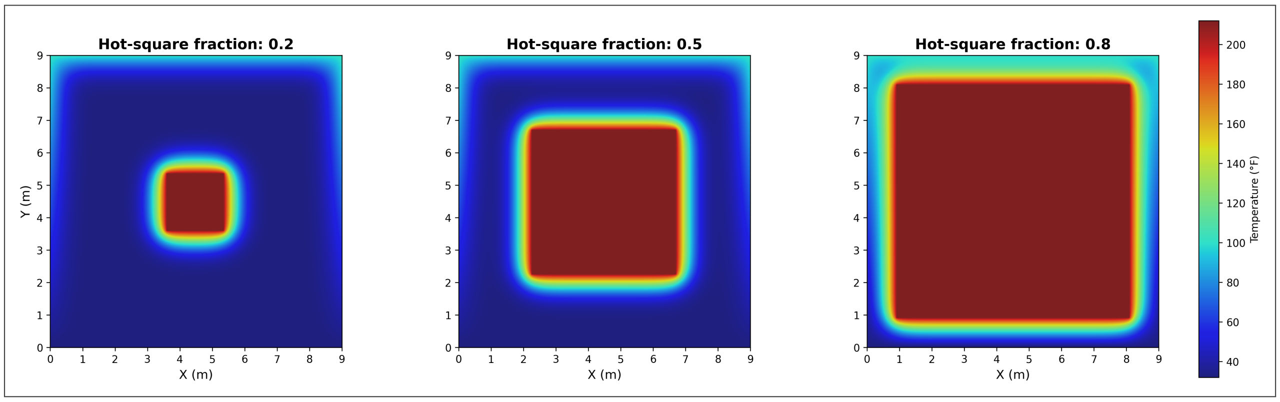

Figure 5: *Three snapshots at iteration 200 with different sizes depicted in the Matplotlib visualization. Left to right; sizes 0.2, 0.5 and 0.8 respectively.*

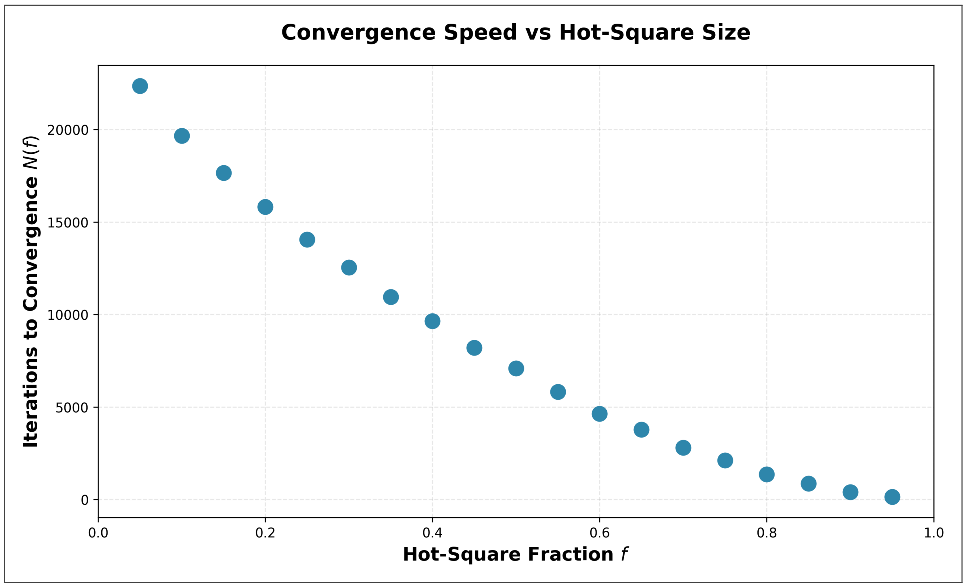

Figure 6: *The course of the convergence speed against the hot-square size.*

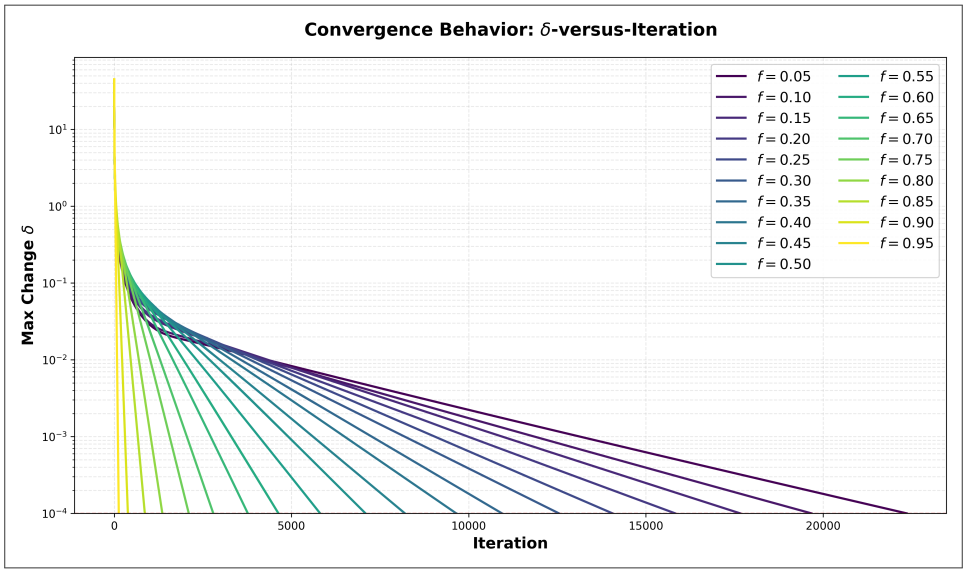

Figure 7: *The δ-versus-iteration behavior for different hot-square sizes, with 'f' meaning 'hot-square fraction'.*

4 Discussion

What we see in the visualizations. With a fixed color range and a single T=150° contour, the early frames show a small hot patch that grows and smooths as the solver runs. Even though the heater is a square, the isotherms quickly look rounded: outside the fixed patch, the solution to Laplace's equation smooths corners and the heat spreads evenly in all directions, so level sets bend into curves rather than keeping a square outline. In the late frames the field is wide and almost concentric around the heater. Larger heaters take over more of the plate earlier, leaving a smaller region that still needs to relax, and the steady pattern appears sooner.

What we see in the graphs. The numbers match what we see in pictures and videos. The iterations-to-tolerance curve $N(f)$ drops as the heater fraction f grows: bigger heated regions need fewer sweeps. The δ–versus–iteration curves all fall quickly at first and then more slowly; on a semilog plot they look roughly linear in the tail. Smaller heaters shift these curves to the right (longer tails), while larger heaters pull them left (shorter tails). Intuitively, increasing f both reduces how many unknown nodes are left and shortens the distance over which corrections must diffuse, so the solver finishes in fewer iterations.

5 Conclusion

These results have given us an answer to our main research question:

How does the size of the central hot square affect the number of iterations a Jacobi solver needs to reach a given tolerance?

Holding geometry, grid, and tolerance fixed, larger interior Dirichlet regions (= bigger hot squares) lead to faster convergence, i.e., fewer Jacobi iterations to reach the max-norm threshold. This trend is consistent across the sweep of heater sizes and is supported both by the iteration counts and by the shape of the δ–history curves.

Looking ahead, this same setup can now serve as a testbed: we can compare different solvers, vary the heater's shape and position, or mix boundary types. With the base in place, exploring these options becomes straightforward.

References

[1] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, Madison, 1983.

[2] Jean Baptiste Joseph Fourier. *Théorie analytique de la chaleur*. Gauthier-Villars et fils, 1888.

[3] Robert B Haber and David A McNabb. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in scientific computing*, 74:93, 1990.

[4] John David Jackson and Ronald F Fox. Classical electrodynamics, 1999.

[5] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.

[6] Yousef Saad. Iterative methods for linear systems of equations: A brief historical journey. *arXiv preprint arXiv:1908.01083*, 2019.

Appendix

For completeness, I also showcase the core script below. The complete repository (containing the experimental setup, the code serving the webapp, the data, the cartoon, and more) is available on GitHub[5].

Listing 1: Data cleaning + visualization script

```python
from __future__ import annotations
import argparse
import math
import os
from pathlib import Path
from typing import Tuple, List
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.colors import LinearSegmentedColormap
import numpy as np

# ----------------------------
# Utilities
# ----------------------------

def linspace(n: int, start: float, end: float) -> List[float]:
   if n == 1:
      return [start]
   step = (end - start) / (n - 1)
   return [start + i * step for i in range(n)]

```

[5] See the full codebase here: `https://github.com/MaartenStork/SVVR`

```
def write_vti_image_data(
    filename: str,
    grid: "list[list[float]]",
    dx: float,
    dy: float,
    origin: Tuple[float, float, float] = (0.0, 0.0, 0.0),
    name: str = "Temperature",
):
    """
    Write 2D scalar field as VTK-XML ImageData (.vti), ASCII.
    """
    ny = len(grid)
    nx = len(grid[0])
    nz = 1
    # VTK-XML expects x fastest, then y, then z (same order you already write)
    with open(filename, "w") as f:
        f.write('<?xml version="1.0"?>\n')
        f.write('<VTKFile type="ImageData" version="0.1" byte_order="LittleEndian">\n')
        f.write(f' <ImageData WholeExtent="0 {nx-1} 0 {ny-1} 0 {nz-1}" '
                f'Origin="{origin[0]} {origin[1]} {origin[2]}" '
                f'Spacing="{dx} {dy} 1">\n')
        f.write(f' <Piece Extent="0 {nx-1} 0 {ny-1} 0 {nz-1}">\n')
        f.write(f' <PointData Scalars="{name}">\n')
        f.write(f' <DataArray type="Float32" Name="{name}" format="ascii">\n')
        for j in range(ny):
            row = grid[j]
            f.write(" ".join(f"{float(row[i]):.6f}" for i in range(nx)) + "\n")
        f.write(' </DataArray>\n')
        f.write(' </PointData>\n')
        f.write(' <CellData/>\n')
        f.write(' </Piece>\n')
        f.write(' </ImageData>\n')
        f.write('</VTKFile>\n')

def write_pvd_collection(pvd_path: str, vtk_filenames: List[str], times: List[float]):
    """
    Write a ParaView Data (PVD) collection file so the sequence loads as a time series.
    """
    assert len(vtk_filenames) == len(times)
    with open(pvd_path, "w") as f:
        f.write('<?xml version="1.0"?>\n')
        f.write('<VTKFile type="Collection" version="0.1" byte_order="LittleEndian">\n')
        f.write(" <Collection>\n")
        for fn, t in zip(vtk_filenames, times):
            f.write(f' <DataSet timestep="{t:.6f}" file="{Path(fn).name}"/>\n')
        f.write(" </Collection>\n")
        f.write("</VTKFile>\n")

def write_csv_snapshot(path: str, grid: "list[list[float]]"):
    ny = len(grid)
    nx = len(grid[0])
    with open(path, "w") as f:
        f.write("i,j,T\n")
        for j in range(ny):
            for i in range(nx):
```

```python
                f.write(f"{i},{j},{grid[j][i]:.6f}\n")

def create_temperature_frame(grid: "list[list[float]]", step: int, delta: float,
                             vmin: float, vmax: float, dpi: int = 100,
                             extent: tuple = None) -> np.ndarray:
    """
    Create a matplotlib figure of the temperature field and return it as an array.
    extent: (xmin, xmax, ymin, ymax) for physical dimensions
    """
    import matplotlib
    import matplotlib.colors as mcolors
    matplotlib.use('Agg') # Use non-interactive backend

    ny = len(grid)
    nx = len(grid[0])

    # Convert grid to numpy array (flip y-axis for correct orientation)
    T_array = np.array(grid)

    fig, ax = plt.subplots(figsize=(8, 7), dpi=dpi)

    # Create desaturated jet colormap to match ParaView's smooth blue-to-red gradient
    base_cmap = plt.cm.jet
    # Desaturate by blending with gray (reduces harsh transitions)
    def desaturate_colormap(cmap, sat=0.7):
        colors = cmap(np.linspace(0, 1, 256))
        # Convert to HSV, reduce saturation, convert back
        h = colors[:, 0]
        s = colors[:, 1] * sat # Reduce saturation
        v = colors[:, 2]
        # For simplicity, just blend with gray
        gray = 0.5
        colors[:, :3] = colors[:, :3] * sat + gray * (1 - sat)
        return mcolors.ListedColormap(colors)

    desaturated_rainbow = desaturate_colormap(base_cmap, sat=0.75)

    # Use extent for physical dimensions if provided
    if extent:
        im = ax.imshow(T_array, origin='lower', cmap=desaturated_rainbow, vmin=vmin, vmax=vmax,
                       aspect='equal', interpolation='bilinear', extent=extent)
    else:
        im = ax.imshow(T_array, origin='lower', cmap=desaturated_rainbow, vmin=vmin, vmax=vmax,
                  aspect='equal', interpolation='bilinear')

    # Add colorbar
    cbar = plt.colorbar(im, ax=ax, label='Temperature (F)')

    # Add title with iteration info
    ax.set_title(f'Temperature Field Evolution\nIteration: {step}, Delta: {delta:.2e}',
                fontsize=12, fontweight='bold')

    # Set axis labels based on whether physical dimensions are provided
    if extent:
        ax.set_xlabel('X Position (meters)')
        ax.set_ylabel('Y Position (meters)')
    else:
        ax.set_xlabel('X Position')
        ax.set_ylabel('Y Position')

    # Convert figure to numpy array (using buffer_rgba for compatibility)
    fig.canvas.draw()
    buf = np.asarray(fig.canvas.buffer_rgba())
    # Convert RGBA to RGB
    frame = buf[:, :, :3].copy()

    plt.close(fig)
    return frame

# ----------------------------
# Physics / Discretization
# ----------------------------

def build_indices_hot_square(nx: int, ny: int, hot_fraction: float = 1/3.0) -> Tuple[int, int, int, int]:
    """
    Hot square is hot_fraction of plate width/height, centered.
    We keep it integer-index aligned and inclusive bounds.
    """
    hot_w = int(round(nx * hot_fraction))
    hot_h = int(round(ny * hot_fraction))

    i0 = (nx - hot_w) // 2
    i1 = i0 + hot_w - 1
    j0 = (ny - hot_h) // 2
    j1 = j0 + hot_h - 1
    return i0, i1, j0, j1

def apply_dirichlet_boundaries(T: "list[list[float]]", top: float, bottom: float):
    ny = len(T)
    nx = len(T[0])

    # Top and Bottom
    for i in range(nx):
        T[ny - 1][i] = top
        T[0][i] = bottom

    # Left/Right linear ramps from bottom (32) to top (100)
    for j in range(ny):
        y = j / (ny - 1) if ny > 1 else 0.0
        ramp = bottom + (top - bottom) * y
        T[j][0] = ramp
        T[j][nx - 1] = ramp

def apply_hot_square(T: "list[list[float]]", i0: int, i1: int, j0: int, j1: int, temp: float):
    for j in range(j0, j1 + 1):
        for i in range(i0, i1 + 1):
            T[j][i] = temp

def jacobi_step(T_old: "list[list[float]]", T_new: "list[list[float]]",
             fixed_mask: "list[list[bool]]") -> float:
```

```
    """
    Perform one Jacobi sweep; return delta = max |T_new - T_old|
    fixed_mask=True means Dirichlet node (boundary or hot square): copy through.
    """
    ny = len(T_old)
    nx = len(T_old[0])
    delta = 0.0

    for j in range(ny):
        jm1 = max(j - 1, 0)
        jp1 = min(j + 1, ny - 1)
        for i in range(nx):
            if fixed_mask[j][i]:
                T_new[j][i] = T_old[j][i]
            else:
                im1 = max(i - 1, 0)
                ip1 = min(i + 1, nx - 1)
                T_new[j][i] = 0.25 * (T_old[j][im1] + T_old[j][ip1] +
                                      T_old[jm1][i] + T_old[jp1][i])
            d = abs(T_new[j][i] - T_old[j][i])
            if d > delta:
                delta = d
    return delta

# ----------------------------
# Main
# ----------------------------

def main(hot_fraction=None, return_data=False):
    ap = argparse.ArgumentParser(description="Laplace (Jacobi) with ParaView-ready outputs.")
    ap.add_argument("--nx", type=int, default=181, help="Grid points in x (columns).")
    ap.add_argument("--ny", type=int, default=181, help="Grid points in y (rows).")
    ap.add_argument("--tol", type=float, default=1e-3, help="Convergence tolerance on delta.")
    ap.add_argument("--max-iters", type=int, default=20000, help="Safety cap on iterations.")
    ap.add_argument("--output-every", type=int, default=20, help="Write VTK every N iterations.")
    ap.add_argument("--out", type=str, default="out_vtk", help="Output directory.")
    ap.add_argument("--also-csv", action="store_true", help="Write CSV snapshots too.")
    ap.add_argument("--gif", action="store_true", help="Generate animated GIF of temperature evolution.")
    ap.add_argument("--gif-fps", type=int, default=10, help="Frames per second for GIF.")
    ap.add_argument("--hot-fraction", type=float, default=1/3.0, help="Hot square size as fraction of domain
        (default: 1/3)")
    ap.add_argument("--no-vtk", action="store_true", help="Skip VTK file writing (for webapp use)")
    args = ap.parse_args()

    # Override with programmatic argument if provided
    if hot_fraction is not None:
        args.hot_fraction = hot_fraction

    # Physical sizes (meters)
    W = 9.0
    H = 9.0
    dx = W / (args.nx - 1)
    dy = H / (args.ny - 1)

    # Temperatures (degrees)
```

```python
    T_BOTTOM = 32.0
    T_TOP = 100.0
    T_HOT = 212.0

    out_dir = Path(args.out)
    out_dir.mkdir(parents=True, exist_ok=True)

    # Allocate fields
    T_old = [[T_BOTTOM for _ in range(args.nx)] for _ in range(args.ny)]
    T_new = [[T_BOTTOM for _ in range(args.nx)] for _ in range(args.ny)]

    # Apply boundaries and hot square to initial field
    apply_dirichlet_boundaries(T_old, T_TOP, T_BOTTOM)

    i0, i1, j0, j1 = build_indices_hot_square(args.nx, args.ny, args.hot_fraction)
    apply_hot_square(T_old, i0, i1, j0, j1, T_HOT)

    # Build fixed mask (Dirichlet everywhere that must not change)
    fixed = [[False for _ in range(args.nx)] for _ in range(args.ny)]
    # Outer boundaries:
    for i in range(args.nx):
        fixed[0][i] = True
        fixed[args.ny - 1][i] = True
    for j in range(args.ny):
        fixed[j][0] = True
        fixed[j][args.nx - 1] = True
    # Hot square region:
    for j in range(j0, j1 + 1):
        for i in range(i0, i1 + 1):
            fixed[j][i] = True

    # Save step 0
    vtk_paths = []
    times = []
    frames = [] if args.gif else None
    step = 0

    if not args.no_vtk:
        vti0 = out_dir / f"step_{step:05d}.vti"
        write_vti_image_data(str(vti0), T_old, dx, dy, name="Temperature")
        vtk_paths.append(str(vti0))
        times.append(float(step))
        if args.also_csv:
            write_csv_snapshot(str(out_dir / f"step_{step:05d}.csv"), T_old)
    if args.gif:
        # Create extent for 9m x 9m physical dimensions
        extent = (0, W, 0, H)
        frames.append(create_temperature_frame(T_old, step, 0.0, T_BOTTOM, T_HOT, extent=extent))

    # Jacobi iteration loop (slide: do { ... } while (delta > tolerance))
    delta = float("inf")
    convergence_history = {"iterations": [], "deltas": []}

    while delta > args.tol and step < args.max_iters:
        delta = jacobi_step(T_old, T_new, fixed)
        T_old, T_new = T_new, T_old # swap buffers
        step += 1

        # Track convergence history
        convergence_history["iterations"].append(step)
        convergence_history["deltas"].append(delta)

        # Output cadence like the slides' "t=0,5,10,20,..."
        if step % args.output_every == 0 or delta <= args.tol:
            if not args.no_vtk:
                vti_path = out_dir / f"step_{step:05d}.vti"
                write_vti_image_data(str(vti_path), T_old, dx, dy, name="Temperature")
                vtk_paths.append(str(vti_path)) # keep the list; it now holds .vti files
                times.append(float(step))
                if args.also_csv:
                    write_csv_snapshot(str(out_dir / f"step_{step:05d}.csv"), T_old)
                print(f"[iter {step:6d}] delta={delta:.6e} -> wrote {vti_path.name}")
            else:
                print(f"[iter {step:6d}] delta={delta:.6e}")
            if args.gif:
                extent = (0, W, 0, H)
                frames.append(create_temperature_frame(T_old, step, delta, T_BOTTOM, T_HOT, extent=extent))

    # PVD collection for ParaView time series
    if not args.no_vtk and len(vtk_paths) > 0:
        write_pvd_collection(str(out_dir / "series.pvd"), vtk_paths, times)

    # Save GIF if requested
    if args.gif and len(frames) > 0:
        gif_path = out_dir / "temperature_evolution.gif"
        print(f"\nSaving GIF animation with {len(frames)} frames...")

        # Use PIL to save GIF
        from PIL import Image
        pil_frames = [Image.fromarray(frame) for frame in frames]
        duration_ms = int(1000 / args.gif_fps)
        pil_frames[0].save(
            str(gif_path),
            save_all=True,
            append_images=pil_frames[1:],
            duration=duration_ms,
            loop=0
        )
        print(f"GIF saved to: {gif_path}")

    print(f"\nDone. Final iter={step}, delta={delta:.6e}")
    if not args.no_vtk:
        print(f"Open in ParaView: {out_dir/'series.pvd'} (time series)")
        print("Tip: add 'Annotate Time' for the iteration number, and show the scalar bar.")
    if args.gif:
        print(f"View animation: {out_dir/'temperature_evolution.gif'}")

    # Return data if called programmatically
    if return_data:
        return {
            "convergence_history": convergence_history,
            "final_grid": T_old,
            "final_iter": step,
            "final_delta": delta,
            "hot_fraction": args.hot_fraction,
            "nx": args.nx,
            "ny": args.ny,
            "frames": frames if args.gif else None
        }

if __name__ == "__main__":
    main()
```