



GitHub repository

[Click here](#)

TL;DR:

Heat Conduction Simulation Tutorial This tutorial walks through implementing heat conduction simulations, starting with basic vanilla C implementations and progressing to more powerful Basilisk C versions. We build from 1D steady-state problems to 2D, annular, and axisymmetric geometries, demonstrating how Basilisk's domain-specific language dramatically simplifies code while adding powerful features like adaptive mesh refinement and embedded boundaries. The tutorial includes eight progressive simulation cases, showing how the same physics can be implemented with increasingly sophisticated numerical methods, and concludes with custom post-processing tools for data extraction and visualization. Basilisk ultimately reduces hundreds of lines of vanilla C code to concise, readable implementations that handle complex geometries with minimal effort.

Introduction

This guide walks you through implementing heat conduction simulations using both vanilla C and the Basilisk framework. We'll progress from simple steady-state problems to more complex transient simulations in multiple dimensions, highlighting the advantages of Basilisk for computational fluid dynamics (CFD) and heat transfer applications.

Learning Objectives

- Implement basic heat conduction solvers in vanilla C
- Transition to equivalent implementations in Basilisk C
- Understand the advantages of Basilisk's domain-specific language
- Develop simulations with increasing complexity (1D → 2D → axisymmetric)
- Appreciate how Basilisk simplifies numerical methods implementation
- Learn to analyze and visualize simulation results with post-processing tools

Throughout this tutorial, we'll follow a logical progression across eight core simulation cases:

1. Steady-state heat conduction in vanilla C (1-conduction-simple.c)
2. Transient heat conduction in vanilla C (1-conduction-transient.c)
3. Steady-state heat conduction in Basilisk (1-conduction-simple-basilisk.c)
4. Transient heat conduction in Basilisk (fill-in exercise) (1-conduction-transient-basilisk.c)
5. Enhanced transient solution with diffusion module (1-conduction-transient-basilisk-withHeaders.c)
6. 2D heat conduction (1-conduction-2D.c)
7. Heat conduction in an annulus (1-conduction-2D-annulus.c)
8. Axisymmetric heat conduction (1-conduction-Axi.c)

After completing the simulations, we'll also cover how to post-process and visualize the results using custom tools.

Part 1: Heat Conduction in Vanilla C

1.1 Steady-State Heat Conduction(1-conduction-simple.c)

We'll begin with a fundamental 1D steady-state heat conduction problem in vanilla C.

 **Problem Statement** Solve the steady-state heat equation: $\frac{d^2 T}{dx^2} = 0$ Subject to Dirichlet boundary conditions:

- $T(0) = 0$
- $T(1) = 1$

The analytical solution is the linear profile $T(x) = x$.

Let's examine the key elements of this implementation:

```

1 // Domain discretization
2 #define N      11      // Number of grid points
3 #define MAX_ITER 10000 // Maximum number of iterations
4 #define TOL     1e-10   // Convergence tolerance
5
6 // Core iterative solver using Gauss-Seidel method
7 double update_interior_points(double t_current[N], double t_new[N]) {
8     int i;
9     double error = 0.0;
10    double diff;
11
12    for (i = 1; i < N - 1; i++) {
13        // Central difference scheme
14        t_new[i] = 0.5 * (t_current[i - 1] + t_current[i + 1]);
15
16        // Track maximum error for convergence check
17        diff = fabs(t_new[i] - t_current[i]);
18        if (diff > error) {
19            error = diff;
20        }
21    }
22
23    return error;
24 }

```

This implementation uses the Gauss-Seidel iterative method with a central difference approximation for the second derivative:

$$T_i \approx \frac{T_{i-1} + T_{i+1}}{2}$$

Key Implementation Details

- Arrays for storing current and new temperature values
- Explicit boundary condition application
- Manual iteration until convergence
- Manual memory management
- Explicit file I/O for results

1.2 Transient Heat Conduction (1-conduction-transient.c)

Next, we'll look at a 1D transient heat conduction problem in vanilla C.

 **Problem Statement** Solve the transient heat equation: $\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2}$ With no-flux boundary conditions and an initial condition approximating a Dirac delta function.

The analytical solution is a Gaussian profile that spreads over time: $T(x, t) = \frac{1}{2\sqrt{\pi t}} e^{-x^2/4t}$

Key aspects of this implementation:

```

1 // Compute heat fluxes at cell interfaces
2 void compute_fluxes(double *temperature, double *flux, double dx) {
3     flux[0] = 0.0; // Left boundary (no flux)
4     flux[N] = 0.0; // Right boundary (no flux)
5
6     for (int i = 0; i < N - 1; i++) {
7         flux[i+1] = - (temperature[i+1] - temperature[i]) / dx;
8     }
9 }
10
11 // Update temperature field based on fluxes
12 void update_temperature(double *t_current, double *t_new, double *flux,
13                         double dx, double dt) {
14     for (int i = 0; i < N; i++) {
15         double dq;
16
17         if (i == 0) {
18             // Leftmost cell: flux difference is  $q[0.5] - q[-0.5]$ , but  $q[-0.5]=0$ 
19             dq = flux[1];
20         } else if (i == N-1) {
21             // Rightmost cell: flux difference is  $q[N-0.5] - q[N-1.5]$ , but  $q[N+0.5]=0$ 
22             dq = - flux[N-1];
23         } else {
24             dq = flux[i+1] - flux[i];
25         }
26
27         t_new[i] = t_current[i] - (dt/dx) * dq;
28     }
29 }

```

Key Implementation Details

- Explicit time-stepping scheme (forward Euler)
- Manual computation of fluxes at cell interfaces
- Explicit handling of boundary conditions
- Stability condition for time step ($dt \leq dx^2/2$)
- Manual memory management and array allocation
- Explicit file I/O for snapshots and results

Exercise 1: Modify the Vanilla C Code

Suggested Tasks

1. Modify `1-conduction-simple.c` to solve a 2D steady-state problem
 - Add a second spatial dimension to the grid
 - Update the Gauss-Seidel iteration for 2D
 - Modify the boundary conditions for all four edges
2. Modify `1-conduction-transient.c` to use a different initial condition
 - Replace the Dirac delta with a Gaussian profile: $T(x, 0) = e^{-x^2/\sigma^2}$
 - Compare the numerical and analytical solutions at different times

Part 2: Introduction to Basilisk C

2.1 Steady-State Heat Conduction in Basilisk (`1-conduction-simple-basilisk.c`)

Now let's examine the equivalent steady-state problem implemented in Basilisk C.

```

1  #include "grid/cartesian1D.h"
2  #include "run.h"
3
4  // Declare scalar field for temperature
5  scalar T[];
6  T[left] = dirichlet(0.0);
7  T[right] = dirichlet(1.0);
8
9  int main() {
10     // Domain setup
11     L0 = 1.0;      // Domain length
12     X0 = 0.0;      // Left boundary
13     N = 200;       // Number of cells
14
15     // Set the timestep based on stability criterion
16     DT = (L0/N)*(L0/N)/2;
17
18     // Create output directory
19     char comm[80];
20     sprintf(comm, "mkdir -p intermediate");
21     system(comm);
22
23     // Run simulation
24     run();
25 }
26
27 // Time integration using explicit finite volume method
28 event marching (i++) {
29     foreach() {
30         // Proper heat equation time stepping
31         T[] += DT*(T[1] - 2*T[] + T[-1])/(Delta*Delta);
32     }
33 }

```

Basilisk Advantages

- Simplified domain and grid setup
- Declarative boundary conditions
- Event-based programming model
- Intuitive stencil operations with `T[1]` and `T[-1]`
- Automatic handling of grid traversal with `foreach()`
- Significantly reduced code length with the same functionality

2.2 Understanding Basilisk's Event System

Basilisk uses an event-based programming model that separates concerns:

- `event init (t = 0)` : Initialization at $t=0$
- `event marching/integration (i++)` : Time integration at each step
- `event end (t = end)` : Final operations at the end of the simulation

Events can also be scheduled at specific times or intervals:

```
1 // Execute every 0.1 time units
2 event writingFiles (t = 0.0; t += 0.1; t < 1.0) {
3     // Save output
4 }
```

Event Scheduling Syntax The event scheduling syntax in Basilisk follows this pattern:

```
1 event name (condition) { ... }
```

Where condition can be:

- `t = 0` - Single execution at $t=0$
- `i++` - Every iteration
- `t += tsnap` - At regular time intervals
- `t = end` - At the end of simulation
- Combinations like `t = 0; t += 0.1; t < 1.0` for repeated execution

Exercise 2: Understand Basilisk Code Structure

Tasks

1. Compare the structure and length of `1-conduction-simple.c` and `1-conduction-simple-basilisk.c`
 - How many lines of code are in each?
 - Which sections of code are simplified or eliminated in the Basilisk version?
2. Identify the advantages of the Basilisk implementation
 - List at least three specific advantages
 - For each advantage, explain how it simplifies code development or maintenance
3. Modify `1-conduction-simple-basilisk.c` to output the relative error
 - Add code to calculate and print the error relative to the analytical solution
 - Use the formula: $\text{error} = |T_{\text{numerical}} - T_{\text{analytical}}| / |T_{\text{analytical}}|$

Part 3: Transient Heat Conduction in Basilisk

3.1 Fill-in-the-Blanks Exercise (1-conduction-transient-basilisk.c)

Let's implement the transient heat equation in Basilisk. Here's a template with some parts for you to fill in:

```

1  #include "grid/cartesian1D.h"
2  #include "run.h"
3
4  // Declare scalar field for temperature
5  scalar T[];
6
7  // Simulation parameters
8  #define EPS 0.1 // Width of initial temperature peak
9  #define tmax 1.0
10 #define tsnap 0.1
11
12 int main() {
13     // Domain setup
14     L0 = 10.0; // Domain length
15     X0 = -L0/2; // Left boundary
16     N = 200; // Number of cells
17
18     // Set the timestep based on stability criterion (CFL condition)
19     // dt = dx^2/2 for explicit scheme
20     DT = (L0/N)*(L0/N)/2;
21
22     // Create output directory
23     char comm[80];
24     sprintf(comm, "mkdir -p intermediate");
25     system(comm);
26
27     // Run simulation
28     run();
29 }
30
31 /
32 * Initialize temperature field
33 *
34 * Sets up a "Dirac delta" approximated by a thin rectangle
35 * centered at x=0 with total integral = 1.
36 */
37 event init (t = 0) {
38     foreach()
39         T[] = (fabs(x) < EPS) ? 1.0/EPS/2.0 : 0.0;

```

```

40 }
41
42 /
43 * Time integration using explicit finite volume method
44 */
45 event integration (i++) {
46     // Get timestep for this iteration
47     double dt = dtnext(DT);
48
49     // Compute fluxes at the faces of the cells
50     //  $q_i = -(T_i - T_{i-1})/\Delta$ 
51     scalar q[];
52     foreach()
53         q[] = -(T[] - T[-1])/Delta;
54
55     // Update temperature
56     //  $dT/dt = -(q_{i+1} - q_i)/\Delta$ 
57     // this enforces the central difference scheme
58     scalar dT[];
59     foreach()
60         dT[] = -(q[1] - q[])/Delta;
61
62     // Explicit Euler step
63     foreach()
64         T[] += dt*dT[];
65 }
66
67 /
68 * Save snapshots at regular intervals
69 */
70 event writingFiles (t += tsnap; t < tmax+tsnap) {
71     char filename[100];
72     sprintf(filename, "intermediate/snapshot-%5.4f.csv", t);
73
74     FILE *fp = fopen(filename, "w");
75     foreach() {
76         fprintf(fp, "%g,%g\n", x, T[]);
77     }
78     fclose(fp);
79 }

```

```

80
81 /
82  * Save final results and comparison with analytical solution
83 */
84 event end (t = end) {
85     char filename[100];
86     sprintf(filename, "conduction-transient.csv");
87
88     FILE *fp = fopen(filename, "w");
89     foreach() {
90         fprintf(fp, "%g,%g\n", x, T[]);
91     }
92     fclose(fp);
93 }

```

 **Stability Condition (K value)** The stability condition for the 1D heat equation with explicit Euler time-stepping requires: $dt \leq \frac{dx^2}{2}$ So K = 2 is the appropriate value to ensure numerical stability.

 **Understanding The Heat Equation Discretization** The heat equation $\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2}$ can be discretized in two steps:

1. Compute heat fluxes at cell interfaces: $q_{i+1/2} = -\frac{T_{i+1}-T_i}{\Delta x}$
2. Update temperature based on flux divergence: $\frac{dT_i}{dt} = -\frac{q_{i+1/2}-q_{i-1/2}}{\Delta x}$

In Basilisk's staggered grid, $q[]$ is at cell faces while $T[]$ is at cell centers.

Part 4: Enhanced Transient Solution with Diffusion Module

Now let's look at an enhanced implementation using Basilisk's built-in diffusion module.

```

1  #include "grid/multigrid1D.h"  /* Multigrid solver is required by diffusion.h */
2  #include "run.h"
3  #include "diffusion.h"
4
5  // Declare scalar field for temperature
6  scalar T[];
7  // Boundary conditions
8  // The diffusion solver will use homogeneous Neumann conditions by default
9  T[left] = neumann(0.);
10 T[right] = neumann(0.);
11
12 // Simulation parameters
13 #define EPS 0.1  // Width of initial temperature peak
14 #define tmax 1.0
15 #define tsnap 0.1
16
17 int main() {
18     // Domain setup
19     L0 = 10.0;    // Domain length
20     X0 = -L0/2;   // Left boundary
21     N = 10000;    // Number of cells
22
23     // We can use a larger timestep with the implicit solver
24     // compared to the explicit version which requires  $dt = dx^2/2$ 
25     DT = 0.01;
26
27     // Create output directory
28     char comm[80];
29     sprintf(comm, "mkdir -p intermediate");
30     system(comm);
31
32     // Run simulation
33     run();
34 }
35
36 /
37 * Initialize temperature field
38 */
39 event init (t = 0) {

```

```

40  foreach()
41      T[] = (fabs(x) < EPS) ? 1.0/EPS/2.0 : 0.0;
42  }
43
44  /
45  * Time integration using implicit diffusion solver
46  */
47  event integration (i++) {
48      // Get timestep for this iteration
49      double dt = dtnext(DT);
50
51      // Use the diffusion() function from diffusion.h to solve the equation
52      diffusion(T, dt);
53  }

```

Advantages of Using diffusion.h

- Implicit time stepping allows for much larger timesteps
- Unconditional stability removes the CFL restriction
- Higher accuracy due to careful implementation of boundary conditions
- Multigrid acceleration for faster convergence
- Adaptive mesh refinement compatible
- Handles complex geometries through embedded boundaries

Part 5: Advanced Heat Conduction Problems

5.1 2D Heat Conduction (1-conduction-2D.c)

Now, let's extend our simulations to two dimensions.

 **Problem Statement** Solve the 2D transient heat equation: $\frac{\partial T}{\partial t} = \nabla^2 T$ With Dirichlet boundary conditions:

- $T = 1$ on the top boundary
- $T = 0$ on all other boundaries

Key aspects of the 2D implementation:

```
1  #include "run.h"
2  #include "diffusion.h"
3
4  // Declare scalar field for temperature
5  scalar T[];
6
7  // Boundary conditions
8  T[top] = dirichlet(1.);
9  T[bottom] = dirichlet(0.);
10 T[left] = dirichlet(0.);
11 T[right] = dirichlet(0.);
12
13 int main() {
14     // Domain setup
15     L0 = 10.0;      // Domain length
16     X0 = -L0/2;     // Left boundary
17     Y0 = -L0/2;     // Bottom boundary
18     init_grid (1 << 7); // 128x128 grid (2^7)
19
20     // Larger timestep with implicit solver
21     DT = 0.01;
22
23     // Run simulation
24     run();
25 }
26
27 event integration (i++) {
28     double dt = dtnext(DT);
29     mgd = diffusion(T, dt);
30 }
31
32 event adapt(i++){
33     adapt_wavelet ((scalar *){T},
34         (double[]){1e-4},
35         10);
36 }
```

2D Implementation Notes

- Similar structure to 1D, but now using a 2D grid
- Uses `init_grid(1 << 7)` for a 128×128 grid
- Boundary conditions for all four sides
- Implicit diffusion solver works in any dimension
- Added adaptive mesh refinement with `adapt_wavelet()`

5.2 Heat Conduction in an Annulus (1-conduction-2D-annulus.c)

Let's look at a more complex geometry: heat conduction in an annular domain.

 **Annulus Problem** Solve the transient heat equation in an annular domain: $\frac{\partial T}{\partial t} = \nabla^2 T$ With Dirichlet boundary conditions:

- $T = 1$ on inner boundary ($r = 1$)
- $T = 0$ on outer boundary ($r = 4$)

```

1  #include "run.h"
2  #include "embed.h"
3  #include "diffusion.h"
4
5  scalar T[];
6
7  // Define inner and outer radii
8  #define INNER_RADIUS 1.0
9  #define OUTER_RADIUS 4.0
10
11 event init (t = 0) {
12     // Define annular geometry using embedded boundaries
13     solid (cs, fs, difference (sq(OUTER_RADIUS) - sq(x) - sq(y),
14                               sq(INNER_RADIUS) - sq(x) - sq(y)));
15
16     // Set boundary conditions
17     T[embed] = dirichlet (sq(x) + sq(y) < sq(INNER_RADIUS + 0.1) ? 1.0 : 0.0);
18 }
19
20 event integration (i++) {
21     // Get timestep for this iteration
22     double dt = dtnext(DT);
23
24     // Define diffusion coefficient field (constant = 1.0)
25     face vector D[];
26     foreach_face()
27         D.x[] = fs.x[] > 0. ? 1.0 : 0.0; // Diffusion only in fluid cells
28
29     // Solve the diffusion equation while respecting embedded boundaries
30     mgd = diffusion (T, dt, D);
31 }

```

Embedded Boundaries

- `embed.h` provides tools for defining complex geometries
- `solid()` creates the embedded boundary using a level-set function
- `difference()` creates the annular region by subtracting two circles
- Boundary conditions are applied using `T[embed]`
- `fs.x[]` restricts diffusion to the fluid domain

5.3 Axisymmetric Heat Conduction (1-conduction-Axi.c)

Finally, let's look at an axisymmetric problem.

 **Axisymmetric Problem** Solve the heat equation in cylindrical coordinates with axisymmetry:

$$\frac{\partial T}{\partial t} = \frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial T}{\partial r} \right) + \frac{\partial^2 T}{\partial z^2} \text{ With boundary conditions:}$$

- Gaussian heat flux at $r=0$ (left boundary)
- $T = 0$ on top and right boundaries

```

1  #include "axi.h"
2  #include "run.h"
3  #include "diffusion.h"
4
5  scalar T[];
6
7  int main() {
8      // Domain setup
9      L0 = 10.0;      // Domain length
10     X0 = -L0/2;     // Left boundary
11     init_grid (1 << 7);      // Number of cells
12
13     // Larger timestep with implicit solver
14     DT = 0.01;
15
16     // Boundary conditions
17     T[top] = dirichlet(0.0);
18     T[left] = neumann(exp(-y*y)); // Gaussian flux profile
19     T[right] = dirichlet(0.);
20
21     // Run simulation
22     run();
23 }
24
25 event integration (i++) {
26     double dt = dtnext(DT);
27
28     face vector D[];
29     foreach_face()
30         D.x[] = fm.x[]; // Use metric terms for axisymmetric problems
31
32     mgd = diffusion(T, dt, D);
33 }

```

Axisymmetric Features

- `axi.h` handles the axisymmetric formulation automatically
- In axisymmetric coordinates, `y` represents the radial direction (r)
- `fm.x[]` includes the metric terms for the axisymmetric Laplacian
- Neumann boundary with Gaussian profile at $r=0$ (left boundary)
- All the complexity of cylindrical coordinates is handled by Basilisk

Exercise 4: Advanced Heat Conduction Problems

Tasks

1. Modify the 2D heat conduction problem
 - Change boundary conditions to have a sinusoidal temperature on top: `T[top] = dirichlet(sin( $\pi$ *x/L0))`
 - Compare the steady-state solution with the new boundary condition
2. Experiment with the annulus problem
 - Try different inner and outer radius values
 - Implement a time-dependent boundary condition on the inner boundary
3. Extend the axisymmetric problem
 - Add a heat source term using `source(T, dt)`
 - Visualize the results with different flux profiles

Part 6: Post-Processing and Visualization

After running simulations, we need tools to extract and visualize the results. We'll use two key tools:

1. `getData-generic-heatEq.c` - A Basilisk program to extract data from simulation snapshots
2. `Video-generic-heatEq.py` - A Python script to create visualizations

6.1 Extracting Data with `getData-generic-heatEq.c`

This Basilisk utility reads simulation snapshots and extracts field data for visualization:

```

1 // getData-generic-heatEq.c
2 #include "utils.h"
3 #include "output.h"
4
5 scalar T[];
6 scalar cs[] = {1};
7
8 char filename[80];
9 int nx, ny, len;
10 double xmin, ymin, xmax, ymax, Deltax, Deltay;
11
12 scalar * list = NULL;
13
14 int main(int a, char const *arguments[])
15 {
16     // Parse command line arguments
17     sprintf (filename, "%s", arguments[1]); // Snapshot file
18     xmin = atof(arguments[2]); ymin = atof(arguments[3]); // Domain min
19     xmax = atof(arguments[4]); ymax = atof(arguments[5]); // Domain max
20     ny = atoi(arguments[6]); // Number of points in y
21
22     // Add fields to extract
23     list = list_add (list, T);
24     list = list_add (list, cs);
25
26     // Restore the snapshot
27     restore (file = filename);
28
29     // Calculate grid parameters
30     FILE * fp = ferr;
31     Deltay = (double)((ymax-ymin)/(ny));
32     nx = (int)((xmax - xmin)/Deltay);
33     Deltax = (double)((xmax-xmin)/(nx));
34     len = list_len(list);
35
36     // Allocate memory for field data
37     double field = (double ) matrix_new (nx, ny+1, len*sizeof(double));
38
39     // Extract data through interpolation

```

```

40  for (int i = 0; i < nx; i++) {
41      double x = Deltax*(i+1./2) + xmin;
42      for (int j = 0; j < ny; j++) {
43          double y = Deltay*(j+1./2) + ymin;
44          int k = 0;
45          for (scalar s in list){
46              field[i][len*j + k++] = interpolate (s, x, y);
47          }
48      }
49  }
50
51  // Output data to stderr (will be captured by Python script)
52  for (int i = 0; i < nx; i++) {
53      double x = Deltax*(i+1./2) + xmin;
54      for (int j = 0; j < ny; j++) {
55          double y = Deltay*(j+1./2) + ymin;
56          fprintf (fp, "%g %g", x, y);
57          int k = 0;
58          for (scalar s in list){
59              fprintf (fp, " %g", field[i][len*j + k++]);
60          }
61          fputc ('\n', fp);
62      }
63  }
64
65  // Clean up
66  fflush (fp);
67  fclose (fp);
68  matrix_free (field);
69  }

```

How `getData-generic-heatEq.c` Works

1. Reads a simulation snapshot file
2. Takes domain extents and resolution as input
3. Restores the simulation state
4. Interpolates field values onto a regular grid
5. Outputs coordinates and field values to `stderr`

This approach allows flexible post-processing of any Basilisk simulation.

6.2 Visualization with `Video-generic-heatEq.py`

The Python script processes the extracted data and creates visualizations:

```

1  # Example of key components from Video-generic-heatEq.py
2  def gettingfield(filename, zmin, zmax, rmin, rmax, nr):
3      # Run the getData utility and capture its output
4      exe = ["/getData-generic-heatEq", filename, str(zmin), str(rmin), str(zmax), str(rmax), str(nr)]
5      p = sp.Popen(exe, stdout=sp.PIPE, stderr=sp.PIPE)
6      stdout, stderr = p.communicate()
7      temp1 = stderr.decode("utf-8")
8      temp2 = temp1.split("\n")
9
10     # Parse the output into arrays
11     Rtemp, Ztemp, Ttemp, ctemp = [], [], [], []
12     for n1 in range(len(temp2)):
13         temp3 = temp2[n1].split(" ")
14         if temp3 == ['']:
15             pass
16         else:
17             Ztemp.append(float(temp3[0]))
18             Rtemp.append(float(temp3[1]))
19             Ttemp.append(float(temp3[2]))
20             ctemp.append(float(temp3[3]))
21
22     # Convert to numpy arrays and reshape
23     R = np.asarray(Rtemp)
24     Z = np.asarray(Ztemp)
25     T = np.asarray(Ttemp)
26     cs = np.asarray(ctemp)
27     nz = int(len(Z)/nr)
28
29     R.resize((nz, nr))
30     Z.resize((nz, nr))
31     T.resize((nz, nr))
32     cs.resize((nz, nr))
33
34     return R, Z, T, cs, nz
35
36 def process_timestep(ti, caseToProcess, folder, tsnap, GridsPerR, rmin, rmax, zmin, zmax, nr):
37     # Calculate time and filenames
38     t = tsnap * ti
39     place = f"{caseToProcess}/intermediate/snapshot-{t:.4f}"

```

```

40     name = f"{folder}/{int(t*1000):08d}.png"
41
42     # Extract data
43     nr = int(GridsPerR * rmax)
44     R, Z, T, cs, nz = gettingfield(place, zmin, zmax, rmin, rmax, nr)
45
46     # Create visualization
47     fig, ax = plt.subplots()
48     fig.set_size_inches(19.20, 10.80)
49
50     # Mask regions outside the fluid domain
51     T = np.ma.masked_where(cs != 1.0, T)
52
53     # Plot temperature field
54     cntrl1 = ax.imshow(T, cmap="coolwarm", interpolation='Bilinear',
55                       origin='lower', extent=[-rminp, -rmaxp, zminp, zmaxp],
56                       vmax=1.0, vmin=0.0)
57
58     # Add colorbar and save
59     cb1 = fig.add_axes([1-0.04, b, 0.03, h])
60     c1 = plt.colorbar(cntrl1, cax=cb1, orientation='vertical')
61     c1.set_label(r'$T$', fontsize=TickLabel, labelpad=5)
62
63     plt.savefig(name, bbox_inches="tight")
64     plt.close()

```

Visualization Process

1. For each timestep:

- Call `getData-generic-heatEq` to extract data
- Parse the output into NumPy arrays
- Create a plot using matplotlib
- Add appropriate colorbars and labels
- Save as an image file

2. Parallelization:

- Uses Python's multiprocessing to handle multiple timesteps simultaneously
- Significantly speeds up the visualization process for large simulations

6.3 Running the Post-Processing Pipeline

To visualize your simulation results, follow these steps:

1. Compile the data extraction utility:

```
1 basilisk1d getData-generic-heatEq.c -o getData-generic-heatEq
```

2. Run the visualization script with appropriate parameters:

```
1 python Video-generic-heatEq.py --CPUs=4 --nGFS=100 --GridsPerR=128 \  
2 --ZMAX=4.0 --RMAX=4.0 --ZMIN=-4.0 --RMIN=-4.0 --tsnap=1.0 \  
3 --caseToProcess=1-conduction-2D-annulus --folderToSave=Video
```

3. Create a video from the generated images (optional):

```
1 ffmpeg -framerate 10 -pattern_type glob -i 'Video/*.png' -c:v libx264 -pix
```

 **Customizing Visualizations** You can adjust the visualization parameters to highlight different aspects of your results:

- Change the colormap using the `cmap` parameter in `ax.imshow()`
- Adjust the value range with `vmin` and `vmax`
- Modify the plot layout or add additional subplots
- Include contour lines with `ax.contour()`

Conclusion

Throughout this tutorial, we've progressed from simple 1D steady-state heat conduction problems in vanilla C to complex geometries and dimensions using Basilisk C. The key takeaways include:

1. Basilisk significantly reduces the complexity and verbosity of numerical simulation code
2. The event-based programming model provides a clean separation of concerns
3. Basilisk's domain-specific language makes numerical methods more intuitive
4. Built-in modules like `diffusion.h` and `embed.h` provide powerful, optimized solvers
5. Extending to higher dimensions and complex geometries is straightforward
6. Post-processing tools are essential for analyzing and interpreting simulation results

Next Steps

- Explore other physical phenomena like advection-diffusion
- Implement variable diffusion coefficients (e.g., temperature-dependent)
- Combine with Navier-Stokes solvers for natural convection problems
- Add source terms for heat generation
- Experiment with more complex geometries using the embedded boundary method
- Create interactive visualizations using tools like Plotly or ParaView

References and Resources

- [Basilisk Documentation](#)
- [Embedded Boundaries in Basilisk](#)
- [Diffusion Module Documentation](#)
- [Axisymmetric Coordinates](#)
- [Adaptive Mesh Refinement](#)
- [Basilisk View for Visualization](#)

Need Help? If you encounter issues with any of the exercises, please check:

- Code syntax and structure
- Boundary condition implementation
- Event scheduling and dependencies
- Numerical stability considerations

Feel free to open an issue on the GitHub repository if you need further assistance.

Info

Thanks for reading, and feel free to dive deeper (or suggest improvements) via the [GitHub repository](#)!

☆ Metadata >

 Back to main website

[Home](#), [Team](#), [Research](#), [Github](#)