Temperature distribution in mixed conduction/convection domains: derivation by the method of finite differences

Matthew Barnard

for MEE 340
Dr. Taewoo Lee, Spring 2022

CONTENTS

Temperature distribution in mixed conduction/convection domains: derivation by the method of finite differences

Matthew Barnard

***Abstract*—We consider a prismatic bar with a thermal conductivity of k , surrounded on three sides by a constant temperature environment T_s , the fourth side contacting a fluid with convection coefficient h. Assuming steady conditions, we apply nodal analysis to approximate the temperature distribution, using finite difference equations of heat transfer to model temperature diffusion. We develop a system of linear equations describing the configuration of nodes and properties; using numerical methods to solve this system yields the approximate steady-state temperature distribution. Finally, we generalize the above process and automate the complete analysis using iterative methods.**

Introduction

ONE of the most miraculous features of mathematics is the unreasonable effectiveness of linear composition in systems modeling. Systems whose behavior is even approximately linear can generally be parameterized on some property of interest, composed into a system of transformations of that property, and solved algorithmically,[1] with the tools of linear algebra.

With the following demonstration we hope to convince you that linear modeling is not only unreasonably effective, but unreasonably straightforward to conceptualize and apply. We take a nontrivial configuration of thermophysical properties and discretize it by spatial sampling. First we discretize time by considering a steady state, i.e. at $t \to \infty$, and solving the final temperature distribution directly. Finally we discretize time in an iterative diffusion process and visualize thermal propagation.

In both these methods we emphasize the elegant simplicity of these linear models: all the analysis, data structures, and algorithms achieve one thing, which is to slide a bit of heat from one place to a nearby place along some *slope* that defines the heat transfer path. With enough places, we could connect every point in the universe with a tiny line and transfer heat across it with extreme fidelity; but it turns out we don't need even half that many lines to get interesting results.

We begin by establishing an example problem for which we develop the analytic solution.

Part 1: Problem statement

[1, Problem 4.58], but refer to this author's note on on page 5 of this document.

A long metal rod with square cross section is maintained at a constant surface temperature $T_{s,h} = 300\,^{\circ}\mathrm{C}$ on three sides; the fourth side is in contact with a fluid sink at $T_{\infty,c} = 100\,^{\circ}\mathrm{C}$. Heat transfer inside the rod occurs via conduction with $k = 2\,\mathrm{W/m{\cdot}K}$. Heat transfer across the fluid boundary is convective with $h = 10\,\mathrm{W/m^2{\cdot}K}$. Using a grid spacing of $0.2\,\mathrm{m}$, determine the midpoint temperature between the bar and the fluid per unit length of the bar.

[1] albeit with cubic time complexity, but it is a *finite number of steps*, which is worth getting excited about when considering the size of the domain of linear modeling.

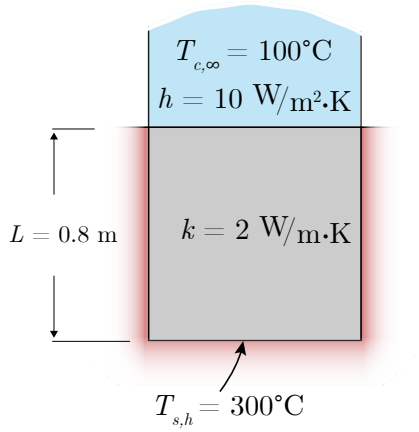


Figure 1. Three sides of the rod are on a constant-temperature boundary (gradient border) and the fourth is on a convective boundary with the fluid above. The rod is viewed in cross-section, and is assumed to be identical everywhere along its length (assumption 1).

I. Primary assumptions

1) Properties are constant in time and uniform with respect to geometry,
2) heat transfer is in a steady state,
3) materials have linear thermal behavior, and
4) I can do math (citation needed).

Because both the high $(T_{s,h})$ and low $(T_{\infty,c})$ temperatures are constant, we can assume that the system has reached a steady state with regard to heat transfer (giving us assumption 2). We therefore need only find the temperature distribution in the rod, which is a function of two dimensions and has a reasonably complex form.

II. Boundary conditions

As depicted in figure 1, we have a 2D cross section of a thermally conductive rod with four sides on two distinct heat exchange boundaries: convective exchange with a fluid at a constant temperature of $T_{\infty,c} = 100\,^{\circ}\mathrm{C}$, and a condition of constant surface temperature $T_{s,h} = 300\,^{\circ}\mathrm{C}$.

A. Constant surface temperature

The three sides of constant surface temperature have an implied heat flux to maintain that temperature. We model

the boundary as an infinite source of heat with negligible thermal resistance; we then ignore the heat flux across the boundary, and consider only the flux caused by the temperature differential $T_{s,h} - T_{\infty,c}$. Although two of the sides are identical due to symmetry, the third side on this boundary has a different conduction path to the sink; for that reason a derivation of the flux across each side is not obvious (for example, it is not simply a third of the total by energy conservation).

B. Convective plane surface

The remaining side undergoes convective heat transfer (*loss*, because $T_{s,h} > T_{\infty,c}$) with the fluid.

III. Finite-difference method

We approach the temperature distribution by considering a rectilinear network of nodal points (the *grid,* see figure 2 on the current page) across the rod's cross section; we then compute heat transfer between these nodes, using them as sensors to report the temperature values at their locations. Because the system is linear and in a steady state, the system of heat transfer equations describing the final state of all nodes will have a direct solution as a matrix equation.

We find the temperature distribution in the following broad steps:

1) Find the spanning set of linear heat equations in terms of node temperatures (section III-A),
2) write these as a matrix equation (III-B), and
3) numerically solve the temperature distribution[2] (III-C).

A. Nodal analysis and control volumes

Each node $T_{m,n}$ is at the center of a control volume, and heat flows between control volumes through their faces. Disregarding the outermost nodes whose temperature is known, this mesh forms two distinct types of control volume: internal conduction nodes with four neighbors and plane convection nodes with three neighbors.

To save space on the page we will refer to a node's neighbors using the compressed notation seen in figure 3.

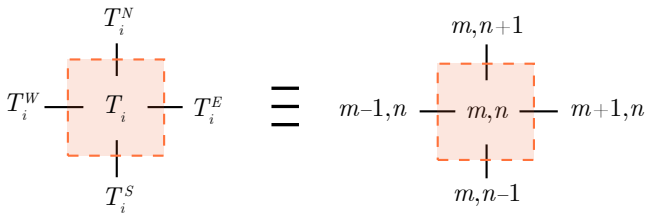


Figure 3. With $T_i \equiv T_{m,n}$, the cardinal directions refer to the neighboring nodes as shown.

[2]Numerical computation and plotting were done using MATLAB, as described in section III-C.

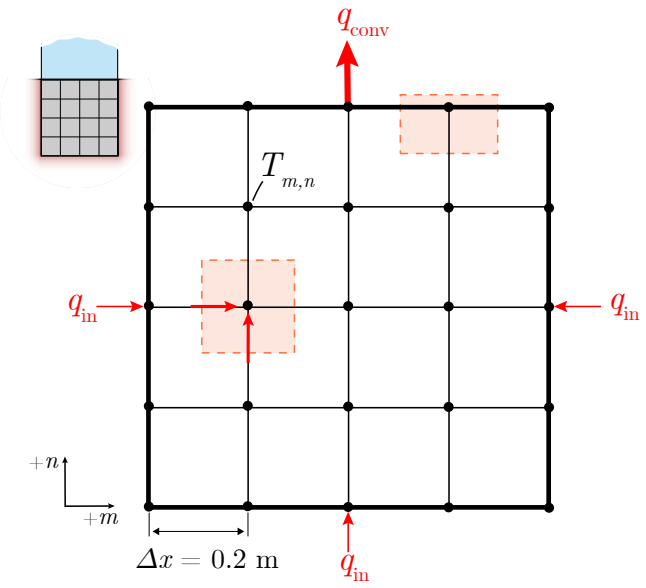


Figure 2. The cross section is overlaid with a grid of nodes evenly spaced by $\Delta x = 0.2\,\text{m}$. Each node is at the center of a control volume, and heat flows between control volumes through their faces. Temperature is the control volume property we observe as we model heat flowing. Dashed boxes outline two types of control volume: internal conduction and plane convection. The inbound heat fluxes q_{in} in this diagram are merely implied by the constant temperature condition and are not part of our analysis (see the paragraph *Constant surface temperature* on the previous page).

1) Node temperature by energy balance: Using the energy balance method, the respective heat equations for the three- and four-neighbor control volumes (CVs) at a node $T_i \equiv T_{m,n}$ are

$$T_i^E + T_i^W + 2T_i^S + 2\tfrac{h\Delta x}{k} T_{\infty,c} - 2\,T_i\left(\tfrac{h\Delta x}{k} + 2\right) = 0 \quad (1)$$

$$T_i^E + T_i^N + T_i^W + T_i^S - 4T_i = 0. \quad (2)$$

The first of these CV equations is the convection boundary; for clarity we write it without isolating T_i:

$$T_i^E + T_i^W + 2T_i^S + 2\frac{h\Delta x}{k}\left(T_{\infty,c} - T_i\right) = 4T_i. \quad (3)$$

Now we see that the temperature at the convection nodes depends on the material properties (rod conduction coefficient k and fluid convection coefficient h), the size of the control volume (Δx along the convection plane), and the temperature difference across the boundary. Consider that the fluid temperature $T_{\infty,c}$ will always be the lowest temperature anywhere, and so $T_{\infty,c} - T_i < 0$; the steady-state behavior of a convection node then is: its three neighbors contribute heat and the fluid removes that heat.

The second node equation is straightforward: at steady-state equilibrium in a linear system (assumptions 2 and 3) each internal node will be at the mean temperature of its neighbors.

2) Reducing the problem space with symmetry: Before applying the heat equations to the nodes, we can eliminate nearly half of the computation effort by recognizing that the values to the left and right of the vertical centerline are identical (see figure 4). We will compute only the left half of the field, and then mirror the results to obtain the

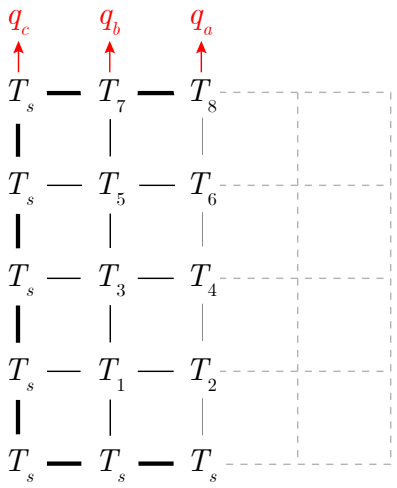


Figure 4. The nodes on the right half of the cross section are identical to those on the left half. The centerline nodes refer to the left-hand neighbor when the right-hand neighbor is called for.

complete view. At the centerline we simply refer to the left-hand neighbor twice, rather than to both the left- and right-hand neighbors.

B. Collect heat equations at each node

Numbering nodes as in figure 4, we see that nodes 1..6 are all interior conduction nodes with four neighbors, while 7 and 8 are convection nodes with three neighbors. We write the internal nodes 1, 3, and 5 with conductive behavior as in equation (2); nodes 2, 4, and 6 are written similarly, except that their left and right neighbors are identical due to symmetry.

Difference equation $(=0)$	Nodes
$T_i^E + T_i^N + T_i^W + T_i^S - 4T_i$	1,3,5
$T_i^N + 2T_i^W + T_i^S - 4T_i$	2,4,6
$T_i^E + T_i^W + 2T_i^S + 2\frac{h\Delta x}{k}T_{\infty,c} - 2T_i\left(\frac{h\Delta x}{k}+2\right)$	7
$2T_i^W + 2T_i^S + 2\frac{h\Delta x}{k}T_{\infty,c} - 2T_i\left(\frac{h\Delta x}{k}+2\right)$	8

Expanding these yields a system of eight linear equations in eight unknown temperatures:

$$
\begin{aligned}
T_2 + T_3 + 2T_s - 4T_1 &= 0\\
T_4 + 2T_1 + T_s - 4T_2 &= 0\\
T_4 + T_5 + T_s + T_1 - 4T_3 &= 0\\
T_6 + 2T_3 + T_2 - 4T_4 &= 0\\
T_6 + T_7 + T_s + T_3 - 4T_5 &= 0\\
T_8 + 2T_5 + T_4 - 4T_6 &= 0\\
T_8 + T_s + 2T_5 + 2\frac{h\Delta x}{k}T_{\infty,c} - 2\,T_7\left(\frac{h\Delta x}{k}+2\right) &= 0\\
T_7 + T_6 + \frac{h\Delta x}{k}T_{\infty,c} - \quad T_8\left(\frac{h\Delta x}{k}+2\right) &= 0.
\end{aligned}
$$

We then isolate constants, order terms by node index, and extract coefficients to write the matrix equation found in figure 5 on the following page. With the system in the form $\boldsymbol{AT} = \boldsymbol{c}$, we find the temperature distribution by finding $\boldsymbol{A}^{-1}$ and taking $\boldsymbol{T} = \boldsymbol{A}^{-1}\boldsymbol{c}$.

C. Numerical solution

The solution presented in the previous section is easily solvable with a matrices package like MATLAB.

We begin by specifying the environment and physical constants (figure 6); i.e. the geometry, material properties, boundary conditions, etc. Note that while MATLAB does have a type system, it enforces only the semantics defined by the language specification, which do not include units or physical dimensions. This leaves numerical methods highly susceptible to silently developing broken semantics and producing meaningless results. We encourage the reader to explore what kinds of mitigations might fortify this code.[3,4] This topic, as well as a number of other forms of program correctness, are as critical for discussion as any other aspect of this implementation. They are, however, outside the scope of this article; and so suffice to say that even this short work was not completed without time spent troubleshooting mismatched units.

Continuing on, in figure 7 we extract the coefficient matrices seen in figure 5, grouping and factoring wherever it improves readability. MATLAB offers a number of provisions for solving matrix equations in this standard form.[5] As non-experts in linear algebra, we prefer the `mldivide` function (operator form $\mathbf{x} = \mathbf{A}\backslash\mathbf{C}$) because it implements any of a variety of methods depending on certain properties of the input matrices.[2, section on Algorithms] This approach allows us to defer any analysis of the input, which may be sophisticated, and still handle corner cases correctly.

IV. Results

We ran the routine outlined in figures 6 and 7 in MATLAB 2021b for Windows 10.0.19044 on an eighth-generation eight-core Intel x64 processor with 32 GB 3200MHz DRAM. The setup and one unique solution are completed in $0.34 \pm 0.23\,\mathrm{ms}$ $(n = 100)$. The time complexity of the solution depends on the implementation chosen by `mldivide`, but is generally $\mathrm{O}\left(n^3\right)$, so our low time cost depends heavily on the small number of nodes.

Taking the resultant vector `T_K` as the solution $\mathbf{T}$ to the final node temperatures, we distribute $\mathbf{T}$ over the grid of nodes and interpret the temperature matrix as a low-resolution bitmap. Because we found a unique solution, our steady state assumption (2), which implies that a unique solution exists at $t \to \infty$, is supported.

[3] A custom structure to represent a dimensional variable can be checked for correctness at run-time, for example. MATLAB's type system enables this because a struct with a unique set of named members constructs a unique type; therefore a struct with tag and value members has necessary semantics to produce safe code involving physical quantities.

[4] The Wolfram Language standard library includes an invasive dimensional type, `Quantity`, which is an effective (if verbose) prototype of the concept. For a fluent syntactic example, consider the C++14 standard library `chrono_literals`, which allows one to strongly-type literal values (e.g. `1ms`) and ensure that the program is *statically* provable to be dimensionally correct.

[5] namely `mldivide`, `linsolve`, and by matrix inverse as $\mathbf{x} = \mathrm{inv}\left(\mathbf{A}\right) * \mathbf{b}$

$$\begin{bmatrix} -4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & -4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -4 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & -4 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -4 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 & -4 & 0 & 1 \\ 0 & 0 & 0 & 0 & 2 & 0 & -2\left(\frac{h\Delta x}{k}+2\right) & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & -\left(\frac{h\Delta x}{k}+2\right) \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \end{bmatrix} = T_s \begin{bmatrix} -2 \\ -1 \\ -1 \\ 0 \\ -1 \\ 0 \\ -1 \\ 0 \end{bmatrix} + \left(\frac{h\Delta x}{k} T_{\infty,c}\right) \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -2 \\ -1 \end{bmatrix}$$

Figure 5. The nodal heat model, a system of finite-difference equations, in the form $\boldsymbol{AT} = \boldsymbol{c}$. The solution to this system is $\boldsymbol{A}^{-1}\boldsymbol{AT} = \boldsymbol{T} = \boldsymbol{A}^{-1}\boldsymbol{c}$.

```
%% [1/3] Configuration

% Units are      meter    Kelvin     Watt

% Geometry
Dx   = 0.2;         %(m) node spacing

% Thermal properties
h    = 10;          %(W/m2.K) convection coeff.
k    = 2;           %(W/m.K)  conduction coeff.
H    = h * Dx / k   %(1) convection node coeff.

% Boundary properties
Ts   = c2k(300)     %(K) constant surface temp.
Tinf = c2k(100)     %(K) fluid sink temp.
```

Figure 6. Configuring the environment to contextualize our model; we also precompute terms wherever possible and homogenize units (MKS).

```
%% [2/3] Heat equations by finite differences

% Convection node coefficients
a7 = -2 * (H + 2)   %(1)
a8 = -1 * (H + 2)   %(1)

% Temperature coefficients A in 'At = c'
A = [
  -4  1  1  0  0  0  0  0    %(1)
   2 -4  0  1  0  0  0  0
   1  0 -4  1  1  0  0  0
   0  1  2 -4  0  1  0  0
   0  0  1  0 -4  1  1  0
   0  0  0  1  2 -4  0  1
   0  0  0  0  2  0 a7  1
   0  0  0  0  0  1  1 a8
];

% Constants (geometry, material propserties,
% boundary conditions, etc.)
C =     Ts   * [-2 -1 -1  0 -1  0 -1  0] ...
  + H * Tinf * [ 0  0  0  0  0  0 -2 -1] ;
C = C' ;

%% [3/3] Numerically solve matrix equation
T_K = A\C
```

Figure 7. The coefficient matrices of figure 5, with constants factored and isolated. Note on line 26 that the linear solution is provided by MATLAB's `mldivide` operator rather than by matrix inversion; this method generalizes to solve a diversity of problems where more specialized methods may perform better than inversion.[2, section on Algorithms]

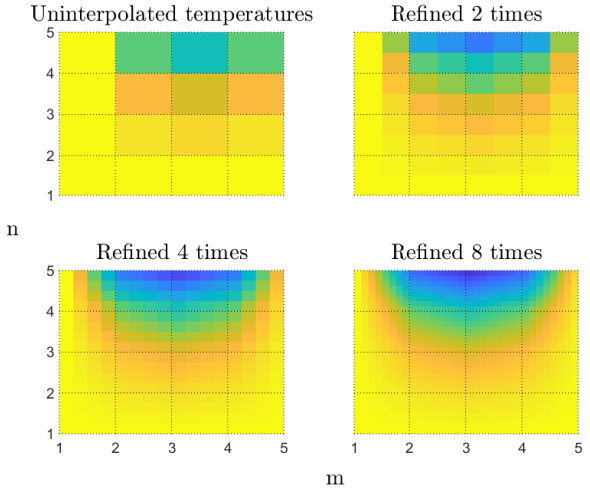


Figure 8. Surface plots of the temperature distribution at different levels of refinement by linear interpolation. Nodes are at grid crossings labeled by (m, n). Although potentially useful for building an initial intuition, these data are heavily synthesized and add a high weight to linearization errors.

A. *Visualizing the temperature distribution*

Before referring to the figures, consider the unreasonable effectiveness of linear modeling in this case: a small number of interpolated subdivisions (*refinements*) produces a nearly continuous solution from a depiction bearing very little detail. The initial result is seen top left of figure 8 on the current page, followed by interpolated visualizations.

We call these visualizations, and neglect to show a temperature scale, to emphasize that these are synthetic images of heavily approximated data. At eight levels of refinement the 25–node grid of values has been embellished by $24,000\%$—data that intentionally neglected the diffusion equations to reduce computational load. Given the linearity assumption holds, however, these may in fact be very good approximations, and so we consider them to be potentially excellent qualitative tools that require additional work to quantify with confidence.

1) Proposed heuristic to justify interpolated surface maps: Consider a test that may give quantitative value to figure 8: for interpolations to a particular level of refinement, perform a complete simulation with a grid refined to the same level, so that the two can be sampled coincidently. Sampling at nodes, take the mean squared error between the distributions. Repeat this process to obtain the mean squared errors for a diversity of geometries—the goal here being to simulate

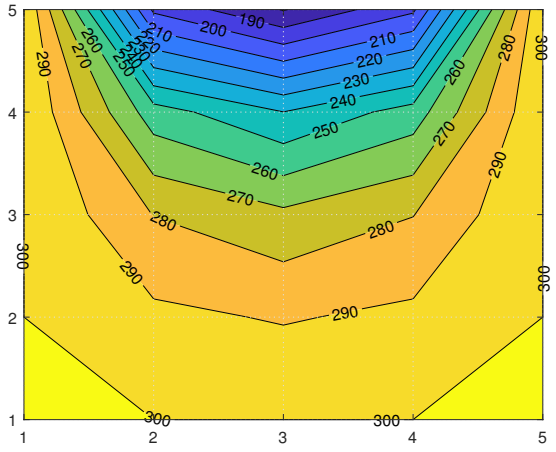


Figure 9. Isothermal contour lines in the temperature distribution; isotherm values are in degrees Celsius. This plot reveals important information about the distribution: when divided into even intervals, the areas of those intervals are equivalent to the frequency axis of a histogram. Put another way, they communicate the proportion of the total distribution represented by that temperature range. This enables us to, for example, analyze the relative temperature gradient by comparing distances between contour lines.

configurations that are similar to, but computationally less expensive than, our target problem; for example, we may test a limited subset of a domain of geometries we'd like to investigate. Quantify the differences in geometries by taking the mean squared error between their nodes' initial temperatures. Dividing the mean squared errors in temperature by those in initial conditions yields a measure of the dependence of interpolation performance on the geometric arrangement; if this value is approximately zero (by student's t-test) then we may consider extrapolating the interpolation performance to novel geometries. The magnitude of the mean squared error in temperatures dictates our confidence in the interpolation at refinement levels on the same order. Because discretization and numerical computation can introduce convergence issues, we should only conservatively interpret any subdivide-and-interpolate refinement at a resolution that has not been directly simulated.

2) Contour-mapping isotherms : Warnings about approximation notwithstanding, MATLAB provides an exceedingly useful approximated distribution plot, seen in figure 9. The suffix in the function's name, `contourf`, refers to a solid color fill between adjacent contour lines. In addition to interpolating for more smoothness and detail, the filled contour plot embeds a histogram by encoding binned frequencies from the temperature distribution in the areas of solid color. This is a marked improvement from our smoothed plots, where only a vague sense of the binned distribution can be had with any confidence.

A key value of the contour plot, in this case, is that we can intuitively grasp the temperature gradient by observing the spacing between contours. This gradient is the same one that drives the heat transfer in our system; its constant value, shown via contours, is indicative of the system's steady state (assumption 2). We explored surface plotting in three dimensions toward the same goal, with slopes in the third dimension representing the temperature gradient (top of figure 16 on page 10).

Although interactive 3D color visualizations can be powerful tools, they have limited utility when projected from a static viewpoint, potentially without color, as result plots often are. Especially when comparing datasets, the clearest projections of two sets may be from different angles so that comparing them from a common viewpoint obscures important details of one. For these reasons we suggest that the filled contour plot of figure 9 is the most broadly useful default visualization for these results.

V. Conclusion to Part 1

Finally we derive an answer to the problem from the temperature matrix $\mathbf{T}$, a value neatly approximated by a vertex on the $270\,^{\circ}\mathrm{C}$ isotherm:

$$T_{\text{midpoint}} \approx 272\,^{\circ}\mathrm{C}.$$

Author's note: *there is, of course, another part to the problem, involving the heat flux. I've chosen to reduce the discussion in this part of the report to be more congruent with Part 2, which does not evaluate the heat flux. Additionally, I do not vary the grid spacing in Part 1.*

In Part 2 we develop a method that can arbitrarily vary grid size (top of figure 15*) and can be extended to compute any value within reason, not just temperature or heat flux. My aim is to focus on that abstraction; and so when viewing Part 1 in the context of Part 2, I feel that the additional implementation details and demonstrations are simply outside the scope of the paper, and I hope to limit any possible distraction.*

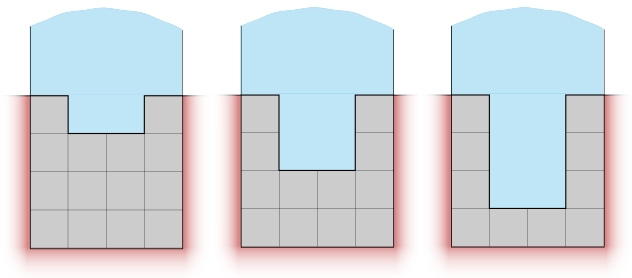


Figure 10. We may be interested in increasing the cooling throughput of this heat exchanger. With the fluid being the thermal sink, an obvious approach is to maximize the convective area. Even with a system of only 25 nodes, producing an analytic solution for every interesting configuration will be impractical.

PART 2: GENERALIZING AND AUTOMATING THE METHOD

VI. INTRODUCTION TO AUTOMATION

A. Motivation

In Part 1 we considered a particular system, analytically developed a heat diffusion model within it, and solved the model's state at $t \to \infty$. Although we developed the model in the context of a particular system, it is developed in terms of abstractions that generalize across many similar systems; therefore if we automate a solution in terms of those abstractions, we can automate solutions to all of those other systems.[6]

In figure 10 we hypothetically interpret the system in Part 1 as a heat exchanger, then propose a research question that involves repeatedly running our analysis on changing geometry. We take this as a motivating example for a value in generalizing and automating our approach.

B. Defining the general problem domain

A necessary precondition for our process is the system's regular rectilinear geometry: this allowed us to uniformly discretize the system without linearizing any nonlinear forms; for example, nowhere do we approximate a curved area by tessellating with squares.

The precondition of regular rectilinear geometry constrains all of the systems we might consider to ones of the form of

- an $m \times n$ graph of temperature values whose edges represent geometric adjacency,
- with four constant edge weights per node, determined by adjacency rules,
- where the adjacency rules target an isomorphic $(m+1)\times(n+1)$ graph of cells whose values represent thermal conditions,
- and where the value at each node is the sum of its edge-weighted neighbors.

Such a form can be trivially constructed by a sequence of operations on one or more $m \times n$ matrices.

[6]The number of *types* of systems in the domain of our method is $3^{m\times n}$ where (m, n) names a square cell identifying one of three thermal behaviors. The number of *systems* is infinite because cell properties are free.

C. Accessible abstraction via live user interaction

With the technology available to us, we have a high degree of flexibility in how we engage with an automated solver for a general kind of problem. We chose to implement this demonstration in MATLAB for a number of reasons, an important one of which being its rich set of user interaction capabilities. Given rich drawing and user input, we are able to project the general system directly onto the screen and allow the user to vary its parameters in place, creating and evaluating arbitrarily complex new systems at will.

This in-place editing paradigm lends itself to exploring problems in the way described in figure 10, where the user progressively explores an idea by making iterative changes in a way that is both experimentally-structured and creatively flexible.

D. Program design

User interactivity demands two phases of execution: computing and displaying the state of the system, and gathering input from the user to modify it. We begin then from the following sketch of program execution:

---

1) Configure the system in an initial state.
↻ 2) Compute the state of the system.
   a) Compute initial values for nodal analysis.
   b) Analyze edges and classify nodes according to which diffusion equation governs its behavior.
   c) Assign weights to node edges according to the result of *(b)*.
   ↻ d) Compute the next system state by computing a node's new value as the sum of its weighted neighbors' old values.
   e) If the system state changed significantly, *repeat from (d)*.
3) Display the system to the user.
4) Gather input from the user.
   a) If the user modified the system, *repeat from (2)*.
   b) If the user closed the program, *finish*.

Our program implements this sketch in the following broad steps. We first establish the physical environment with default initial values. Here we construct an object to represent the physical context for any procedure which computes a physical quantity:

```
%(m) cross section length on a side
Env.Geometry.L        = 0.8;
%(W/m2.K) convection coefficient
Env.Materials.fluid.h = 10;
%(W/m.K)  conduction coefficient
Env.Materials.metal.k =  2;
```

Notice that this abstraction is firmly grounded in computing a process across a square cross-sectional area. Recalling the difference equations in section III-A1, diffusion between nodes depends on the distance Δx between them. It would not be difficult to extend this implementation to support rectilinear

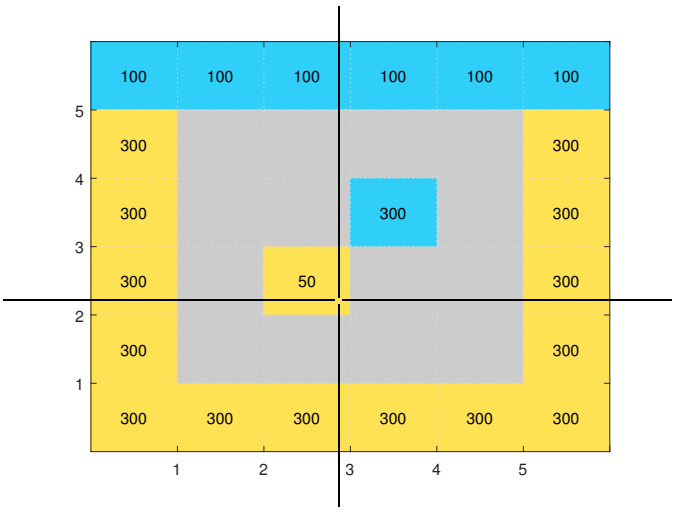


Figure 11. The field is divided into cells that represent a particular set of material properties. The user can use the mouse to cycle a cell's type or modify its constant properties. Constant-temperature cells (yellow) have a settable temperature that is given to any node touching that cell. Fluid-sink cells (blue) similarly have a settable temperature, represented by T_∞ in the convection equations. Metal conductor cells (grey) receive their temperature via heat transfer with the prior types.

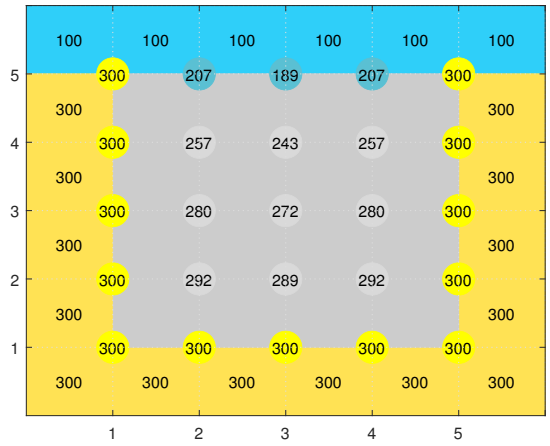


Figure 12. After a short duration without user interaction the analyzer will construct the grid of nodes and the iterator will begin diffusing temperature across it. The results of the nodal analyzer can be seen in the node colors: yellow nodes receive a constant temperature; light blue nodes have one convective neighbor, while dark blue nodes have three. Iteration finishes when the maximum global change in temperature between iterations is sufficiently small.

sections by defining Δx along axes. We simplify the implementation by computing Δx from the side length of a square section and the number of nodes on its edge.

At initialization, we also give the user the option to override the values of environmental constants. We do not, however, allow the user to vary physical properties of materials between cells, because this would violate assumption 2.

For user interaction we leverage MATLAB's plotting tools to construct a layered graphic. The simulation area (the *field*) is divided into $m \times n$ *cells*, and the cells are drawn as pixels in an $m \times n$ image. Their colors represent their types, and textual labels represent the cell value. In this case the cell value is temperature in degrees Kelvin, displayed in degrees Celsius, but we emphasize that the cell value is any computable property of the system. A more elaborate implementation may compute several values at every point, simply by adding variations on the matrices used here for temperature.

The user manipulates the system by clicking on cells: the left mouse button cycles the cell through types, and the right mouse button prompts the user to edit the cell's value.

Every session of user input initially begins by waiting for an unlimited amount of time. During this period the user can look over the system and make decisions. After the first click, we clear the previously-simulated state and enter an *editing mode* where the user can freely change cells. If a period elapses without any user input while in the editing mode, then the program exits that mode and begins simulating the system as-is. The program has now gone from step (4) to step (2) of the execution sketch.

In the following sections we will describe how the program computes the system state. We divide step (2) into two phases: *node classification* and *iterative diffusion*. In section VII-A we lay out the key data structures that define our system in software. In section VII-B we implement the diffusion equations[7] with respect to those key data structures, i.e. as coefficient matrices. Node classification is the process of determining those diffusion equation coefficients (*weights*) for every edge between nodes. In section §VIII we describe iterative diffusion, the process of computing new node temperatures from the system state using those weights. Finally, in section §IX we present an interesting example of the program's output for a system that would be nontrivial if presented in Part 1.

VII. Pass 1: Node Classification

The first analysis pass starts with a matrix of cells defining the field and results in a matrix of nodes with diffusion paths between them. We begin by defining the components of that process; then we outline how the components are initialized.

A. Nodes, edges, cells, and temperature values

a) Cells & classifying nodes: Key to interactivity is the matrix of cells which the user configures to define a system. The way that configuring cells defines the system is via node classification. A node is classified as having either constant properties (representing a boundary) or properties that depend on its edges. A node whose properties depend on its edges has weights assigned to its edges depending on the cells bordering them. When a node has been assigned either constant properties or edge weights it is considered classified and may participate in heat transfer.

b) Edges between nodes in an adjacency graph: Since the matrix of nodes is homeomorphic to the cross-sectional area, we can define adjacencies between nodes (m, n) and $(m+1, n)$, (m, n) and $(m, n+1)$, and so on. These adjacencies represent heat transfer paths, and so we capture them as

[7] See difference equations in section III-A1.

edges between nodes. Edges are implemented by a vector W of four weights at each node. Each edge weight is a sum of the weights appropriate for the two cells bordering that edge. Computing these edge weights is the primary purpose of the node classification phase.

Two cells bordering an edge between nodes may define a conduction path between those nodes; in this case the behavior of those nodes will be a superposition of this conduction path with the nodes' other paths. Our model is designed for mixed convection and conduction with constant-temperature boundaries; there are therefore nine possible ways to classify each edge based on its bordering cells (note that edge classifications are not independent).

Given an edge that exits the node (m, n) in the direction $\mathbf{u}$, and the node is surrounded by cells with centers (x, y), we classify the edge by assigning its weight W in the following way:

procedure CLASSIFYEDGE($m, n, \mathbf{u}$)
  $W \leftarrow 0$ ▷ initialize edge weight
  $\mathbf{V} \leftarrow (x, y, v)_{1\ldots4}$ ▷ get cells
  for all (x, y, v) in $\mathbf{V}$ **do**
    $\mathbf{v} \leftarrow (x - m, y - n)$
    if $\mathrm{dot}\,(\mathbf{u}, \mathbf{v}) > 0$ **then**
      $W \leftarrow W + v$
    end if
  end for
end procedure

where v is $1/2$ for conductive cells, or a number that depends on the number of convective cells.

A node is classified when all of its edge weights have been assigned.

c) The structure of a node: The matrix of nodes represents the set of points (m, n) at which we sample the system's state; as such, the minimum type that can comprise a node is a scalar value representing the temperature at that point.

In fact, representing nodes as a set of matrices, each reflecting a particular system property, has features that are computationally desirable. For one, when the type of a node is compact, a greater number of nodes can fit in the processor's fastest caches. Especially when the operations are relatively simple, as in linear systems, the overhead of retrieving nodes from RAM can easily dominate the time performance of our system; compact node types therefore have significant upside opportunity. Also, if operations on a particular value are independent of others, then the entire matrix for that value can be transferred to a parallel process with no risk of program incorrectness and no synchronization cost. This is considered a *data-oriented* design, and it can be especially suited to this type of system.

Rather than a data-oriented design, we use an *object-oriented* design for nodes. Here the type of a node is a composite of all the properties relevant to that node. The primary motivation for this is in consideration of the reader, who may be more familiar with object-oriented designs, as that is the dominant paradigm taught in universities and used in production. Ostensibly, it may benefit one in conceptualizing the system if a node's entire state is encapsulated in one mentally-portable symbol.

B. Heat diffusion equations

When classification is complete, every edge between nodes has sufficient information encoded in its coefficient to model heat transfer between any two nodes. Now, if we compute the transfer between *every* two nodes, then the result is a quasi-equilibrium state where temperature values have diffused along heat transfer paths.[8]

The computation accomplishing diffusion at each node[9] then is

$$(T_i)_{\text{next}} = \begin{bmatrix} T_i^E & T_i^N & T_i^W & T_i^S & 1 \end{bmatrix} \begin{bmatrix} W_i^{\mathrm{E}} \\ W_i^{\mathrm{N}} \\ W_i^{\mathrm{W}} \\ W_i^{\mathrm{S}} \\ W_i^{\text{const}} \end{bmatrix}. \quad (4)$$

Because this equation depends only on the current state, which is fixed, the computation is embarrassingly parallel for any number of nodes.[10]

Recall that in section §IX we justified the steady-state assumption (2) by confirming the existence of a unique state at $t \to \infty$. If we correctly implement the same system in an iterative way, then it follows that the iteration will converge to the unique end state. Following that confirmation, we would take failure to converge as a sign of program incorrectness (including numerical error, which is controlled in a correct program).

A major implication of the end state being at $t \to \infty$ is that we cannot partition the duration between start and end into a finite number of finite durations. In other words, a diffusion model that iterates through intermediate states cannot converge in a finite number of steps. With that in mind, in section §VIII we discuss a strategy for arriving at a single solution in a reasonable amount of time.

VIII. Pass 2: Iterative Diffusion

A. Convergence

With the node equations (4) producing intermediate states, we need only repeatedly and progressively take these states until one is a sufficiently near approximation of the end state. A common method for halting such a process is to assign

[8] If the steady-state assumption (2) holds, then the quasi-equilibrium states are distributed over $t = (0, \infty)$. Although we have taken a finite number of intermediate states, those states would only converge at $t \to \infty$, so there are infinitely many of them. We can partition an infinite duration into infinitely many finite durations, so the time intervals between intermediate states are knowable. Our system simply does not encode any relations involving rate, so we cannot discuss the time steps of our iterator as is.

Note that in Part 1 the assumption did hold, so a failure at this point may signify incorrect implementation.

[9] Recall the neighbor naming convention from figure 3.

[10] Nodes can be evaluated iteratively, but that can also be arbitrarily partitioned into parallel iterative processes. We have the freedom to dictate memory layouts and take arbitrary batches of nodes, which is sufficient to suggest that we can always find the optimal parallelization strategy for this problem at run-time.

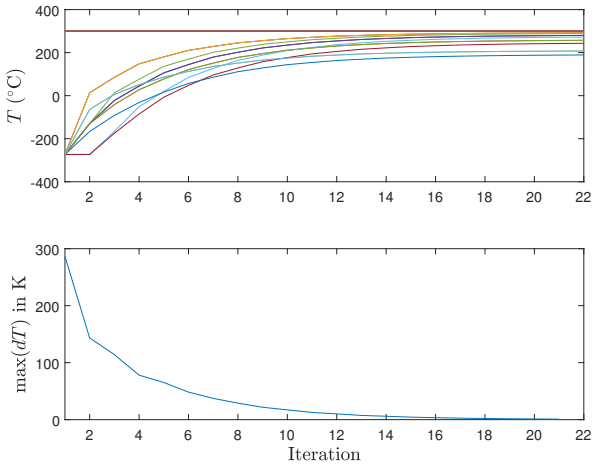


Figure 13. Plotting all node temperatures (top) shows that convergence occurs in relatively few iterations and with no oscillation (despite no effort being made to adapt step sizes or otherwise improve performance). Given that the temperatures are on the order of 10^2, we determined the distribution to be converged when the maximum temperature change is less than one Kelvin ($\sim 1\%$) per iteration (bottom).

a minimum cutoff value for the magnitude of the difference between states.

We use the minimum delta cutoff to finish iterating when the maximum change of any node is smaller than one Kelvin.[11]

In figure 13 we plot the temperatures of each node and the maximum change in temperature between iterations. Not only does the system converge, it converges in as few as $\sim 10^1$ steps.

B. Time performance

When we describe a program as *interactive* we adopt a duty to give the user a reasonably frustration-free experience. For our system, a primary source of user frustration is delay between inputs and outcomes.

In the interaction phase, the user may desire to input multiple changes in rapid succession; delays related to changing cells would therefore impede the user's freedom to express their desire at a desired rate. There is very little logic involved in effecting the user's actions, though, so that phase is trivially optimizable for a smooth experience.

In the iteration phase, our program's contract to result in a single end state means that user input is meaningless in the context of intermediate states. The entire iteration phase is therefore a delay to user input, and potential source of frustration.

For the purpose of demonstration, we choose that the intermediate node values are visible during iteration. We believe this provides the user an enhanced perspective on diffusion as an ongoing process—something that is hidden by the method in Part 1. As we see in figure 14, drawing those updates

[11]The specific value of one Kelvin represents a change on the order of 0.1% of the initial temperatures in the problem from Part 1 (figure 1 on page 1). For this reason, our program may not be suitable for configurations with temperatures lower than $-10^2\,^\circ\mathrm{C}$ because iteration will end while changes are greater than 1% of initial conditions.

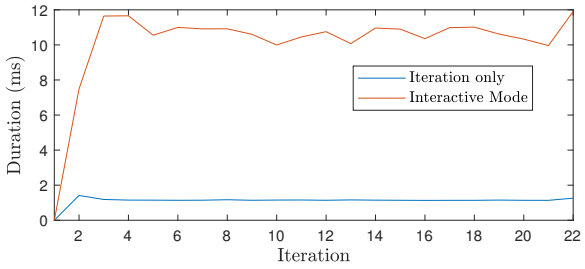


Figure 14. We evaluated the time performance of our method by timing each iteration, shown here for two configurations: the long steps (red) are produced in interactive mode, where a majority of iteration time is spent updating the user interface; the short steps (blue) are measuring iteration only, after warming up for fewer than five runs. We can see that, without graphical concerns, the solution is approximated in fewer than $50\,\mathrm{ms}$.

poses an order of magnitude time penalty to each iteration step. We face a trade-off between improving the experience with visible diffusion and harming the experience with longer delays between input.

Our solution is to accept the time cost of a visibly dynamic system, because we value that the dynamic visualization is a unique quality of an iterative implementation. At this point we can improve the user experience in a number of ways: reduce the time per iteration, reduce the number of iterations, reduce how frequently it performs iterations, etc. But the system in figure 14 is performing $\sim 10^1$ iterations at $\sim 10^1\,\mathrm{ms}$ each; we would need to reduce one of those by an order of magnitude to produce an uninterrupted user experience. We believe that simply performing iterations less often is the simplest solution, and benefits from being independent of the simulation; i.e. the user will have a controlled uninterrupted experience until simulation, no matter how long simulation takes.

The user may want to configure more than one element of their system before they care to see any result; we think this desire specifies the solution: allow the user to input multiple times in sequence before simulating their changes, but begin simulating automatically. We think a responsive design that gets out of the user's way, but also takes over control when the time is right, can provide a minimum of friction between the material and the person engaging with it.

IX. Results of Automated Analysis & Simulation

With the main program loop defined by the user interaction pattern, we draw the cells and nodes to the screen and iterate. When the program starts it determines that it has been a substantial amount of time since the last time the user clicked, which was likely never, and so it initializes the nodes with a simulation of the default state. When the iteration converges, the nodes are left visible in their last state.

Top of figure 15 on the following page we see the default state with a few modifications by the user, the program having finished iterating and begun waiting for input. We point out that the default state reflects the problem from Part 1, but the grid has been expanded from twenty-five to eighty-one nodes, a 224% increase.

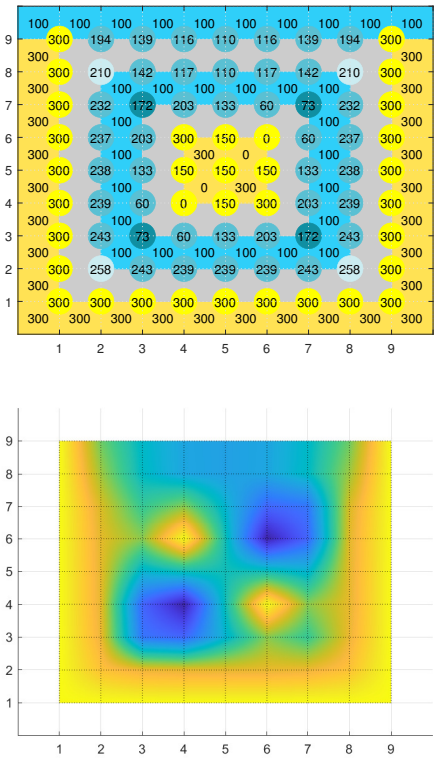


Figure 15. With our automated method it's trivial to approximate distributions on arbitrarily large grids of nodes. Here the rod is divided into segments $\approx .09\,\mathrm{m}$ long, resulting in a 2.25 times greater node density than in our original solution. Deriving this linear system analytically as we did before would involve finding equations of 81 temperatures values; put another way, our method found a total of 405 edge coefficients in fewer than 12 milliseconds.

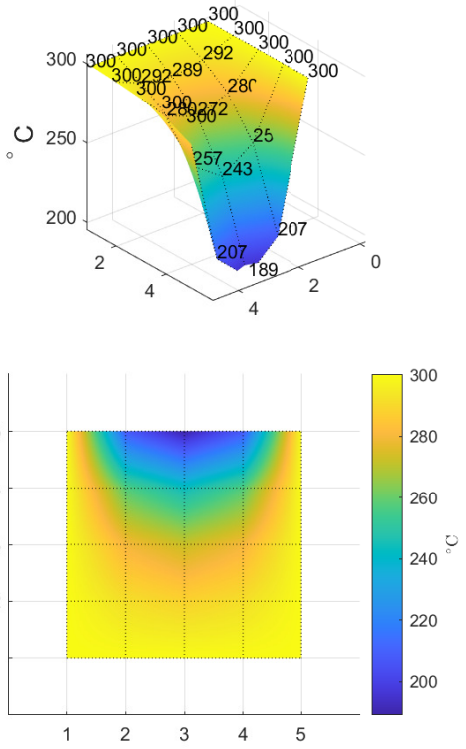


Figure 16. We finally experimented with different methods of visualizing the distribution. The surface plot in three dimensions offers a compelling view of the slope of the temperature gradient, but the view in two dimensions allows the temperature to be reasoned about in terms of the cross-section's geometry.

In fact the grid is an arbitrarily large square, but we keep the resolution coarse for a legible demonstration. We highlight that difference for two reasons:

- We propose just the increase in resolution to be sufficient motivation to carry out this work. If there is any reason to solve this problem, then we should be interested in whether the solution we find is independent of the method used. Adjusting the stride of our sampling, i.e. increasing the number of nodes, can help to reveal discretization or numerical artifacts, as well as reveal details otherwise lost in approximations. If we should perform a simulation at one level of detail, then we should perform a simulation at a variety of levels—and tools like these are the only efficient way to affect that.
- We want to restate how unreasonably effective linear modeling is: we identified components in a system, defined lines connecting them according to rules, and mapped that conceptual model nearly directly onto a computational model, with smooth high-resolution graphics. What a time to be alive.

At the bottom of figure 15 is the result of our efforts: a temperature distribution for an arbitrary configuration of elements from Part 1, produced in fractions of a second, accessible by anyone who can use a pointing device.

In section IV-A we discussed visualizing the results with area and contour plots. We also explored three-dimensional visualization; for example, in figure 16 we see that the distribution plots shown previously were orthographic projections of a surface. Although we find these plots indispensible in an interactive context, they pose a challenge to media interoperability. We believe a system like this should be primarily capable of producing visualizations at its highest quality for a static flat medium, so that the information it reveals can be maximally accessible.

X. Conclusion

We exercised the method of finite differences to construct a model of heat diffusion in a cross section. We used linearity to superimpose different diffusion functions, constructing arbitrary combinations of certain physical-effect "submodels." We generalized this process until it could be automated at high speed. Throughout our work we made only straightforward conceptual steps. We hope that our work goes a small way to demonstrating how inspiringly simple and useful building these linear worlds can be.

References

[1] T. L. Bergman and A. S. Lavine, *Fundamentals of heat and mass transfer*, 8th ed., 2017, oCLC: 1084350679.

[2] MathWorks, "Solve systems of linear equations - MATLAB mldivide." [Online]. Available: https://www.mathworks.com/help/matlab/ref/mldivide.html

~

Table of Contents

PART 1

Solving a system of equations by mldivide

```
clear;clc

% Unit conversion
c2k = @(c) c + 273.15;
k2c = @(k) k - 273.15;
```

[1/3] Environment

```
% Geometry
Dx   = 0.2;         %(m) node spacing

% Thermal properties
h    = 10;          %(W / m2.K) convection coeff.
k    = 2;           %(W / m.K)  conduction coeff.
H    = h * Dx / k   %(1) coeff. in convection

% Boundary properties
Ts   = c2k(300)     %(K) rod const. surface temp.
Tinf = c2k(100)     %(K) fluid sink temp.
```

```
H =

     1


Ts =

  573.1500


Tinf =

  373.1500
```

[2/3] Heat equations by finite differences

```
% Convection node coefficients
```

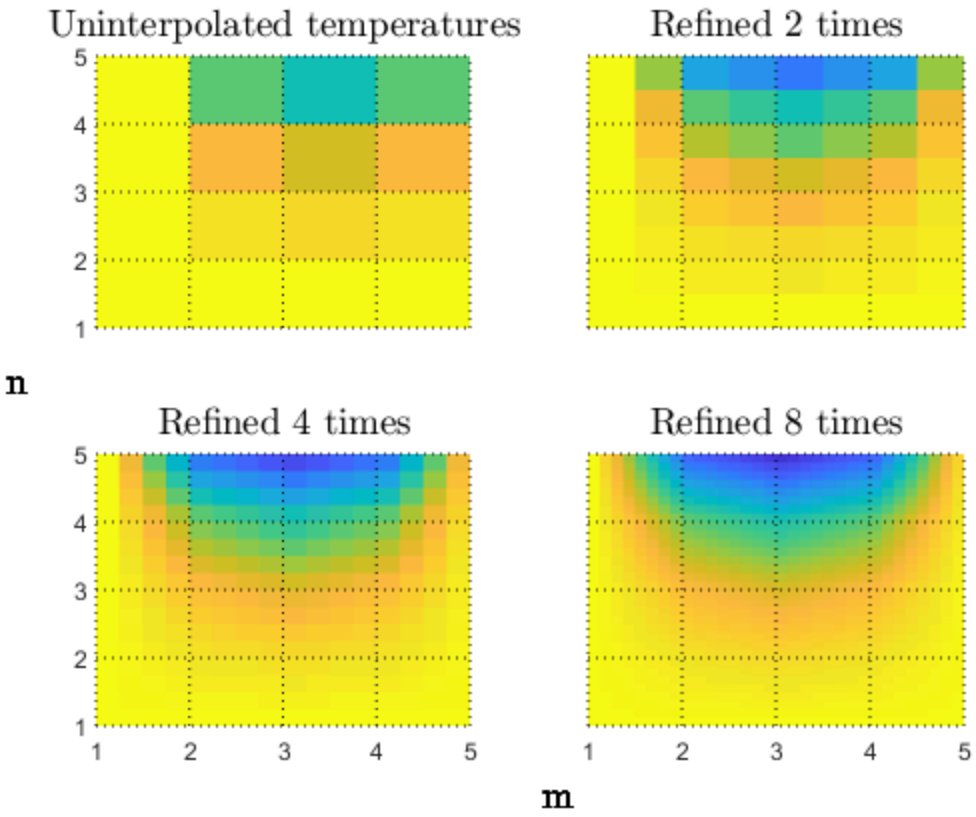
```
a7 = -2 * (H + 2)   %(1)
a8 = -1 * (H + 2)   %(1)

% Temperature coefficients A in 'At = c'
A = [
  -4  1  1  0  0  0  0  0    %(1)
   2 -4  0  1  0  0  0  0
   1  0 -4  1  1  0  0  0
   0  1  2 -4  0  1  0  0
   0  0  1  0 -4  1  1  0
   0  0  0  1  2 -4  0  1
   0  0  0  0  2  0 a7  1
   0  0  0  0  0  1  1 a8
];

% Constants (geometry, material properties, and boundary conditions)
C =     Ts   * [-2 -1 -1  0 -1  0 -1  0] ...
  + H * Tinf * [ 0  0  0  0  0  0 -2 -1];
C = C';

% Solve the temperature distribution
T_K = A\C;

% Display result
Temperatures = ...
  table(T_K, k2c(T_K), 'variablenames', {'T_Kelvin', 'T_Celsius'})
```

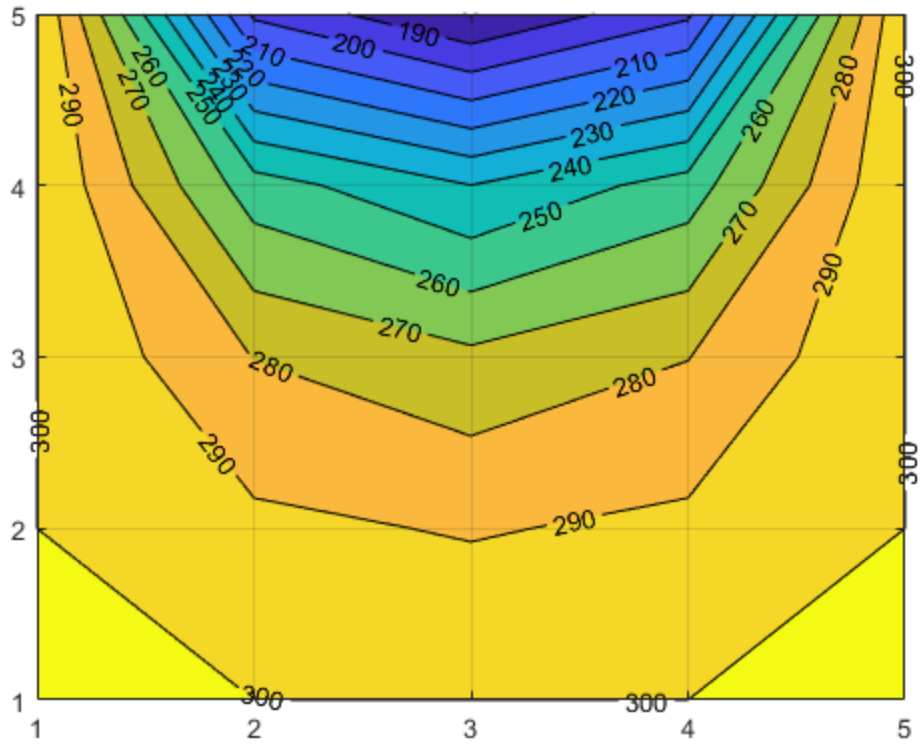
```
a7 =

    -6


a8 =

    -3


Temperatures =

  8×2 table

    T_Kelvin    T_Celsius
    ________    _________

     565.37      292.22
      562.3      289.15
     552.87      279.72
     545.33      272.18
     527.63      254.48
     513.28      240.13
     471.21      198.06
     452.55       179.4
```

[3/3] Visualization

```
% Fill the whole grid with Ts and then overwrite with the solved
% temperatures, leaving Ts on three sides.
T = Ts * ones(5,5);
% Unzip the dumb stride 2 numbering scheme we used
for i = 2:2:8
   % Orient so it plots with the fluid on top
   row         = 1+i/2;
   % Mirror to restore symmetry
   T(row, 2:4) = T_K(i-1);
   T(row, 3)   = T_K(i);
end


figure(2);

% Unrefined nodes grid
[M,N]   = meshgrid(1:5);
% Refined grid spacing
dg = 1;

for i = 1:4
    % Make a refined grid to interpolate onto
    [Mq,Nq] = meshgrid(1:dg:5);
    dg = dg / 2;

    % Interpolate onto refined grid
    Tq = interp2(M, N, T, Mq, Nq);

    % Draw interpolated temps
    subplot(2,2,i);
    s = surf(Mq,Nq,Tq);
    s.EdgeColor = 'none';
    view(2);

    % Draw grid
    hold on;
    s = surf(M,N,T);
    s.FaceColor = 'none';
    s.EdgeAlpha = 1;
    s.LineStyle = ':';
    view(2);

    % Clean up axis labels
    if i < 3;     xticklabels([]); end
    if ~mod(i,2); yticklabels([]); end

    if i == 1
        title('Uninterpolated
 temperatures', 'interpreter', 'latex', 'fontsize', 14);
    else
```

```
        title(sprintf('Refined %d times',
 2^(i-1)), 'interpreter', 'latex', 'fontsize', 14);
    end
end

text(0.5, -0.1, 'm', 'interpreter', 'latex', 'fontsize', 14);
text(-5.2, 6,'n', 'interpreter', 'latex', 'fontsize', 14);

figure(1);
% Draw contour plot with temperature values labeled
c =
 contourf(k2c(T), 'showtext', 'on', 'levelstepmode', 'manual', 'levelstep',
 10);
% Draw node grid
grid on;
set(gca, 'xtick', 1:5);
set(gca, 'ytick', 1:5);
```





Published with MATLAB® R2021a

Table of Contents

PART 2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Temperature distribution in a mixed conduction/convection domain:
%   derivation by the method of finite differences
%
% Matthew Barnard
% for MEE 340 Heat Transfer
```

```
% Spring 2022
%
% Dear reader:
%
%   In general this code is written for structural simplicity and not
 efficiency.
% Where obvious I've taken the naive approach so that execution
 follows the
% path one imagines when sketching the problem.
%   If you want to implement this kind of thing for real then my first
% suggestion is to divest yourself of any object-oriented inclinations
 and turn
% this into operations on matrices. This would be considered a classic
 example
% of a data-oriented design problem.
%   For an excellent introduction to data orientation in general,
% see Mike Acton's talk "Data-Oriented Design and C++" at CppCon 2014.
% You don't need to know C++ to see a solution to this problem in
 Mike's
% description of his problem.
%

clear;clear global;clc;

global DEBUG INFO PROMPT_PROPERTIES;
DEBUG             = false;
INFO              = true;
% If true then I'll ask you what the material properties should be.
PROMPT_PROPERTIES = false;
```

Track mouse input and cause a delayed effect on mouse up

We track the mouse click so we can effect a delayed reaction: we don't start simulating until the user has stopped clicking for a while.

```
global Input;
Input.last_click_t = uint64(0);
Input.mouse_timer  = timer;
Input.mouse_timer.StartDelay = 1;
Input.mouse_timer.TimerFcn = @pressEnter;
Input.last_m = 0;
Input.last_n = 0;
```

Cache plot objects, which are expensive to construct and destroy

```
global Handle;
Handle.surf3 = 0;
Handle.ctexts = [];
```

Define the system's physical environment

This procedural style with heavy global definitions up front is great for getting ideas working. If you're going to write code like this that you plan to share or maintain then all of these globals need to be passed as state.

```
global Env
% As taken from the original problem 4.58
Env.Geometry.L        = 0.8;  %(m) cross section length on a side

% The materials should have uniform properties for our heat equations
Env.Materials.fluid.h = 10;   %(W/m2.K) convection coefficient
Env.Materials.metal.k =  2;   %(W/m.K)  conduction coefficient
```

Show default physical constants to user and allow changing before start

True if we have a valid value

```
hb = false;
kb = false;

while PROMPT_PROPERTIES && (~hb || ~kb)
  prompt = {'Metal conduction coefficient k = ', 'Fluid convection
 coefficient h = '};
  dlgtitle = 'Material Properties';
  dims = [1 35];
  definput = {num2str(Env.Materials.metal.k),
 num2str(Env.Materials.fluid.h)};
  answer = inputdlg(prompt,dlgtitle,dims,definput);
  if length(answer) == 2
    % Accept numbers that aren't j
    % TODO: does it interpret pi?
    [k, kb] = str2num(answer{1});
    [h, hb] = str2num(answer{2});
    if ~isreal(k); kb = false; end
    if ~isreal(h); hb = false; end
    if hb; Env.Materials.fluid.h = h; end
    if kb; Env.Materials.metal.k = k; end
  end
end
```

Cells define the constant and initial properties throughout the field

For example, the interior of a cell is a uniform material with uniform thermal properties. The interior of a cell may also be a boundary condition, in which case nodes in contact with that cell will acquire that condition (or, if a node contacts multiple distinct conditions, it will aquire a mean).

```
% Cell types
global ConstantTemp Conductive Convective
```

```
ConstantTemp = 0;   % Boundary condition -- fixes a node that touches
 it.
% These types select a diffusion equation across the edge bordering
 the cell.
Conductive   = 1;   % F
Convective   = 2;   % Surface convection condition

% 1 = metal rod interior
% 2 = fluid
% Any other number = a constant surface temperature in *C
initial_cells = [%          <------- m
  300 300 300 300 300 300 300 300 300 300  %   n
  300  1   1   1   1   1   1   1   1  300  %   |
  300  1   1   1   1   1   1   1   1  300  %   |
  300  1   1   1   1   1   1   1   1  300  %   |
  300  1   1   1   1   1   1   1   1  300  %   v
  300  1   1   1   1   1   1   1   1  300
  300  1   1   1   1   1   1   1   1  300
  300  1   1   1   1   1   1   1   1  300
  300  1   1   1   1   1   1   1   1  300
   2   2   2   2   2   2   2   2   2   2
%      top of screen
];


global width_cells height_cells num_cells
width_cells  = size(initial_cells, 2);  % I can look up the order of
 axes for size every day until I die and I still won't learn it
height_cells = size(initial_cells, 1);
num_cells    = width_cells * height_cells;

global cells;
for m = 1:width_cells
  for n = 1:height_cells
    % STRUCT CELL {
    cells(n,m).m = m;
    cells(n,m).n = n;
    % Naming your variables with units is the least you can do.
    % If you're writing critical engineering software then use C++ and
 develop
    % a templated unit system so you can _statically_ prove that you
 won't
    % crash anything into mars.
    cells(n,m).T_K = CtoK(100);

    cell = initial_cells(n,m);
    if cell < 1 || cell > 2
      % A constant temp value was given as the cell type
      cells(n,m).T_K = CtoK(cell);
      cells(n,m).type = ConstantTemp;
    else
      cells(n,m).type = cell;
    end
    % } // STRUCT CELL
```

```
  end
end
```

Nodes are the simulation points where the field is evaluated

```
global width_nodes height_nodes
% Not counting the two outermost cells in each row/column,
% every two cells share a third node in the middle.
width_nodes  = (width_cells  - 2) + 1;
height_nodes = (height_cells - 2) + 1;

global num_nodes
num_nodes = width_nodes * height_nodes;

global nodes;
for m = 1:width_nodes
  for n = 1:height_nodes
    % STRUCT NODE {
    nodes(n,m).m = m;
    nodes(n,m).n = n;
    nodes(n,m).case = 0;
    nodes(n,m).T_K = 0;
    % These coefficients define the heat transfer into a node as a
 function
    % of its four neighbors, each scaled by a coefficient, and all
 summed
    % with the constant term.
    %                    E N W S +constant
    nodes(n,m).coeff = [0 0 0 0 0];
    nodes(n,m).active = true;
    nodes(n,m).hLabel = 0;
    % } // STRUCT NODE
  end
end
```

Finish setting up the geometry now that we know the refinement

```
Env.Geometry.Dx = Env.Geometry.L / width_nodes;
```

Precomputed coefficients for heat equations

```
Env.Coeff.H = Env.Materials.fluid.h * Env.Geometry.Dx
   ...
            / Env.Materials.metal.k;
```

Arrange figures on screen

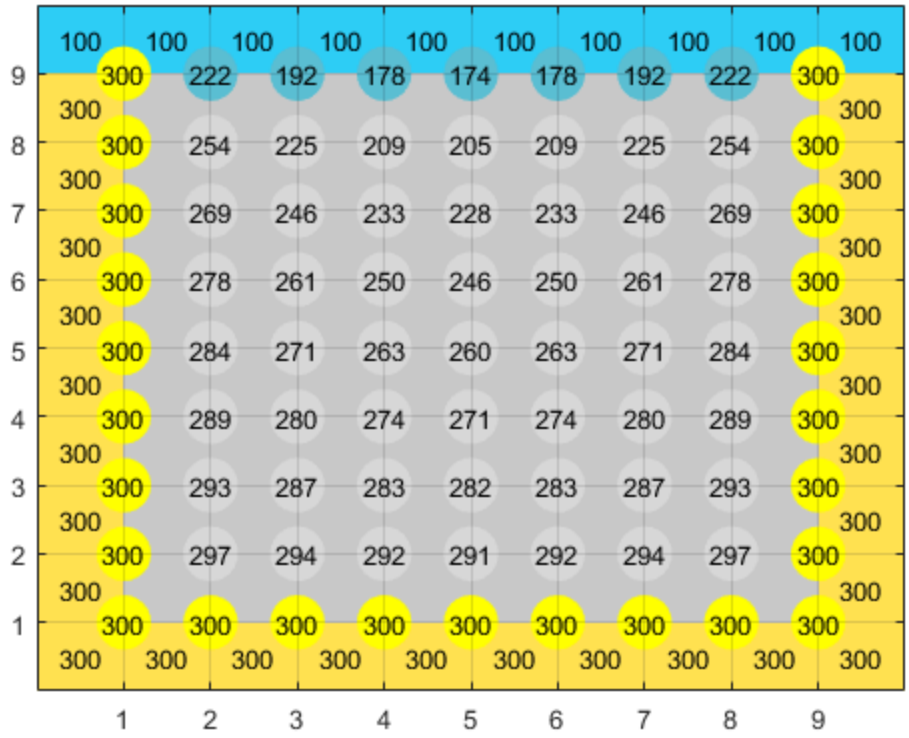
```
set(0,'units','pixels');
```

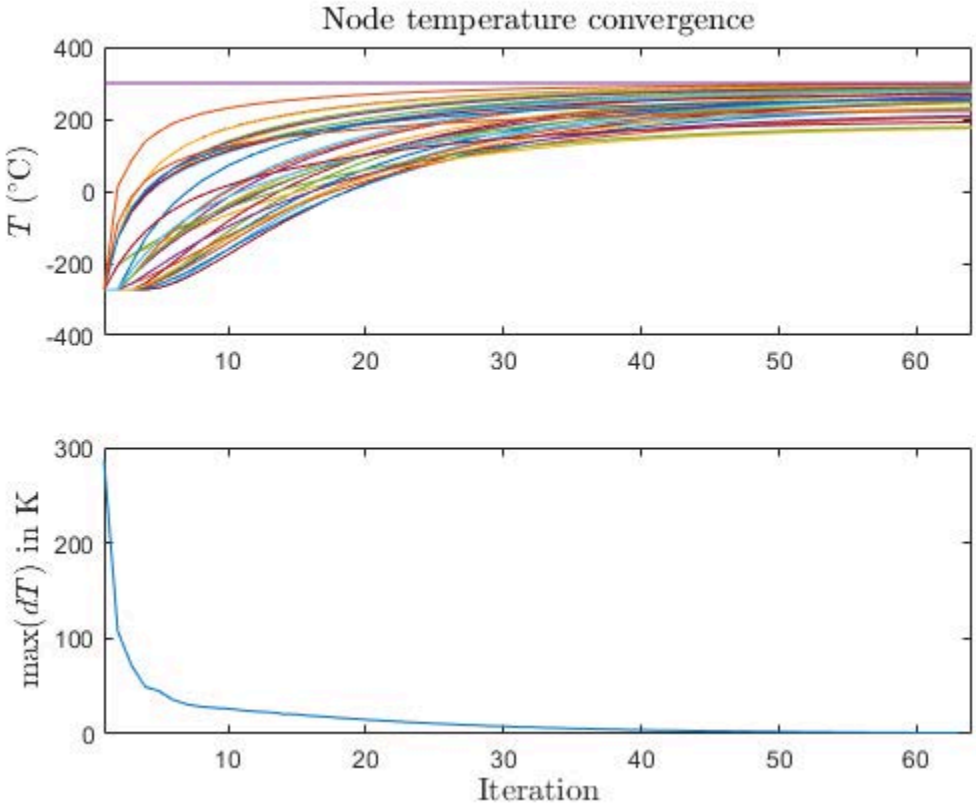
```
screen_size = get(0,'screensize');
screen_half_w = screen_size(3) / 2;
screen_half_h = screen_size(4) / 2;

for i = 1:4
  f = figure(i);
  f.ToolBar = 'none';
  f.MenuBar = 'none';
  p = f.Position;
  p(3) = 560;
  p(4) = 420;
  p(1) = (screen_half_w - p(3)) + mod(i,2) * 560;
  p(2) = (screen_half_h - 2*p(4) - 30) + ceil(i/2) * 450;
  f.Position = p;
end
figure(5);
clf;cla;
```

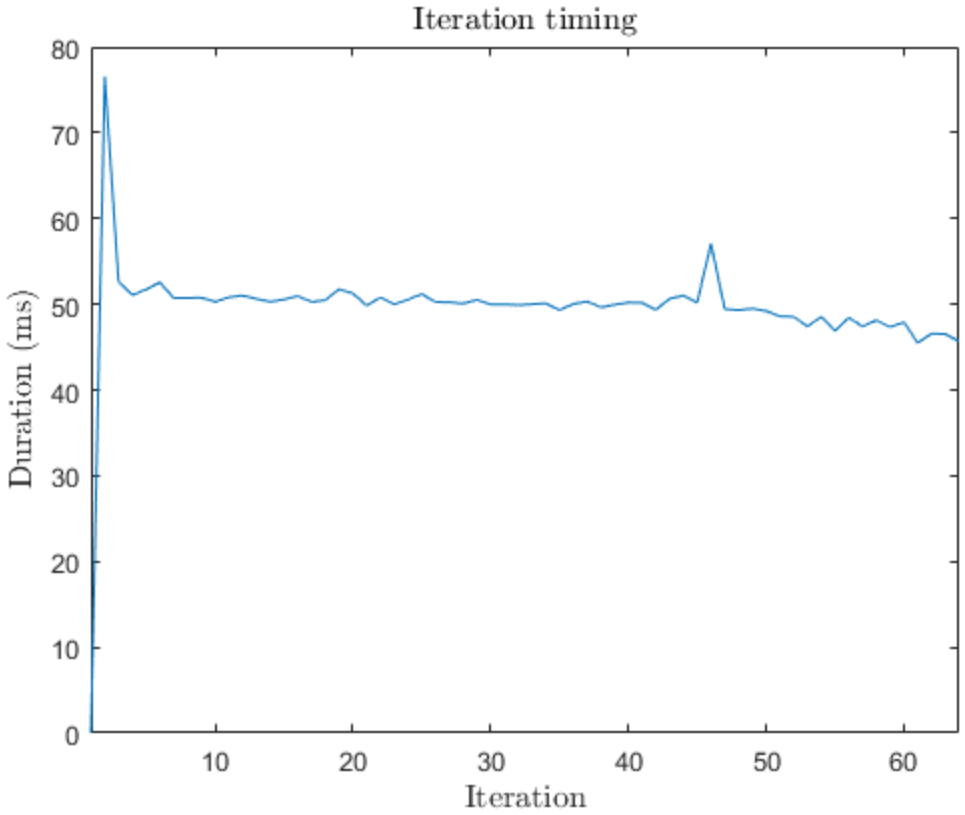
vvvvvv User interaction loop vvvvvv

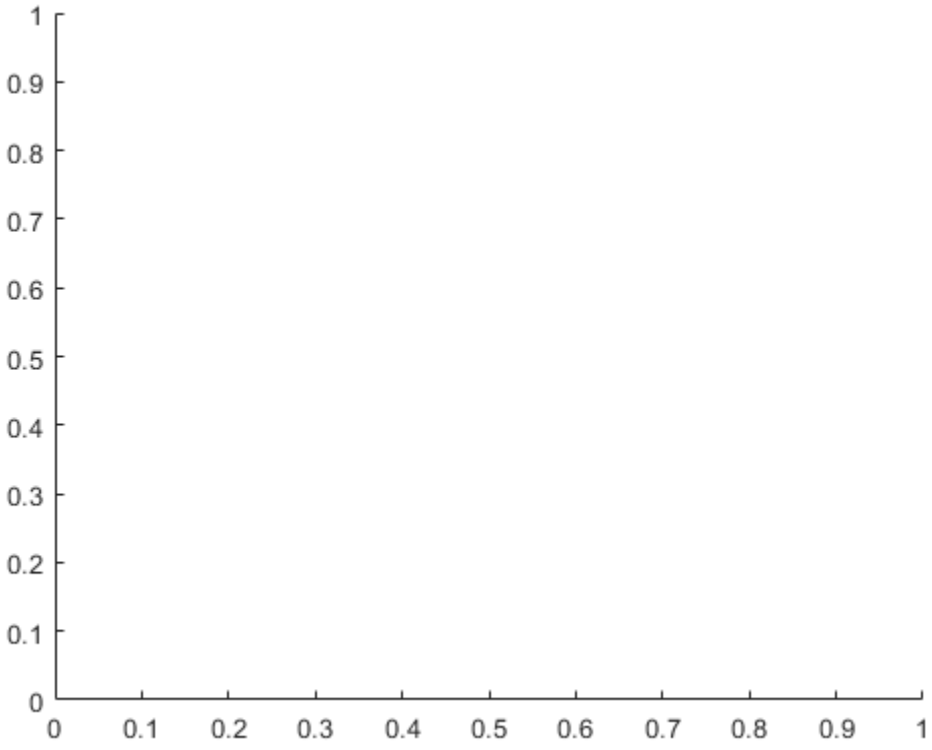
```
% Measuring stats between iterations
hTemps = 0;
hConverge = 0;

% Until ctrl-c, close figure, etc.
while true
  % This has to come first so we can draw on it as a background
  draw_cells();

if (toc(Input.last_click_t)) > 0.5

  % Compute the temperature distribution for the new mesh
  tic
  initialize_nodes();
  toc
  draw_nodes();

  node_temps = [];
  dTs = [];
  times = [];

  %disp("Iterating...");
  i = 1;
  node_temps(i,:) = [nodes.T_K];

  dT = 2;

  tic
  while dT > 1
    tic

    dT = iterate();
    dTs(i) = dT;

    draw_nodes();

    i = i + 1;
    node_temps(i,:) = [nodes.T_K];

    times(i) = toc * 1000;
  end

  %disp("Done.");

  figure(3);
  clf;cla;

  subplot(2,1,1);
  plot(KtoC(node_temps));
  xlim([1 i]);
  tex_title("Node temperature convergence");
```

```
tex_y('$T\;\mathrm{\left(^\circ C\right)}$');

subplot(2,1,2);
plot(dTs);
xlim([1 i]);
tex_x('Iteration');
tex_y('max$\left(dT\right)$ in K');

figure(4);
plot(times);
xlim([1 i]);
tex_title("Iteration timing");
tex_x("Iteration");
tex_y("Duration (ms)");

figure(2);
clf;cla;
% Unrefined nodes grid
[M,N]   = meshgrid(1:width_nodes);

% Make a refined grid to interpolate onto
[Mq,Nq] = meshgrid(1:0.01:width_nodes);

for m = 1:width_nodes
  for n = 1:height_nodes
    T(n,m) = KtoC(nodes(n,m).T_K);
  end
end


% Interpolate onto refined grid
Tq = interp2(M, N, T, Mq, Nq);
```


100 100 100 100 100 100 100 100 100 100
9 300 222 192 178 174 178 192 222 300 300
8 300 254 225 209 205 209 225 254 300 300
7 300 269 246 233 228 233 246 269 300 300
6 300 278 261 250 246 250 261 278 300 300
5 300 284 271 263 260 263 271 284 300 300
4 300 289 280 274 271 274 280 289 300 300
3 300 293 287 283 282 283 287 293 300 300
2 300 297 294 292 291 292 294 297 300 300
1 300 300 300 300 300 300 300 300 300
300 300 300 300 300 300 300 300 300 300
1 2 3 4 5 6 7 8 9


Node temperature convergence
T (°C)
400
200
0
-200
-400
10 20 30 40 50 60
max(dT) in K
300
200
100
0
10 20 30 40 50 60
Iteration

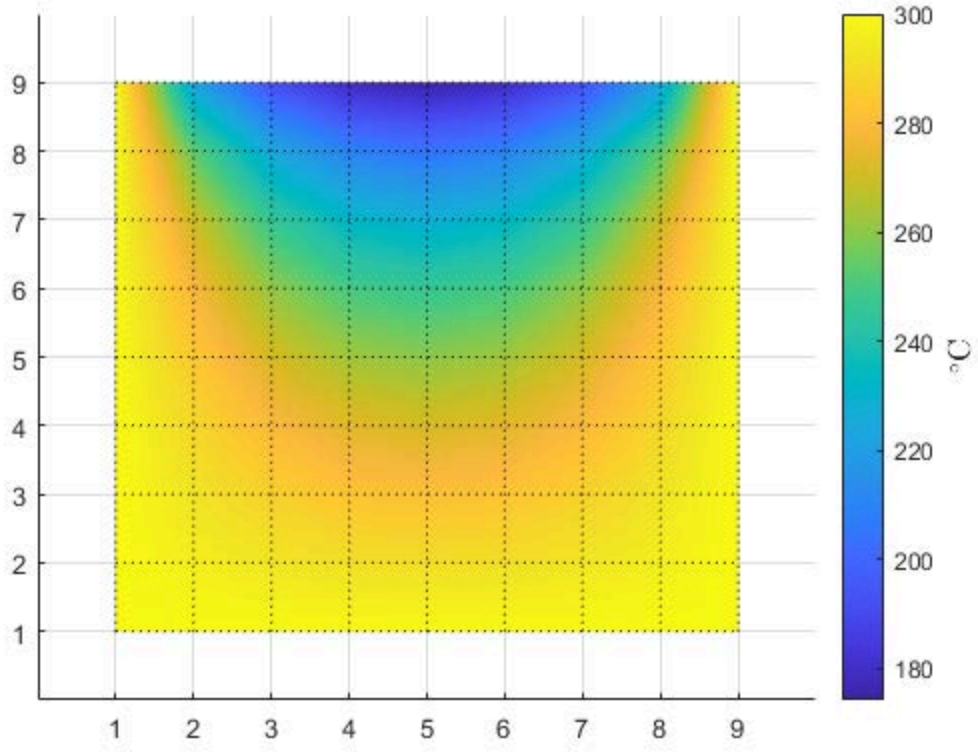

Iteration timing
Duration (ms)
Iteration
0
10
20
30
40
50
60
70
80

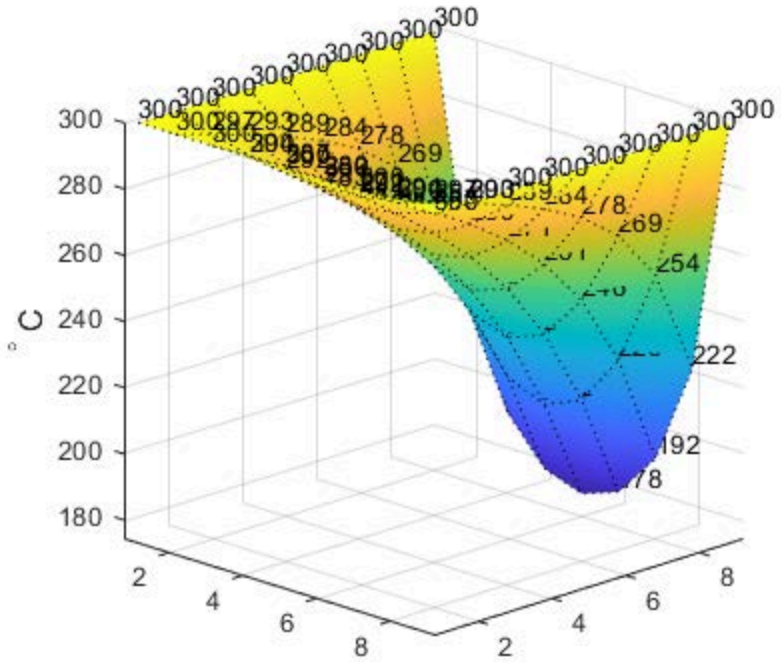

0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

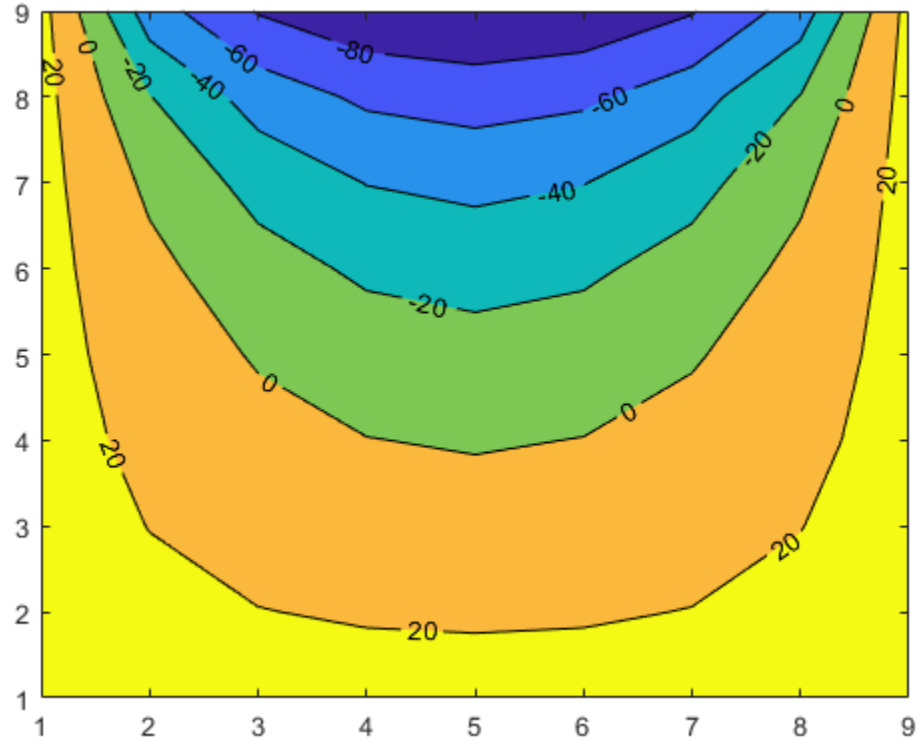
Draw results

```
% Draw interpolated temps
colormap default;
s = surf(Mq,Nq,Tq);
s.EdgeColor = 'none';
c = colorbar;
c.Label.String = '$^\circ$C';
c.Label.Interpreter = 'latex';
c.Label.FontSize = 12;
view(2);

% Draw grid
hold on;
s = surf(M,N,T);
s.FaceColor = 'none';
s.EdgeAlpha = 1;
s.LineStyle = ':';
view(2);

xlim([0 width_cells]);
ylim([0 height_cells]);
xticks(1:width_nodes);
yticks(1:height_nodes);

figure(5);
if Handle.surf3 ~= 0
  Handle.surf3.ZData = Tq;
  Handle.surf3g.ZData = T;
else
  Handle.surf3 = surf(Mq,Nq,Tq);
  hold on;

  axis vis3d;
  view(45, 18);
  zlabel("^\circ C", 'fontsize', 12);
  Handle.surf3.EdgeColor = 'None';
  % Draw grid
  hold on;
  Handle.surf3g = surf(M,N,T);
  Handle.surf3g.FaceColor = 'none';
  Handle.surf3g.EdgeAlpha = 1;
  Handle.surf3g.LineStyle = ':';

end

delete(Handle.ctexts);

for m = 1:width_nodes
  for n = 1:height_nodes
    Handle.ctexts = [Handle.ctexts text(m, n, T(n,m) + 5,
sprintf("%.0f", T(n,m)))];
  end
```

```
end

figure(6);
clf;cla;
colormap default;
contourf(KtoC(T), 'showtext', 'on');

draw_nodes();
```




300
280
260
240
220
200
180
°C
2
4
6
8


9
8
7
6
5
4
3
2
1
-80
-60
-40
-20
0
20

```
end % if click_wait_elapsed then iterate and draw

  % Start of a new interaction loop
  try
    [m, n, button] = wait_for_clicked_cell();

    if is_valid_cell(m,n)
      Input.last_click_t = tic;
      stop(Input.mouse_timer);
      start(Input.mouse_timer);
    end

    % Ugly hack for a big UX improvement (delayed simulation start)
    m = Input.last_m;
    n = Input.last_n;

    if ~is_valid_cell(m,n); continue; end

    % totally normal language with no case fallthrough
    if button == 2; button = 1; end
    switch button
    case 1; cycle_cell_type(m, n);
    case 3; prompt_new_temp_for_cell(m, n);
    end

  catch ex
    switch ex.identifier
      case 'MATLAB:ginput:FigureDeletionPause'
        break
      otherwise
        rethrow(ex)
    end
  end
end


info("\n\nGoodbye!\n\n");
```

^^^^^^^ User interaction loop ^^^^^^^

Replaces the current system state with the next computed state.

Returns the maximum magnitude of change in temperature among any node.

```
function [max_dT] = iterate()
  global nodes width_nodes height_nodes;

  next_nodes = nodes;

  % Maximum temperature change this iteration; converged when small
  max_dT = 0;
```

```
  for m = 1:width_nodes
    for n = 1:height_nodes
      node = get_node(m, n);

      if node.case == 0 || ~node.active; continue; end

      % Initialize to the constant term and sum the weighted neighbors
      node.T_K = node.coeff(5);

      neighbors = nodes_around(m, n);

      for neighbor = neighbors
        coeff  = node.coeff(neighbor.dir);
        node.T_K = node.T_K + coeff * neighbor.T_K;
      end

      % Measure convergence
      max_dT = max(max_dT, abs(node.T_K - next_nodes(n,m).T_K));

      next_nodes(n,m) = node;
    end
  end

  nodes = next_nodes;

end
```

Analyzes the material cells and configures the nodes

Node equations chosen according to: Bergman & Lavine 2017 - Fundamentals of Heat and Mass Transfer, p. 226 Table 4.2: Summary of nodal finite-difference equations

```
function initialize_nodes()
  global width_nodes height_nodes;

  debug("Determining node types:\n");

  for m = 1:width_nodes
    for n = 1:height_nodes
      node  = reset_node(m, n);
      cells = cells_around(m, n);

      debug("(%d,%d)", m, n);

      % Short circuiting
      try_case0(node, cells) || ...
      try_case1(node, cells) || ...
      try_cases234(node, cells);
  %#ok<VUNUS> % leave me alone I know it's dumb
    end
```

```
  end

end % initialize_nodes()
```

Draws and labels the nodes' states on top of the field

```
function draw_nodes()
  global width_nodes height_nodes;

  figure(1);

  for m = 1:width_nodes
    for n = 1:height_nodes
      node = get_node(m,n);

      % Ignore nodes that have become unreachable due to drowning
      if ~node.active
        if node.hLabel ~= 0
          node.hLabel.Visible = false;
        end
        continue;
      end

      if node.hLabel == 0
        color = 'w';
        switch node.case
        case 0
          color = 'y';
        case 1
          color = [.85 .85 .85];
        case 2
          color = [0.796, 0.925, 0.945];% If I had kept the first
 colors I chose
        case 3                          % then I would have submitted
 on time
          color = [0.351, 0.753, 0.826];
        case 4
          color = [0.062, 0.560, 0.635];
        end

        [x, y] = mn_to_xy(m, n);
        plot(x, y, '.w', 'markersize', 70, 'markeredgecolor', color);

        node.hLabel = label_mn(m, n, sprintf("%d",
 round(KtoC(node.T_K))));
        commit_node(node);

      else
        set(node.hLabel, 'string', round(KtoC(node.T_K)));

      end
```

```
    end
  end

  drawnow();

end
```

Tries to process the node as having a constant surface temperature

```
function [classified] = try_case0(node, neighbor_cells)
  global ConstantTemp

  classified = false;

  % Case 0: node touching a constant-temp cell is constant temp

  cells = [neighbor_cells.cell];
  types = [cells.type];
  cells = cells(types == ConstantTemp);

  % Average the neighboring cell temps
  Ts = mean(arrayfun(@(c) c.T_K, cells));
  if isempty(Ts) || isnan(Ts) || Ts == 1 || Ts == 2
    % not a constant temperature
    return
  end

  node.case = 0;
  node.T_K = Ts;

  debug(" --> constant temp\n");

  commit_node(node);
  classified = true;

end % try_case0()
```

Tries to process the node as an interior conduction node

```
function [classified] = try_case1(node, neighbor_cells)
  global Conductive

  classified = false;

  % Case 1: all edges are interior (between two conduction cells)
  cells = [neighbor_cells.cell];
  types = [cells.type];
  cells = cells(types == Conductive);
```

```
  if length(cells) == 4
    node.case = 1;
    % eq. 4.29, Bergman & Lavine 2017 p. 226
    node.coeff = [.25 .25 .25 .25 0];

    debug(" --> conductive\n");

    commit_node(node);
    classified = true;
  end

end % try_case1()
```

Tries to process the node as having convective neighbors

```
function [classified] = try_cases234(node, neighbor_cells)
  global Convective Conductive Env

  cells = [neighbor_cells.cell];
  types = [cells.type];

  conv_cells = cells(types == Convective);
  cond_cells = neighbor_cells(types == Conductive);

  % The number of convective cells touching this node determines the
  % heat equation
  n_conv = length(conv_cells);

  switch n_conv
  case 0
    throw("It couldn't be any other type of node");

  case 4
    % The node has become unreachable in a fluid domain
    debug(" node drowned\n")
    node.T_K      = mean([conv_cells.T_K]);
    node.active = false;

  otherwise
    node.case = 1 + n_conv;

    debug(" convection with %d cells, with conduction along:\n",
n_conv);
    % The convective node equation is of the form
    % Ti = Ci ( Ce Te + Cn Tn + Cw Tw + Cs Ts + H Tinf )
    % Where
    %   - Ci is determined from the number of convective nodes,
    %   - each of Ce, Cn, Cw, Cs is determined from the number of
    %     conductive cells bordering the edge to that node, and
    %   - Tinf is the average of all the convective nodes'
 temperatures.
```

```
    % Constant term
    Tinf          = mean([conv_cells.T_K]);
    node.coeff(5) = Env.Coeff.H * Tinf;

    % Examine the cells touching each conduction path. Each conductive
    % cell contributes half of the influence from that neighboring
  node.
    % Edges are in CCW order: E N W S

    % If dotting the vector from the node to a cell's center with one
  of
    % these vectors yields a positive result then that cell is
  touching
    % that edge.
    edges = [ 1 0 ; 0 1 ; -1 0 ; 0 -1 ];

    for i = 1:4
      edge = edges(i,:);
      debug("    edge %d (%d,%d)\tvia: ", i, edge(1,1), edge(1,2));
      for cell = cond_cells
        if dot(edge, cell.v) > 0
          node.coeff(i) = node.coeff(i) + 0.5;
          debug("(%d,%d) ", cell.m, cell.n);
        end
      end
      debug("\n");
    end

    % Overall coefficient
    C = 1 / (4 - n_conv + Env.Coeff.H);
    node.coeff = C * node.coeff;

  end % switch n_conv

  commit_node(node);
  classified = true;

end % try_case234()
```

Draws and labels the material cells as the base of a new image

```
function draw_cells()
  global Env width_cells height_cells Conductive;

  pixels = zeros(width_cells, height_cells);

  for m = 1:width_cells
    for n = 1:height_cells
      cell = get_cell(m,n);
      pixels(n,m) = cell.type;
```

```
    end
  end


  figure(1);
  clf;cla;

  % Arranged so that the cell types index into this colormap correctly
        %       yellow       grey         blue
  colormap([1 .890 .333; .8 .8 .8; .188 .812 .977 ])

  imagesc(pixels, [0 2]);
  view(2);
  grid on;
  hold on;
  % The default axes have the ticks marking the cell centers instead
of
  % the nodes. Adjust everything by half a cell to shift the ticks
over
  % to the nodes and make the label integral.
  xticks(mn_to_xy(1:width_cells));
  xticklabels(1:width_cells-1);
  yticks(mn_to_xy(1:height_cells));
  yticklabels(1:height_cells-1);
  % Flip the Y axis so the ticks increase vertically
  set(gca,'YDir','normal')

  for m = 1:width_cells
    for n = 1:height_cells
      c = get_cell(m, n);
      if c.type ~= Conductive
        % Label cell centers instead of nodes (i.e. don't adjust from
the
        % imagesc coordinate system).
        label_xy(m, n, sprintf("%d", round(KtoC(c.T_K))));
      end
    end
  end

  %tex_title(["Click to change cell type;\quad Right click to set
temperature", sprintf("Yellow: constant temp ($^\\circ$C); grey
metal (k=%.0f); blue fluid (h=%.0f)", Env.Materials.metal.k,
Env.Materials.fluid.h)]);

  drawnow();

end % draw_cells()
```

Returns location details of the cells contacting the node at (m,n)

```
function [cells] = cells_around(m, n)
```

```
  global width_cells height_cells;

  if (m >= width_cells) || (n >= height_cells) || (m <= 0) || (n <= 0)
    cells = [];
    return
  end

  m = m + 1;
  n = n + 1;

  % Every node m,n is the bottom-left corner of a cell m,n
  M = [ m  m-1  m-1   m  ];
  N = [ n   n   n-1  n-1 ];

  % Clip at the major border (clipping on the minor border is implicit
  % because the mouse picker gives indices starting at (1,1) and the
 cell
  % coordinates we're going to get are 0 based).
  is_valid = (M <= width_cells) .* (N <= height_cells);
  M = M(is_valid > 0);
  N = N(is_valid > 0);

  % Get a vector to the cell's center. A cell will affect an edge
 between
  % nodes A and B if the vector from A-->B projects onto this vector
 with
  % a positive sign.
  V = [((M-m)*2+1)' ((N-n)*2+1)'];

  for i = 1:length(M)
    cells(i).m = M(i);
    cells(i).n = N(i);
    cells(i).v = V(i,:);
    cells(i).cell = get_cell(M(i), N(i));
  end

end % cells_around()
```

Returns location details of the nodes contacting the node at (m,n)

```
function [nodes] = nodes_around(m, n)
  global width_nodes height_nodes;

  if (m > width_nodes) || (n > height_nodes) || (m <= 0) || (n <= 0)
    nodes = [];
    return
  end

    %    E    N    W    S
  M = [ m+1   m   m-1   m  ];
  N = [  n   n+1   n   n-1 ];
```

```
  is_valid = (M <= width_nodes) .* (N <= height_nodes) .* (M > 0) .*
 (N > 0);
  M = M(is_valid > 0);
  N = N(is_valid > 0);

  nodes = arrayfun(@get_node, M, N);

  for i = 1:length(nodes)
    nodes(i).dir = 1+mod(round((atan2(nodes(i).n - n, nodes(i).m - m)
 + 2*pi) * 2/pi), 4);
  end

end % nodes_around()
```

Changes the given cell to the next type (e.g. metal, fluid, constant temp)

```
function cycle_cell_type(m, n)
  global ConstantTemp;

  cell = get_cell(m, n);

  cell.type = mod(cell.type + 1, 3);
  if cell.type == ConstantTemp
    cell.T_K = CtoK(300);
  end

  commit_cell(cell);
end
```

Prompts the user for a temperature in degrees Celsius

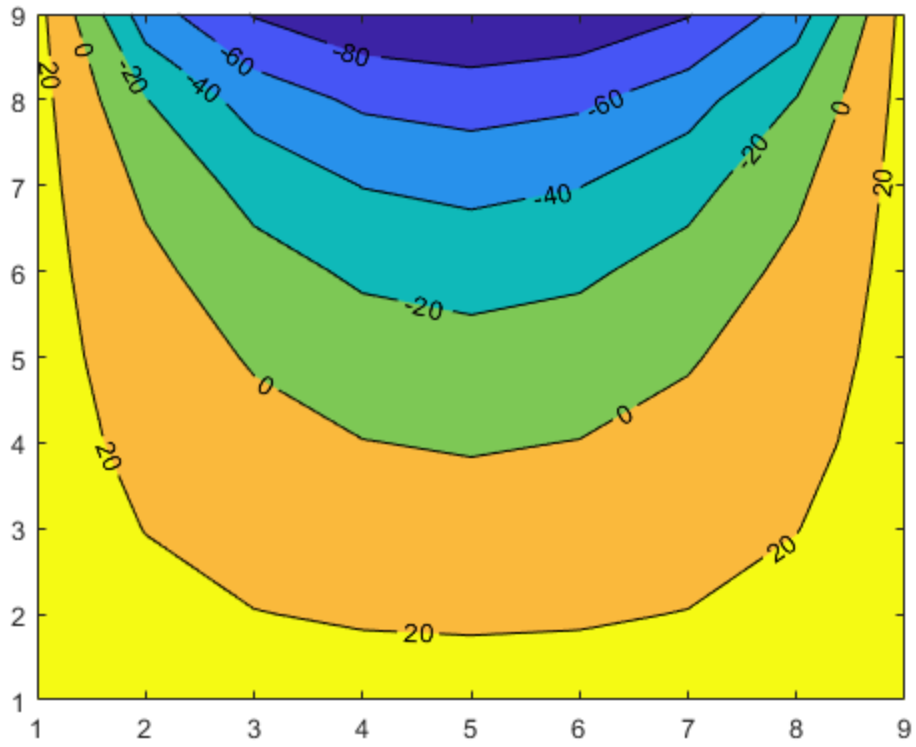
```
        If a valid temperature is given then the given cell is set to that
        temperature.

function prompt_new_temp_for_cell(m, n)
  global Input Conductive ConstantTemp

  stop(Input.mouse_timer);

  response = inputdlg("Enter a temperature in *C");
  if ~isempty(response)
    new_temp = str2double(response{1});
    if ~isempty(new_temp) && new_temp >= 0
      cell = get_cell(m, n);
      cell.T_K = CtoK(new_temp);
      if cell.type == Conductive
        cell.type = ConstantTemp;
```

```
      end
      commit_cell(cell);
    end
  end
  Input.last_click_t = tic;
  start(Input.mouse_timer);
end
```

Get a clicked point from the user until the point is inside a cell.

```
function [m, n, button] = wait_for_clicked_cell()
  global Input

  m = -1;
  n = -1;

  figure(1);

  [mx, my, button] = ginput(1);
  m = xy_to_mn(mx);
  n = xy_to_mn(my);

  if is_valid_cell(m,n)
    Input.last_m = m;
    Input.last_n = n;
  else
    Input.last_m = 0;
    Input.last_n = 0;
  end
end



function pressEnter(~,~)
  import java.awt.*;
  import java.awt.event.*;
  rob = Robot;
  rob.keyPress(KeyEvent.VK_ENTER)

  rob.keyRelease(KeyEvent.VK_ENTER)
end
```

Goodbye!



Returns true if the given coordinates are in the field

```
function [is_valid] = is_valid_cell(m, n)
  global width_cells height_cells;
  is_valid = ~isempty(m) && ~isempty(n) && (m >= 1) && (m <=
 width_cells) && (n >= 1) && (n <= height_cells);
end
```

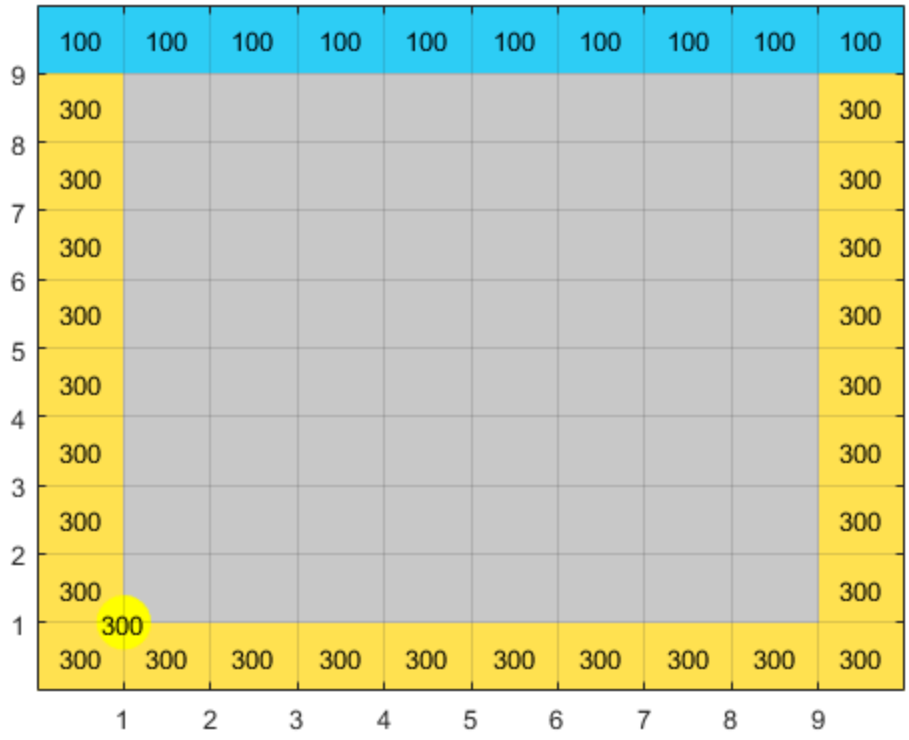
Returns the node at the given coordinates

```
function [node] = get_node(m, n)
  global nodes;
  node = nodes(n, m);
end
```

Overwrites the node in the field with the given value

```
function set_node(m, n, value)
  global nodes;
```

```
  nodes(n, m) = value;
end

Elapsed time is 1.928866 seconds.
```



Updates the node in the field from the given value

```
function commit_node(node)
  set_node(node.m, node.n, node);
end
```

Resets the node at the given coordinates to its initial state

```
function [node] = reset_node(m, n)
  global nodes;
  node = nodes(n, m);
  node.T_K = 0;
  node.hLabel = 0;
  node.active = true;
  node.coeff = [0 0 0 0 0];
  commit_node(node);
end
```

Returns the cell whose bottom-left node is at the given coordinates

```
function [cell] = get_cell(m, n)
  global cells;
  cell = cells(n, m);
end
```

Overwrites the cell whose bottom-left node is at the given coordinates

```
function set_cell(m, n, value)
  global cells;
  cells(n, m) = value;
end
```

Updates the cell in the field from the given value

```
function commit_cell(cell)
  set_cell(cell.m, cell.n, cell);
end
```

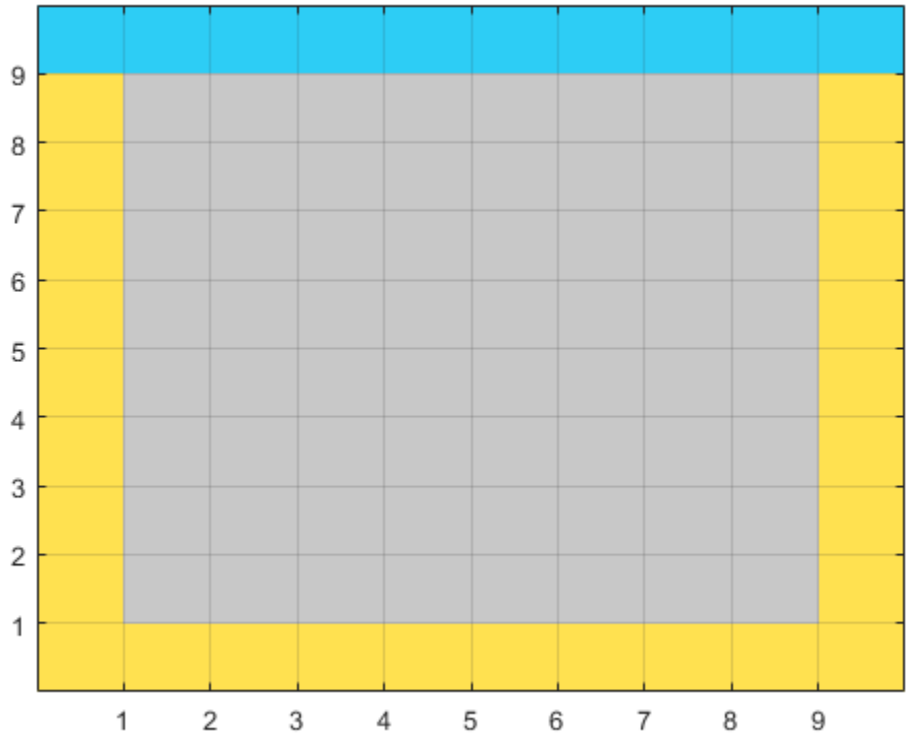
Converts imagesc coordinates to node coordinates

The image has weird coordinates because the cells are pixels, so the integral coordinates are at cell centers and nodes are on half-coordinates.

```
function [varargout] = xy_to_mn(varargin) %#ok<*DEFNU>
  for i = 1:min(nargin,nargout)
    varargout{i} = floor(varargin{i} + 0.5);
  end
end
```

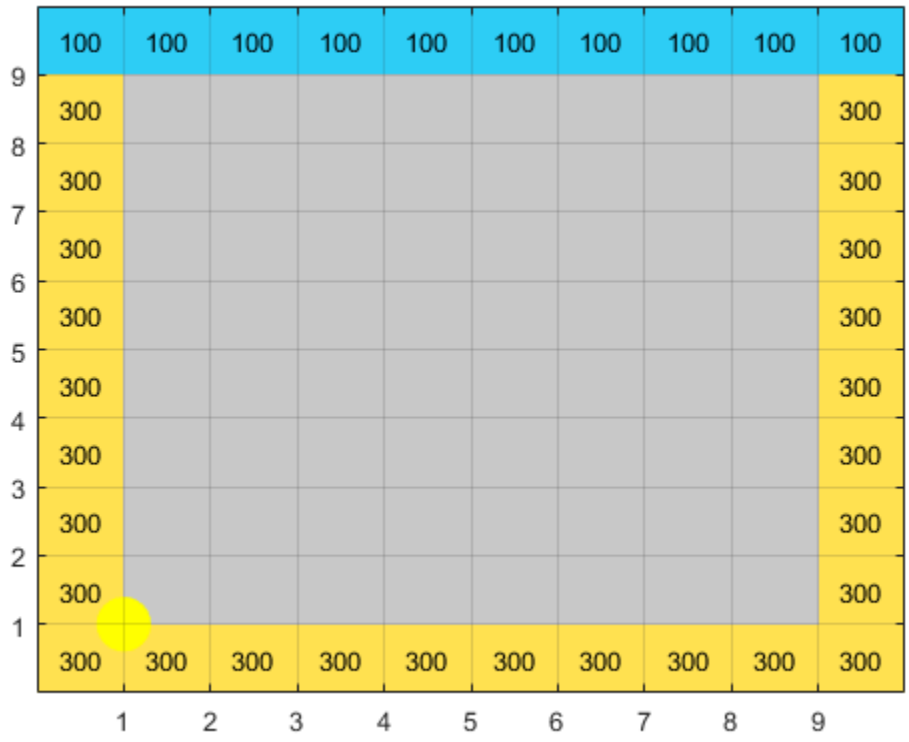
Converts node coordinates to imagesc coordinates

```
function [varargout] = mn_to_xy(varargin)
  for i = 1:min(nargin,nargout)
    varargout{i} = varargin{i} + 0.5; %#ok<*AGROW>
  end
end
```



Labels the current imagesc at a point in imagesc coordinates

```
function [h] = label_xy(x, y, s)
  h = text(x, y, s, 'horizontalalignment', 'center');
end
```



Labels the current imagesc at a point translated from node coordinates

```
function [h] = label_mn(m, n, s)
  [x, y] = mn_to_xy(m, n);
  h = label_xy(x, y, s);
end



function [C] = KtoC(K)
  C = K - 273.15;
end

function [K] = CtoK(C)
  K = C + 273.15;
end


function tex_title(s)
  title(s, 'interpreter', 'latex', 'fontsize', 12);
end

function tex_x(s)
  xlabel(s, 'interpreter', 'latex', 'fontsize', 12);
end

function tex_y(s)
  ylabel(s, 'interpreter', 'latex', 'fontsize', 12);
end

function message(level, varargin)
  if ~level; return; end
  fprintf(1, varargin{:});
end



function debug(varargin)
  global DEBUG;
  message(DEBUG, varargin{:});
end



function info(varargin)
  global INFO;
  message(INFO, varargin{:});
end
```

Published with MATLAB® R2021a