The Mathematica® Journal

Diffusion Modeling
Programming in Multiple Paradigms

George E. Hrabovsky

This article describes how to model diffusion using `NDSolve`, and then compares that to constructing your own methods using procedural, functional, rule-based, and modular programming. While based on the diffusion equation, these techniques can be applied to any partial differential equation.

Introduction

Using the built-in *Mathematica* command `NDSolve` to solve partial differential equations is very simple to do, but it can hide what is really going on. You can always consult references about using *Mathematica* for differential equations [1, 2]. Exploring various programming methods also lets you create your own procedures that can be incorporated into `NDSolve` [1]. I chose the diffusion equation as the main example because there is so much material available for it and because of its high level of interest [3, 4, 5]. In this article I am using *Mathematica* 8.

Begin with a model of diffusion, in this case, the diffusion equation. The continuity equation is

$$\frac{\partial \rho}{\partial t} - \nabla(\rho\, u_i) = 0,$$

where ρ is the density, t is time, and u_i is the i^{th} component of velocity. We can assume the rate of diffusion D is constant and write the diffusion equation,

$$\frac{\partial \rho}{\partial t} = D\, \nabla^2 \rho. \tag{1}$$

For complete generality, we can write

$$\frac{\partial \rho}{\partial t} = \partial_i D_{ij}\, \partial_j \rho,$$

where D_{ij} is a positive-definite matrix of diffusion coefficients that may or may not depend on position and time; this is the diffusion tensor.

Building Up Your Program Idea in Small Steps

First, try built-in functions to see if they can solve your problem right away. The plan here is to use `DSolve` and then, when it likely fails to solve the PDE, try `NDSolve`. We are solving for a ring, so we will use only one spatial dimension. We begin by writing our diffusion equation.

```
eq1 = ∂t ρ[x, t] == Dif ∂x,x ρ[x, t];
```

Now we try `DSolve`.

```
DSolve[eq1, {ρ[x, t]}, {x, t}]
```

```
DSolve[ρ^(0,1)[x, t] == Dif ρ^(2,0)[x, t], {ρ[x, t]}, {x, t}]
```

`DSolve` is unable to reach a solution to the naively formulated problem. We next provide a set of initial conditions.

```
iv = ρ[x, 0] == ρ0;
```

We incorporate the initial value and try again.

```
DSolve[{eq1, iv}, {ρ[x, t]}, {x, t}]
```

```
DSolve[{ρ^(0,1)[x, t] == Dif ρ^(2,0)[x, t], ρ[x, 0] == ρ0},
 {ρ[x, t]}, {x, t}]
```

Again it fails; we could play with the initial conditions until we found a set that worked, or we could try `NDSolve`. Let us say that we want to investigate the diffusion in a tube of gas. We consider this tube to be one dimensional to keep things simple. We assume our ring has a fixed length 1. We then "unwrap" the ring, making it a line, which lets us plot results. We will solve the equation for 10 units of time. We also specify the rate of diffusion.

```
Dif = 0.
```

```
0.
```

Let us try changing the initial density value to 0.

```
iv = ρ[x, 0] == 0;
```

```
bcs = {ρ[0, t] == 0, ρ[1, t] == 0};
```

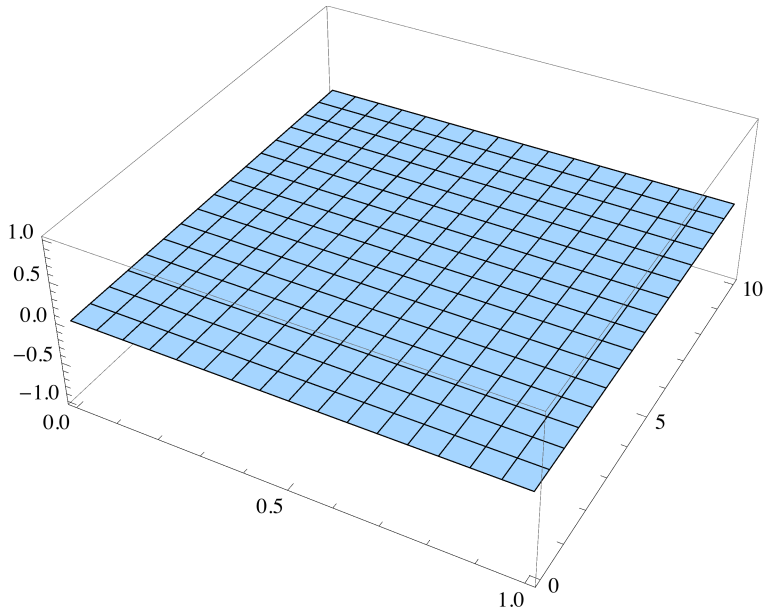
We put all of these elements together into `NDSolve` and get a solution in the form of an interpolating function.

```
sl1 = NDSolve[{eq1, iv, bcs}, {ρ[x, t]}, {x, 0, 1}, {t, 0, 10}]
```

```
{{ρ[x, t] →
    InterpolatingFunction[{{0., 1.}, {0., 10.}}, <>][x, t]}}
```

We can plot this function.

```
Plot3D[ρ[x, t] /. sl1, {x, 0, 1}, {t, 0, 10}]
```



▲ **Figure 1.** Solution to the diffusion equation with initial density of 0 in empty space.

Let us try another initial value, say a sinusoidal density wave.

```
iv = {ρ[x, 0] == 0, ρ[0, t] == Sin[t], ρ[1, t] == 0};
```

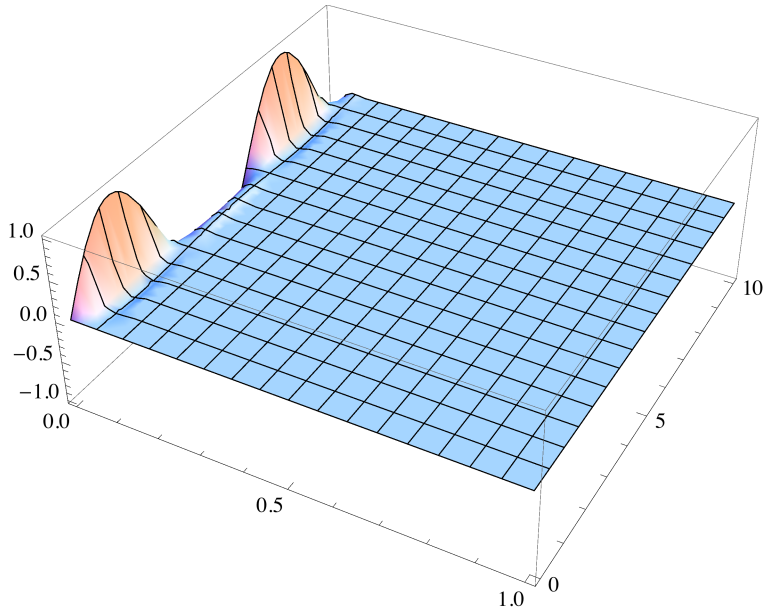
Now we try a solution.

```
sl2 = NDSolve[{eq1, iv}, {ρ[x, t]}, {x, 0, 1}, {t, 0, 10}]
```

```
{{ρ[x, t] →
    InterpolatingFunction[{{0., 1.}, {0., 10.}}, <>][x, t]}}
```

We can plot this.

```
Plot3D[ρ[x, t] /. sl2, {x, 0, 1}, {t, 0, 10}]
```



▲ **Figure 2.** Solution to the diffusion equation with initial density based on a sine function. We have not determined the rate of diffusion.

This does not seem realistic, as the density drops to zero immediately. Now we need to determine what D is, say 1.

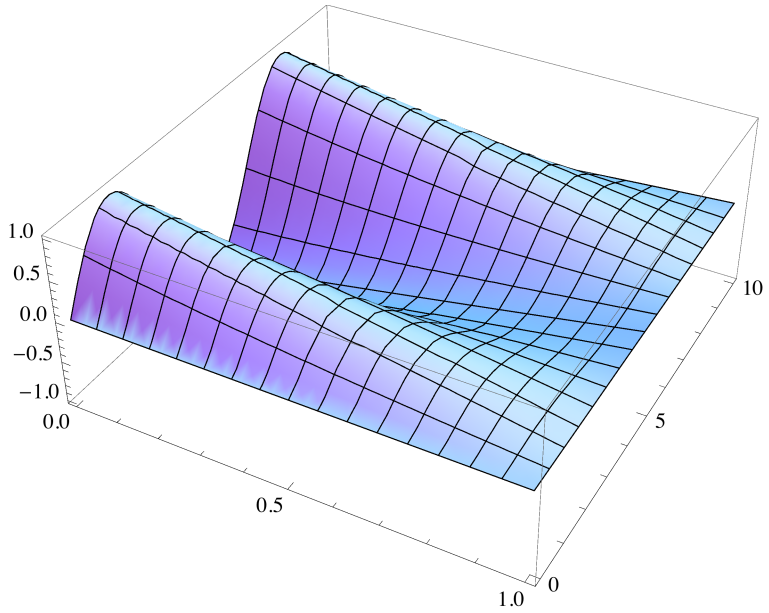
```
Dif = 1;
```

```
sl21 = NDSolve[{eq1, iv}, {ρ[x, t]}, {x, 0, 1}, {t, 0, 10}]
```

```
{{ρ[x, t] →
    InterpolatingFunction[{{0., 1.}, {0., 10.}}, <>][x, t]}}
```

Here is the plot.

```
Plot3D[ρ[x, t] /. sl21, {x, 0, 1}, {t, 0, 10}]
```



▲ **Figure 3.** Solution to the diffusion equation with sinusoidal boundary conditions.

This seems more realistic than Figure 2, but the boundary conditions do not match up. If we think of this as a circle (wrapping the line to form a ring), we suddenly get a discontinuity when we go from $x = 1$ to $x = 0$. We can repair our ring solution by using periodic boundary conditions.

```
iv = {ρ[x, 0] == 1, ρ[0, t] == Sin[t], ρ[2 π, t] == Sin[t]};
```

We are forcing an initial uniform density of 1, despite the sinusoidal boundary conditions in time, which would give us values of 0 density at the origin, and we do not want that. We get the solution, along with a warning message about inconsistent boundary and initial value conditions. Please ignore that in this case.

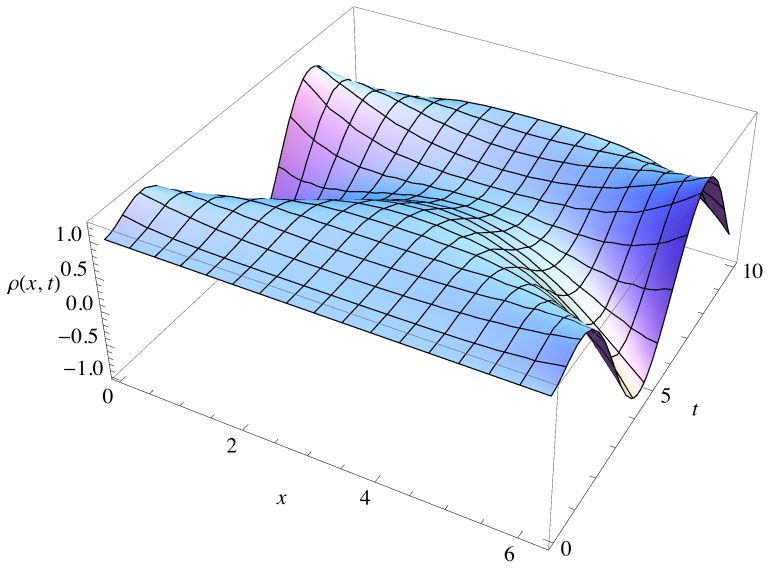
```
sl23 = NDSolve[{eq1, iv}, {ρ[x, t]}, {x, 0, 2 π}, {t, 0, 10}]
```

```
⚠ NDSolve::ibcinc : Warning: Boundary and initial conditions are inconsistent. ≫
```

```
{{ρ[x, t] → InterpolatingFunction[
      {{0., 6.28319}, {0., 10.}}, <>][x, t]}}
```

We can plot this.

```
Plot3D[ρ[x, t] /. sl23, {x, 0, 2 π}, {t, 0, 10},
 AxesLabel → {x, t, ρ[x, t]}]
```



▲ **Figure 4.** A fairly good model of diffusion on a ring with sinusoidal boundary conditions.

This looks reasonable. The boundary conditions are now periodic, as on a ring. How do we write a program to do this?

■ Procedural Programming of Diffusion

Those of you who have programmed in languages like FORTRAN, C++, or Java are familiar with the idea of procedural programming. Procedures are executed in looping structures. We implement a finite-difference scheme to solve our equation. `NDSolve` is able to use finite differences as a specific method [1].

Begin by converting the diffusion equation to its finite-difference equivalent,

$$\frac{\rho_i^{n+1} - \rho_i^n}{\Delta t} = D \frac{\rho_{i+1}^n - 2\,\rho_i^n + \rho_{i-1}^n}{(\Delta x)^2}. \tag{2}$$

Here a ring is divided into X cells for N time steps. Specify a given cell's density by considering the i^{th} cell for the n^{th} time step. So i extends from 0 to X, and time extends from 0 to N. For any specific time step,

$$\frac{\rho_i^{n+1}}{\Delta t} = \frac{\rho_i^n}{\Delta t} + D \frac{\rho_{i+1}^n - 2\,\rho_i^n + \rho_{i-1}^n}{(\Delta x)^2}.$$

The stability condition [2] is

$$\frac{D\Delta t}{(\Delta x)^2} \leq \frac{1}{2}. \tag{3}$$

If this ratio is greater than 1/2, the solution becomes unstable in time. Write

$$\rho_i^{n+1} = \Delta t\left(\rho_i^n + D\,\frac{\rho_{i+1}^n - 2\,\rho_i^n + \rho_{i-1}^n}{(\Delta x)^2}\right). \tag{4}$$

If `ρ0` is some set of initial conditions for ρ, in the next time step we can use this equation to get a set of values for ρ. We can then relabel these as the new `ρ0` values and continue the calculation. We have to establish the initial conditions and use the procedural iterator `Table[`*`function`*`, {`*`iterator`*`}]` to produce this result.

```
init[X_, inifun_] := Table[inifun, {i, 0, X}];
```

Choose this for positions 0 to 6.

```
ic = init[6, 1]
```

```
{1, 1, 1, 1, 1, 1, 1}
```

We then develop the current time step as the initial conditions. This specifies the part of the list considered, using `list[[`*`indices of sublist`*`]]`.

```
ρ0[x_] := ic[[x]]
```

Here are the periodic boundary conditions.

```
ρ0[1 - 1] = ρ0[Length[ic]]
```

```
1
```

```
ρ0[Length[ic] + 1] = ρ0[1]
```

```
1
```

This completes the initialization step, establishing the initial conditions.

We now construct a single time step by rewriting `ρ[x]` to include the boundary conditions, using `If[condition, condition is met, condition is not met]`.

```
ρ[x_, t_, d_, Δx_, Δt_, X_, bc_] :=
 If[x == 1 || x == X, bc,
  Δt (ρ0[x] + d/Δx^2 (ρ0[x + 1] + ρ0[x - 1] - 2 ρ0[x]))]
```

We test this.

```
ρ[1, 1, 1, 1, 1, 6, Sin[t]]
```

```
Sin[t]
```

```
ρ[3, 1, 1, 1, 1, 6, Sin[t]]
```

```
1
```

```
ρ[6, 1, 1, 1, 1, 6, Sin[t]]
```

```
Sin[t]
```

Define a time step, again using `Table` to generate the list.

```
step[X_, t_, d_, Δx_, Δt_, bc_] :=
 Table[ρ[i, t, d, Δx, Δt, X, bc], {i, 1, X, Δx}]
```

We check this.

```
step[6, 1, 1, 1, 1, Sin[t]] // N
```

```
{Sin[t], 1., 1., 1., 1., Sin[t]}
```

We now construct the solution in time, using `Table`.

```
run[X_, d_, Δx_, Δt_, N_, bc_] :=
 Table[a = step[X, t, d, Δx, Δt, bc]; ic = a, {t, 1, N, Δt}]
```

This completes the model development.

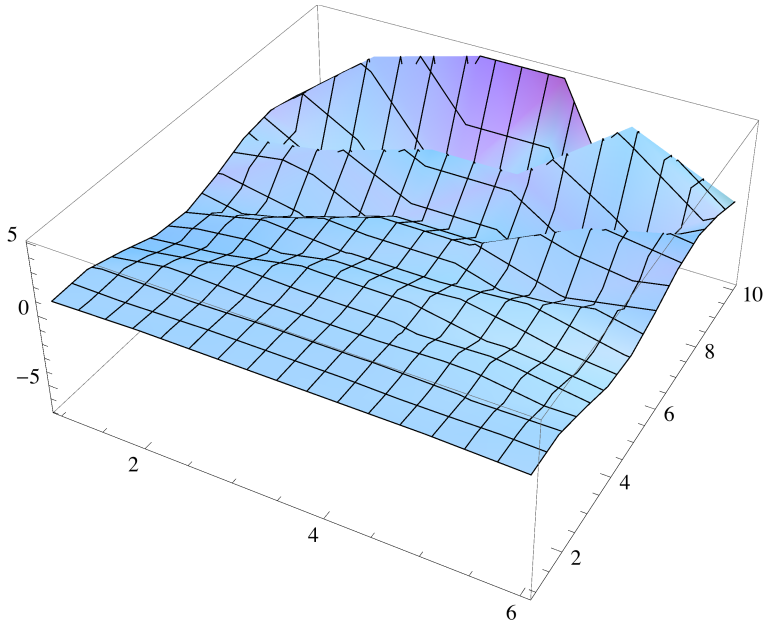
Now we need to display the results. I make a test called `run1`.

```
run1 = run[6, 1, 1, 1, 10, Sin[t]] // N
```

```
{{0.841471, 1., 1., 1., 1., 0.841471},
 {0.909297, 0.841471, 1., 1., 0.841471, 0.909297},
 {0.14112, 1.06783, 0.841471, 0.841471, 1.06783, 0.14112},
 {-0.756802, -0.0852354, 1.06783, 1.06783, -0.0852354,
  -0.756802}, {-0.958924, 0.396259, -0.0852354,
  -0.0852354, 0.396259, -0.958924}, {-0.279415,
  -1.44042, 0.396259, 0.396259, -1.44042, -0.279415},
 {0.656987, 1.55726, -1.44042, -1.44042, 1.55726, 0.656987},
 {0.989358, -2.3407, 1.55726, 1.55726, -2.3407, 0.989358},
 {0.412118, 4.88732, -2.3407, -2.3407, 4.88732, 0.412118},
 {-0.544021, -6.81589, 4.88732,
  4.88732, -6.81589, -0.544021}}
```

Here is the plot of these results.

```
ListPlot3D[run1]
```



▲ **Figure 5.** A jumbled mess of a solution for diffusion, not much like Figure 4.

This distorts and gets noisy very fast; of course we violated the stability condition with our choice of D, so let us try it again. We know from (3) that $D \leq 1/4$. We need to reinitialize $\rho 0$.

```
ic = init[6, 1]
```

```
{1, 1, 1, 1, 1, 1, 1}
```

```
ρ0[x_] := ic[[x]]
```

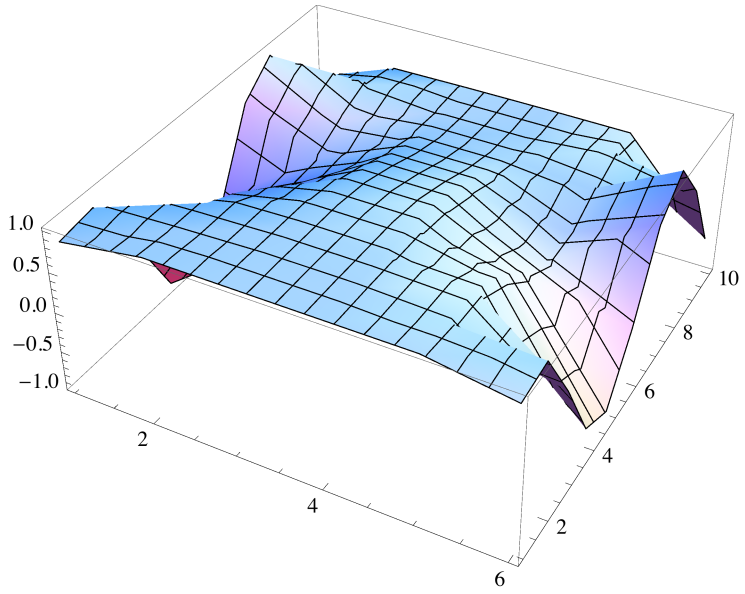
Then we construct `run` using the correct value for D.

```
run2 = run[6, 1/4, 1, 1, 10, Sin[t]] // N
```

```
{{0.841471, 1., 1., 1., 1., 0.841471},
 {0.909297, 0.960368, 1., 1., 0.960368, 0.909297},
 {0.14112, 0.957508, 0.990092, 0.990092, 0.957508, 0.14112},
 {-0.756802, 0.761557, 0.981946, 0.981946, 0.761557,
  -0.756802}, {-0.958924, 0.437064, 0.926849,
  0.926849, 0.437064, -0.958924}, {-0.279415,
  0.210513, 0.804403, 0.804403, 0.210513, -0.279415},
 {0.656987, 0.236503, 0.65593, 0.65593, 0.236503, 0.656987},
 {0.989358, 0.446481, 0.551074, 0.551074, 0.446481,
  0.989358}, {0.412118, 0.608348, 0.524925, 0.524925,
  0.608348, 0.412118}, {-0.544021, 0.538435,
  0.545781, 0.545781, 0.538435, -0.544021}}
```

This produces the result.

```
ListPlot3D[run2]
```



▲ **Figure 6.** A better solution that looks like Figure 4.

That looks better. This looks like the result we got from `NDSolve`, with some extra noise. We could use a finer grid and a better difference scheme if we wanted to get more precise and accurate. That is, however, the subject for another article.

■ Functional Programming

Another style of programming uses $f(x)$ as the application of f to x. So we start by establishing a function in the traditional way, using `Function[body][parameter replacement]`. Here we use the notation ⌗ to represent a formal argument.

```
Function[√(#^3)][x]
```

$$\sqrt{x^3}$$

We could write this in short form without the word `Function`, but then we need an ampersand following the function statement so that *Mathematica* knows that the function statement is complete.

```
Sqrt[#^3] &[x]
```

$$\sqrt{x^3}$$

We write `#1` and `#2` for two arguments.

```
Function[Sqrt[#1^3 + #2^2]][x, y]
```

$$\sqrt{x^3 + y^2}$$

Or again we can write this in short form.

```
Sqrt[#1^3 + #2^2] &[x, y]
```

$$\sqrt{x^3 + y^2}$$

We can apply a function to each element of a list by the use of the `Map[`*`function`*`, `*`list`*`]` command.

```
Map[Sqrt, {1, 2, 3, 4}]
```

$$\left\{1, \sqrt{2}, \sqrt{3}, 2\right\}$$

We could also use the short form *`function`* `/@`.

```
Sqrt /@ {1, 2, 3, 4}
```

$$\left\{1, \sqrt{2}, \sqrt{3}, 2\right\}$$

This is equivalent.

```
Sqrt[#] & /@ {1, 2, 3, 4}
```

$$\left\{1, \sqrt{2}, \sqrt{3}, 2\right\}$$

Say we want to program the diffusion equation using functional programming. We begin by defining the list of cells in the ring. We use `Range[max element]` to produce a list from 1 to *max element*.

```
fcells[X_] := Range[X]
```

We decide we want to go from 1 to 6.

```
fcells[6]
```

```
{1, 2, 3, 4, 5, 6}
```

We now define the initial conditions.

```
finit[X_, inifun_] := inifun /@ fcells[X]
```

For our choice of initial conditions, we have 1 everywhere, so we just say to divide by itself at each location.

```
ρ0 = finit[6, # / # &]
```

```
{1, 1, 1, 1, 1, 1}
```

Now we specify the diffusion equation. Since we have periodic boundary conditions and a list, we can just rotate the list right and left.

```
fρ[d_, Δx_, Δt_] :=
 Δt (ρ0 + d/Δx^2 (RotateRight[ρ0] + RotateLeft[ρ0] - 2 ρ0))
```

We can test this. First, clear the procedural definition of ρ from the previous section.

```
Clear[ρ]; ρ = fρ[1/4, 1, 1]
```

```
{1, 1, 1, 1, 1, 1}
```

We now need to impose our boundary conditions. Here we are replacing the end parts with the boundary conditions, using `ReplacePart[list, {parts and replacements}]`.

```
bc1[bc_, t_] := ReplacePart[ρ, {1 → bc[t], Length[ρ] → bc[t]}]
```

We will test these.

```
ρ = bc1[Sin, 1]
```

```
{Sin[1], 1, 1, 1, 1, Sin[1]}
```

So we construct a single step.

```
step[d_, Δx_, Δt_, bc_, t_] :=
 fρ[d, Δx, Δt] //
  ReplacePart[#, {1 → bc[t], Length[ρ] → bc[t]}] &
```

Let us try it.

```
step[1/4, 1, 1, Sin, 1]
```

```
{Sin[1], 1, 1, 1, 1, Sin[1]}
```

We can construct a table to produce the result.

```
frun[d_, Δx_, Δt_, bc_, N_] :=
 Table[a = step[d, Δx, Δt, bc, t]; ρ0 = a, {t, 1, N, Δt}]
```

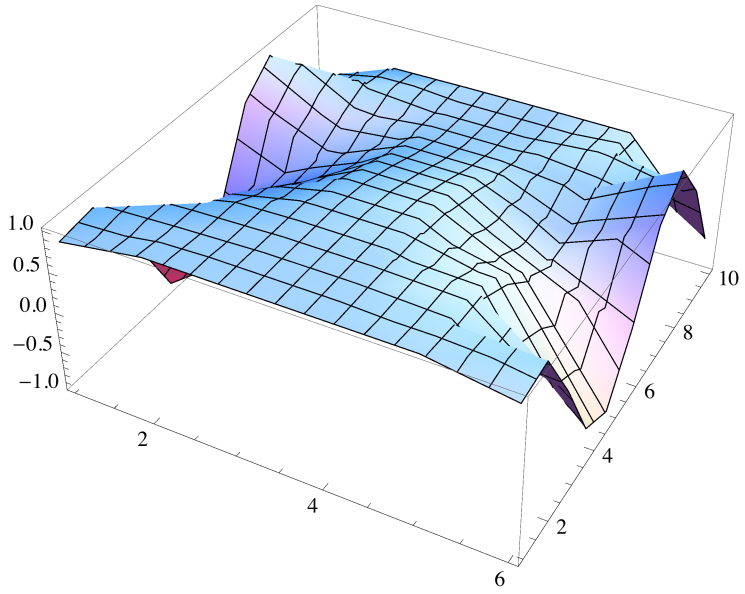
We can try this.

```
testrun = frun[1/4, 1, 1, Sin, 10] // N
```

```
{{0.841471, 1., 1., 1., 1., 0.841471},
 {0.909297, 0.960368, 1., 1., 0.960368, 0.909297},
 {0.14112, 0.957508, 0.990092, 0.990092, 0.957508, 0.14112},
 {-0.756802, 0.761557, 0.981946, 0.981946, 0.761557,
  -0.756802}, {-0.958924, 0.437064, 0.926849,
  0.926849, 0.437064, -0.958924}, {-0.279415,
  0.210513, 0.804403, 0.804403, 0.210513, -0.279415},
 {0.656987, 0.236503, 0.65593, 0.65593, 0.236503, 0.656987},
 {0.989358, 0.446481, 0.551074, 0.551074, 0.446481,
  0.989358}, {0.412118, 0.608348, 0.524925, 0.524925,
  0.608348, 0.412118}, {-0.544021, 0.538435,
  0.545781, 0.545781, 0.538435, -0.544021}}
```

We can plot this result.

```
ListPlot3D[testrun]
```



▲ **Figure 7.** Another solution that looks like Figure 4.

We have obtained the same result as with procedural programming.

■ Rule-Based Programming and Pattern-Matching

Another style of programming uses the symbolic nature of *Mathematica* to transform expressions. Rule-based programming uses the notion of transformation rules, which rewrite expressions based on their form. Take the expression $\sqrt{b^3}$ and apply a transformation rule to it.

```
Sqrt[b^3] /. b → x
```

$$\sqrt{x^3}$$

Here is a more complicated example.

```
Sqrt[b^3] /. b → (x^2 - y^2) // PowerExpand
```

$$(x^2 - y^2)^{3/2}$$

Coming back to the diffusion equation, we begin with the initial conditions.

```
rinit[X_, rule_] := Table[x, {X}] /. rule
```

Then we execute it.

```
ric = rinit[7, x → 1]
```

```
{1, 1, 1, 1, 1, 1, 1}
```

Then we have the diffusion equation. Here the double underscore __ following the ρ0 signifies a list.

```
rρ[ρ0__, Δt_, Δx_, d_] :=
 Δt (ρ0 + d/Δx^2 (RotateRight[ρ0] + RotateLeft[ρ0] - 2 ρ0))
```

For our example, define t1.

```
t1 = rρ[rinit[7, x → 1], 1, 1, 1 / 4]
```

```
{1, 1, 1, 1, 1, 1, 1}
```

Define the boundary conditions.

```
rstep[ρ0__, d_, Δx_, Δt_, bc_, t_] :=
 rρ[ρ0, d, Δx, Δt] //
  ReplacePart[#, {1 → bc[t], Length[ρ0] → bc[t]}] &
```

Test this for the first time step.

```
t1 = rstep[rinit[7, x → 1], 1, 1, 1 / 4, Sin, 1] // N
```

```
{0.841471, 1., 1., 1., 1., 1., 0.841471}
```

We can test this again.

```
rstep[rstep[rinit[7, x → 1], 1, 1, 1 / 4, Sin, 1], 1, 1,
  1 / 4, Sin, 2] // N
```

```
{0.909297, 0.960368, 1., 1., 1., 0.960368, 0.909297}
```

We need to include an intermediate function that specifies the values for our model for the `FoldList` command.

```
rρ1[ρ0__] := rρ[ρ0, 1, 1, 1/4]
```

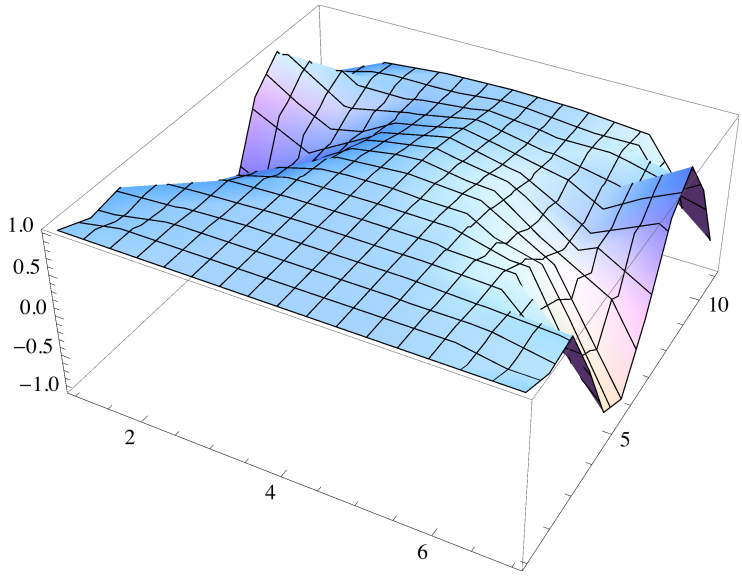
We can then produce our model `runt1`.

```
runt1 =
 FoldList[
   ReplacePart[rρ1[#1],
     {1 → Sin[#2], Length[#1] → Sin[#2]}] &, ric, Range[10]] //
  N
```

```
{{1., 1., 1., 1., 1., 1., 1.},
 {0.841471, 1., 1., 1., 1., 1., 0.841471},
 {0.909297, 0.960368, 1., 1., 1., 0.960368, 0.909297},
 {0.14112, 0.957508, 0.990092, 1., 0.990092,
  0.957508, 0.14112}, {-0.756802, 0.761557,
  0.984423, 0.995046, 0.984423, 0.761557, -0.756802},
 {-0.958924, 0.437684, 0.931362, 0.989734, 0.931362,
  0.437684, -0.958924}, {-0.279415, 0.211951,
  0.822536, 0.960548, 0.822536, 0.211951, -0.279415},
 {0.656987, 0.241756, 0.704393, 0.891542, 0.704393,
  0.241756, 0.656987}, {0.989358, 0.461223,
  0.635521, 0.797967, 0.635521, 0.461223, 0.989358},
 {0.412118, 0.636831, 0.632558, 0.716744, 0.632558,
  0.636831, 0.412118}, {-0.544021, 0.579585,
  0.654673, 0.674651, 0.654673, 0.579585, -0.544021}}
```

We can plot this.

```
ListPlot3D[runt1]
```



▲ **Figure 8.** Yet another solution that looks like Figure 4.

■ Scoping Constructs

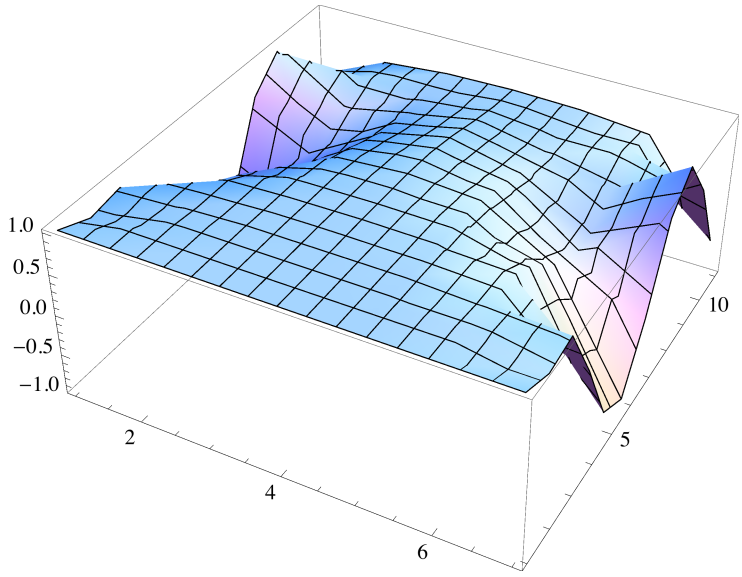
A scoping construct makes symbols local so that they do not create or redefine global values. We can make local object names using the command `Module[{names}, expression]`. We can temporarily assign values to variables with the command `Block[{variables}, expression]`. I will demonstrate the use of the `Module` command.

We can build a module for what we have been doing.

```
mrun[ρ0__, bc_, d_, s_, t_] :=
 Module[{Δt, Δx}, Δt = s; Δx = s;
  FoldList[
    ReplacePart[
      Δt (#1 + d/Δx^2 (RotateRight[#1] + RotateLeft[#1] - 2 #1)),
      {1 → bc[#2], Length[#1] → bc[#2]}] &, ρ0, Range[t]] //
   N // ListPlot3D[#] &]
```

We can run this and plot the result.

```
mrun[{1, 1, 1, 1, 1, 1, 1}, Sin, 1 / 4, 1, 10]
```



▲ **Figure 9.** Yet again another solution that looks like Figure 4.

Conclusion

We have seen that we can apply the canonical programming paradigms of *Mathematica* to the problem of diffusion. I have only scratched the surface of the diffusion problem. The methods I have presented could easily form the template for new methods for `NDSolve`, whose implementation is documented [1]. This flexibility in being able to introduce new methods into the existing structure of a *Mathematica* function is extremely powerful. Of course, the most powerful programming methods merge the paradigms.

References

[1] M. Sofroniou and R. Knapp, *Advanced Numerical Differential Equation Solving in Mathematica*, Champaign, IL: Wolfram Research, Inc., 2008. www.wolfram.com/learningcenter/tutorialcollection/AdvancedNumericalDifferentialEquationSolvingInMathematica.

[2] V. G. Ganzha and E. V. Vorozhtsov, *Numerical Solutions for Partial Differential Equations—Problem Solving Using Mathematica*, Boca Raton, FL: CRC Press, Inc., 1996.

[3] R. Ghez, *A Primer of Diffusion Problems*, New York: John Wiley and Sons, Inc., 1988. onlinelibrary.wiley.com/book/10.1002/3527602836.

[4] E. L. Cussler, *Diffusion*: *Mass Transfer in Fluid Systems*, 3rd ed., New York: Cambridge University Press, 2009.

[5] J. Crank, *The Mathematics of Diffusion*, 2nd ed., Oxford, England: Clarendon Press, 1975.

G. E. Hrabovsky, "Diffusion Modeling," *The Mathematica Journal*, 2012. dx.doi.org/doi:10.3888/tmj.14-6.

About the Author

George Hrabovsky is the president and founder of Madison Area Science and Technology, a nonprofit scientific research and education organization. He has been a *Mathematica* user and programmer for more than 20 years. He does research into theoretical physics, atmospheric science, and computational physics.

George E. Hrabovsky
105 Alhambra Place #2
Madison, WI 53713
george@madscitech.org