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

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

where $\rho$ 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,

$$\frac{\partial \rho}{\partial t} = D \nabla^2 \rho. \quad (1)$$

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.

\[
eq = \partial_t \rho(x, t) = \text{Dif} \partial_x \rho(x, t);
\]

Now we try DSolve.

\[
\text{DSolve}[\{eq1, \{\rho[x, t]\}, \{x, t\}\}]
\]

\[
\text{DSolve}\left[\{\rho^{(0,1)}[x, t] = \text{Dif} \rho^{(2,0)}[x, t], \{\rho[x, t]\}, \{x, t\}\} \right]
\]

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

\[
i v = \rho[x, 0] = \rho 0;
\]

We incorporate the initial value and try again.

\[
\text{DSolve}[\{eq1, iv\}, \{\rho[x, t]\}, \{x, t\}\}
\]

\[
\text{DSolve}\left[\{\rho^{(0,1)}[x, t] = \text{Dif} \rho^{(2,0)}[x, t], \{\rho[x, 0] = \rho 0\}, \rho[x, t]\}, \{x, t\}\} \right]
\]

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.

\[
\text{Dif} = 0.
\]

0.

Let us try changing the initial density value to 0.

\[
i v = \rho[x, 0] = 0;
\]

\[
\text{bcs} = \{\rho[0, t] = 0, \rho[1, t] = 0\};
\]
We put all of these elements together into NDSolve and get a solution in the form of an interpolating function.

\[
\text{sl1} = \text{NDSolve}[[\text{eq1, iv, bcs}, \{\rho[x, t]\}, \{x, 0, 1\}, \{t, 0, 10\}]]
\]

\[
\{\{\rho[x, t] \rightarrow \\
\quad \text{InterpolatingFunction}[[\{0., 1.\}, \{0., 10.\}], <>][x, t]\}\}
\]

We can plot this function.

\[
\text{Plot3D}[\rho[x, t] /. \text{sl1}, \{x, 0, 1\}, \{t, 0, 10\}]
\]

\[\text{Figure 1. Solution to the diffusion equation with initial density of 0 in empty space.}\]

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

\[
\text{iv} = \{\rho[x, 0] = 0, \rho[0, t] = \text{Sin}[t], \rho[1, t] = 0\};
\]

Now we try a solution.

\[
\text{sl2} = \text{NDSolve}[[\text{eq1, iv}, \{\rho[x, t]\}, \{x, 0, 1\}, \{t, 0, 10\}]]
\]

\[
\{\{\rho[x, t] \rightarrow \\
\quad \text{InterpolatingFunction}[[\{0., 1.\}, \{0., 10.\}], <>][x, t]\}\}
\]
We can plot this.

\[ \text{Plot3D}[\rho[x, t]/. \text{sl2}, \{x, 0, 1\}, \{t, 0, 10\}] \]

\[ \text{Figure 2. Solution to the diffusion equation with initial density based on a sine function. We have not determined the rate of diffusion.} \]

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

\[ \text{Dif} = 1; \]

\[ \text{sl21 = NDSolve[}\{\text{eq1, iv, }\{\rho[x, t]\}, \{x, 0, 1\}, \{t, 0, 10\}] \]

\[ \{\{\rho[x, t] \rightarrow \text{InterpolatingFunction[\{\{0. \}, \{0., 10.\}, <\}][x, t}\}\} \]
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}]
```

```
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]}]
```

\[\text{Figure 4. A fairly good model of diffusion on a ring with sinusoidal boundary conditions.}\]

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,

$$\frac{ρ_i^{n+1} - ρ_i^n}{Δt} = D \frac{ρ_{i+1}^n - 2 ρ_i^n + ρ_{i-1}^n}{(Δx)^2}. \quad (2)$$

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,

$$\frac{ρ_i^{n+1}}{Δt} = \frac{ρ_i^n}{Δt} + D \frac{ρ_{i+1}^n - 2 ρ_i^n + ρ_{i-1}^n}{(Δx)^2}.$$
The stability condition [2] is
\[
\frac{D\Delta t}{(\Delta x)^2} \leq \frac{1}{2}.
\] (3)

If this ratio is greater than 1/2, the solution becomes unstable in time. Write
\[
\rho_i^{n+1} = \Delta t \left( \rho_i^n + D \frac{\rho_{i+1}^n - 2\rho_i^n + \rho_{i-1}^n}{(\Delta x)^2} \right).
\] (4)

If \(\rho_0\) is some set of initial conditions for \(\rho\), in the next time step we can use this equation to get a set of values for \(\rho\). We can then relabel these as the new \(\rho_0\) values and continue the calculation. We have to establish the initial conditions and use the procedural iterator \(\text{Table}[\text{function}, \{\text{iterator}\}]\) to produce this result.

\[
\text{init}[X_, \text{inifun}_\_] := \text{Table}[\text{inifun}, \{i, 0, X\}];
\]

Choose this for positions 0 to 6.

\[
\text{ic} = \text{init}[6, 1]
\]

\[
\{1, 1, 1, 1, 1, 1, 1\}
\]

We then develop the current time step as the initial conditions. This specifies the part of the list considered, using \(\text{list}[\text{indices of sublist}]\).

\[
\rho_0[x_] := \text{ic}[[x]]
\]

Here are the periodic boundary conditions.

\[
\rho_0[1-1] = \rho_0[\text{Length}[\text{ic}]]
\]

\[
1
\]

\[
\rho_0[\text{Length}[\text{ic}]+1] = \rho_0[1]
\]

\[
1
\]

This completes the initialization step, establishing the initial conditions.
We now construct a single time step by rewriting \( \rho[x] \) to include the boundary conditions, using 

\[
\text{If}[\text{condition}, \text{condition is met}, \text{condition is not met}].
\]

\[
\rho[x_, t_, d_, \Delta x_, \Delta t_, X_, bc_] := 
\text{If}[x == 1 || x == X, bc, 
\Delta t \left( \rho_0[x] + \frac{d}{\Delta x^2} \left( \rho_0[x+1] + \rho_0[x-1] - 2 \rho_0[x] \right) \right)
\]

We test this.

\[
\rho[1, 1, 1, 1, 1, 6, \text{Sin}[t]]
\]

\[
\text{Sin}[t]
\]

\[
\rho[3, 1, 1, 1, 1, 6, \text{Sin}[t]]
\]

\[
1
\]

\[
\rho[6, 1, 1, 1, 1, 6, \text{Sin}[t]]
\]

\[
\text{Sin}[t]
\]

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

\[
\text{step}[X_, t_, d_, \Delta x_, \Delta t_, bc_] := 
\text{Table}[\rho[i, t, d, \Delta x, \Delta t, X, bc], \{i, 1, X, \Delta x\}]
\]

We check this.

\[
\text{step}[6, 1, 1, 1, 1, \text{Sin}[t]] // \text{N}
\]

\[
[\text{Sin}[t], 1., 1., 1., 1., 1., \text{Sin}[t]]
\]

We now construct the solution in time, using Table.

\[
\text{run}[X_, d_, \Delta x_, \Delta t_, N_, bc_] := 
\text{Table}[a = \text{step}[X, t, d, \Delta x, \Delta t, bc]; \text{ic} = a, \{t, 1, N, \Delta t\}]
\]

This completes the model development.
Now we need to display the results. I make a test called \texttt{run1}.

\begin{verbatim}
run1 = run[6, 1, 1, 10, Sin[t]] // N
\end{verbatim}

\begin{verbatim}
{{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}}
\end{verbatim}

Here is the plot of these results.

\texttt{ListPlot3D[run1]}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{A jumbled mess of a solution for diffusion, not much like Figure 4.}
\end{figure}
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 $\rho 0$.

\begin{verbatim}
ic = init[6, 1]
(1, 1, 1, 1, 1, 1)

\rho 0[x_] := ic[[x]]
\end{verbatim}

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

\begin{verbatim}
run2 = run[6, \frac{1}{4}, 1, 1, 10, Sin[t]] // N
\end{verbatim}

\begin{verbatim}
{(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})
\end{verbatim}
This produces the result.

\[
\text{ListPlot3D[run2]}
\]

\[\text{Figure 6. A better solution that looks like Figure 4.}\]

That looks better. This looks like the result we got from \texttt{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.

\section*{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 \texttt{Function[body][parameter replacement]}. Here we use the notation \(\square\) to represent a formal argument.

\[
\text{Function}[^x^3][x] \\
\sqrt{x^3}
\]
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.

\[ \sqrt{x^3} \&[x] \]
\[ \sqrt{x^3} \]

We write \( \&1 \) and \( \&2 \) for two arguments.

\[ \text{Function} \left[ \sqrt{\#1^3 + \#2^2} \right] [x, y] \]
\[ \sqrt{x^3 + y^2} \]

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 \( \text{Map}[\text{function}, \text{list}] \) command.

\[ \text{Map} [\text{Sqrt,} \{1, 2, 3, 4\}] \]
\[ \{1, \sqrt{2}, \sqrt{3}, 2\} \]

We could also use the short form \( \text{function} /@ \).

\[ \text{Sqrt} /@ \{1, 2, 3, 4\} \]
\[ \{1, \sqrt{2}, \sqrt{3}, 2\} \]

This is equivalent.

\[ \text{Sqrt}[\#] \& /@ \{1, 2, 3, 4\} \]
\[ \{1, \sqrt{2}, \sqrt{3}, 2\} \]
Say we want to program the diffusion equation using functional programming. We begin by defining the list of cells in the ring. We use \texttt{Range[max element]} to produce a list from 1 to \texttt{max element}.

\[ \texttt{fcells[X_] := Range[X]} \]

We decide we want to go from 1 to 6.

\[ \texttt{fcells[6]} \]

\[ \{1, 2, 3, 4, 5, 6\} \]

We now define the initial conditions.

\[ \texttt{finit[X_, inifun_] := inifun /@ fcells[X]} \]

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

\[ \rho_0 = \texttt{finit[6, \# / \# &]} \]

\[ \{1, 1, 1, 1, 1, 1\} \]

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

\[ \texttt{f\rho[d_, \Delta x_, \Delta t_] :=} \]
\[ \Delta t \left( \rho_0 + \frac{d}{\Delta x^2} (\text{RotateRight}[\rho_0] + \text{RotateLeft}[\rho_0] - 2 \rho_0) \right) \]

We can test this. First, clear the procedural definition of \( \rho \) from the previous section.

\[ \texttt{Clear[\rho]; \rho = f\rho[\frac{1}{4}, 1, 1]} \]

\[ \{1, 1, 1, 1, 1\} \]

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

\[ \texttt{bcl[bc_, t_] := ReplacePart[\rho, \{1 \rightarrow bc[t], \text{Length}[\rho] \rightarrow bc[t]\}]} \]
We will test these.

\[ \rho = \text{bci}[\sin, 1] \]

\[ \{\sin[1], 1, 1, 1, 1, \sin[1]\} \]

So we construct a single step.

\[
\text{step}[d\_, \Delta x\_, \Delta t\_, bc\_, t\_] :=
\]

\[
\text{frun}[d\_, \Delta x\_, \Delta t\_, bc\_, N\_] :=
\]

Let us try it.

\[
\text{step}[\frac{1}{4}, 1, 1, \sin, 1]
\]

\[ \{\sin[1], 1, 1, 1, 1, \sin[1]\} \]

We can construct a table to produce the result.

\[
\text{frun}[d\_, \Delta x\_, \Delta t\_, bc\_, N\_] :=
\]

We can try this.

\[
\text{testrun} = \text{frun}[\frac{1}{4}, 1, 1, \sin, 10] // N
\]

\[
\{\{0.841471, 1., 1., 1., 1., 0.841471\},
0.909297, 0.960368, 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.

\[ \text{ListPlot3D[testrun]} \]

![Figure 7](image)

▲ 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.

\[ \sqrt{b^3 \ / \ b \rightarrow x} \]

\[ \sqrt{x^3} \]

Here is a more complicated example.

\[ \sqrt{b^3 \ / \ b \rightarrow (x^2 - y^2) \ // \ PowerExpand} \]

\[ (x^2 - y^2)^{3/2} \]
Coming back to the diffusion equation, we begin with the initial conditions.

\[
\text{rinit}[X_\text{__}, \text{rule}_\text{__}] := \text{Table}[x, \{X\}] / . \text{rule}
\]

Then we execute it.

\[
\text{ric} = \text{rinit}[7, x \to 1]
\]

\[
\{1, 1, 1, 1, 1, 1, 1\}
\]

Then we have the diffusion equation. Here the double underscore \(\_\_\) following the \(\rho\) signifies a list.

\[
\text{rr}[\rho\_\_\_, \Delta t\_\_, \Delta x\_\_, d\_\_] :=
\Delta t \left( \rho_0 \cdot \frac{d}{\Delta x^2} (\text{RotateRight}[\rho_0] + \text{RotateLeft}[\rho_0] - 2 \rho_0) \right)
\]

For our example, define \(t_1\).

\[
t_1 = \text{rr}[\text{rinit}[7, x \to 1], 1, 1, 1/4]
\]

\[
\{1, 1, 1, 1, 1, 1, 1\}
\]

Define the boundary conditions.

\[
\text{rstep}[\rho\_\_\_, d\_\_, \Delta x\_\_, \Delta t\_\_, \text{bc}_\_\_, t\_\_] :=
\text{rr}[\rho_0, d, \Delta x, \Delta t] //
\text{ReplacePart}[\#, \{1 \to \text{bc}[t], \text{Length}[\rho_0] \to \text{bc}[t]\}] &
\]

Test this for the first time step.

\[
t_1 = \text{rstep}[\text{rinit}[7, x \to 1], 1, 1, 1/4, \text{Sin}, 1] // \text{N}
\]

\[
\{0.841471, 1., 1., 1., 1., 1., 0.841471\}
\]

We can test this again.

\[
\text{rstep}[\text{rstep}[\text{rinit}[7, x \to 1], 1, 1, 1/4, \text{Sin}, 1], 1, 1, 1/4, \text{Sin}, 2] // \text{N}
\]

\[
\{0.909297, 0.960368, 1., 1., 1., 0.960368, 0.909297\}
\]
We need to include an intermediate function that specifies the values for our model for the FoldList command.

\[ r\rho_1[\rho0_] := r\rho[\rho0, 1, 1, \frac{1}{4}] \]

We can then produce our model `runt1`.

```mathematica
runt1 = FoldList[
  ReplacePart[r\rho[\#1],
    {1 \rightarrow \text{Sin}[\#2], \text{Length}[\#1] \rightarrow \text{Sin}[\#2]}) &, \text{ric}, \text{Range}[10]] //
  N

{(1., 1., 1., 1., 1., 1., 1., 1.),
 (0.841471, 1., 1., 1., 1., 1., 1., 0.841471),
 (0.909297, 0.960368, 1., 1., 1., 1., 0.909297),
 (0.14112, 0.957508, 0.990092, 1., 0.990092,
  0.957508, 0.14112), {-0.756802, 0.761557, 0.984423, 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.

\[
\text{ListPlot3D[runt1]}
\]

\[\text{Figure 8. Yet another solution that looks like Figure 4.}\]

### 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[\rho0\_, bc\_, d\_, s\_, t\_] :=
\quad\text{Module[}\{\Delta t, \Delta x\}, \Delta t = s; \Delta x = s;
\quad\text{FoldList[}
\quad\quad\text{ReplacePart[}
\quad\quad\quad\Delta t \left(\frac{d}{\Delta x^2} (\text{RotateRight}[\#1] + \text{RotateLeft}[\#1] - 2 \#1)\right),
\quad\quad\quad\{1 \rightarrow \text{bc}[\#2], \text{Length}[\#1] \rightarrow \text{bc}[\#2]\}\), \&, \rho0, \text{Range}[t]\}\text{] //}
\quad\quad\text{N // ListPlot3D[\#] \&}
\]
We can run this and plot the result.

\[ \texttt{mrun[\{1, 1, 1, 1, 1, 1, 1\}, Sin, 1/4, 1, 10]} \]

\[ \text{\textbf{Figure 9.}} \text{ Yet again another solution that looks like Figure 4.} \]

\section*{Conclusion}

We have seen that we can apply the canonical programming paradigms of \textit{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 \texttt{NDSolve}, whose implementation is documented \cite{1}. This flexibility in being able to introduce new methods into the existing structure of a \textit{Mathematica} function is extremely powerful. Of course, the most powerful programming methods merge the paradigms.

\section*{References}

\begin{enumerate}
\end{enumerate}


**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