MathPDE: A Package to Solve PDEs by Finite Differences

K. Sheshadri
Peter Fritzson

A package for solving time-dependent partial differential equations (PDEs), *MathPDE*, is presented. It implements finite-difference methods. After making a sequence of symbolic transformations on the PDE and its initial and boundary conditions, *MathPDE* automatically generates a problem-specific set of *Mathematica* functions to solve the numerical problem, which is essentially a system of algebraic equations. *MathPDE* then internally calls *MathCode*, a *Mathematica*-to-C++ code generator, to generate a C++ program for solving the algebraic problem, and compiles it into an executable that can be run via *MathLink*. When the algebraic system is nonlinear, the Newton–Raphson method is used and SuperLU, a library for sparse systems, is used for matrix operations. This article discusses the wide range of PDEs that can be handled by *MathPDE*, the accuracy of the finite-difference schemes used, and importantly, the ability to handle both regular and irregular spatial domains. Since a standalone C++ program is generated to compute the numerical solution, the package offers portability.

■ 1. Introduction

Mathematical problems described by partial differential equations (PDEs) are ubiquitous in science and engineering. Examples range from the simple (but very common) diffusion equation, through the wave and Laplace equations, to the nonlinear equations of fluid mechanics, elasticity, and chaos theory. However, few PDEs have closed-form analytical solutions, making numerical methods necessary. The numerical task is made difficult by the dimensionality and geometry of the independent variables, the nonlinearities in the system, sensitivity to boundary conditions, a lack of formal understanding of the kind of solution method to be employed for a particular problem, and so on.

A number of methods have been developed to deal with the numerical solution of PDEs. These fall into two broad categories: the finite-difference methods and the finite-element methods. Roughly speaking, both transform a PDE problem to the problem of solving a system of coupled algebraic equations. In finite-difference methods, the domain of the independent variables is approximated by a discrete set of points called a grid, and the dependent variables are defined only at these points. Any derivative then automatically acquires the meaning of a certain kind of difference between dependent variable values at the grid points. This identification helps us transform the PDE problem to a system of coupled algebraic equations. In the finite-element method, the dependent variable is approximated by an interpolation polynomial. The domain is subdivided into a set of non-overlapping regions that completely cover it; each such region is called a finite element. The interpolation polynomial is determined by a set of coefficients in each element; the small size of the elements generally ensures that a low-degree polynomial is sufficient. A weighted residual method is then used to set up a system of algebraic equations for these coefficients. The numerical problem in both methods therefore is one of solving a system of algebraic equations.

In this article, we present the *Mathematica* package *MathPDE* that implements the finite-difference method for time-dependent PDE problems. We give examples of the way in which symbolic programming lends a degree of generality to *MathPDE*. Before doing so, we would like to put our work in perspective. We first review, very briefly, the extant PDE packages that we are familiar with that have been developed over the years.

Mathematica has a built-in function `NDSolve` that can numerically solve a variety of PDEs and offers the user a very simple and quick route to solving PDEs numerically. So a natural question that arises is why another solver is needed. In this regard, we emphasize two points. First, *MathPDE* is able to handle irregular spatial domains, not just rectangular or cubic domains. Second, *MathPDE* can produce a standalone executable that runs independently of *Mathematica*, providing portability.

The following systems use either a compiled language directly, or have a high-level interface language that preprocesses the input and employs subroutines in a compiled language. None of them uses symbolic programming.

Diffpack (see [1]) presents an object-oriented problem-solving environment for the numerical solution of PDEs. It implements finite-difference as well as finite-element methods and provides C++ modules with a wide selection of interchangeable and application-independent components. ELLPACK (see [2]) presents a high-level interface language for formulating elliptic PDE problems and presents more than 50 problem-solving modules for handling complex elliptic boundary value problems. It is implemented as a FORTRAN preprocessor and can handle a variety of system geometries in two dimensions (both finite differences and finite elements) and rectangular geometries in three dimensions (finite differences only). Cogito and COMPOSE were developed at Uppsala University, Sweden (see [3, 4]). Both implement finite-difference schemes for time-dependent problems and exploit object-oriented technology to develop a new kind of software library with parts that can be flexibly combined, enhancing easy construction and modification of programs. Cogito is a high-performance solver that comprises the three layers Parallel, Grid, and Solver, the lower two layers being Fortran 90 parallel code, while the Solver is written in C++. COMPOSE is a C++ object-oriented system that exploits Overture [5] for grid generation.

The work on numerical PDE libraries is far too extensive for us to be able to present an exhaustive review here (see [6, 7, 8, 9] for overviews). We would like to emphasize that while *MathPDE* has the generality to handle a wide range of PDE problems, it is not useful, at least in its present form, for certain problems where very specialized finite-difference decoupling schemes are known to work much better than the straightforward finite differencing that our approach uses. The Navier–Stokes equation is one such example (see [4] for an approach to this problem). In general, the work discussed in this paper is not adequate for problems where specialized numerical libraries have been known to perform effectively. Instead, it is suitable for PDE problems that are often encountered in engineering modeling situations. Even for such problems, several static subroutine libraries have been developed that perform efficiently, as discussed above. On the other hand, *MathPDE* can treat a wide range of spatial domains (see Section 3). There is also the great potential to combine the enormous benefits of the interactive *Mathematica* environment with its large library of built-in mathematical functions, which enables the user to experiment with a large class of PDE problems. Finally, *MathPDE* generates C++ source code and a standalone executable, addressing the important issue of portability.

Our preliminary results were reported in the references [10, 11, and 12]. We now give a brief summary of the present paper. *MathPDE* accepts the PDE and the initial and boundary conditions, along with the geometry of the system, in *Mathematica*'s list format. It supports geometries of fairly general shapes and dimensions. The symbolic engine applies explicit and implicit difference methods and discretizes the geometry to a grid. *MathPDE* then generates a program for solving the numerical problem, which is essentially an algebraic equation system on the grid; if the system is nonlinear, then a multidimensional Newton–Raphson method [7] is used to solve it. This program makes use of the external library SuperLU to efficiently handle sparse linear algebraic systems [13]. *MathPDE* internally calls *MathCode* [14] to translate the numerical program into C++ and then to generate an executable. All these operations are done by invoking a single function (`SetupMathPDE`) provided by *MathPDE*. The executable can then be run either from within *Mathematica* or externally to obtain the numerical solution of the PDE problem.

Here is an outline of this article. In Section 2, we provide a quick introduction to *MathPDE* with the example of a one-dimensional diffusion equation.

We discuss the ideas behind the package in Section 3. Section 3.1 is about the numerical algorithms used, such as the discretization of derivatives, the solution of an algebraic system, and Fourier stability. In Section 3.2, we discuss how the spatial domain is discretized. Section 3.3 discusses code generation and its translation into C++ using *MathCode*.

In Section 4, we discuss application examples that illustrate the strengths and limitations of *MathPDE*. The examples chosen are divided into problems in one, two, and three dimensions, and there is a section on two nonlinear problems.

In Section 5, we give a summary of our work and make some concluding remarks.

2. A Quick Tour of *MathPDE*

In this section we provide a brief tour of *MathPDE*, taking the example of a one-dimensional diffusion equation. We illustrate how to define the PDE problem, set up the numerical problem using *MathPDE*, and generate C++ code using *MathCode*. The package *MathPDE* is loaded in the usual way.

```
Needs["MathPDE`"];
```

2.1 Problem Definition

Let us solve the one-dimensional diffusion equation

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2}, \tag{1}$$

with Dirichlet conditions

$u(0,t) = u(1,t) = 0,$

at the boundaries of the domain

$0 \le x \le 1,$

and the initial condition

$u(x,0) = x(1-x).$

The problem is defined as a list.

```
diffusion1D = {
    {
     {∂{t,1} u[x, t] == ∂{x,2} u[x, t]},
     {x == 0, u[x, t] == 0},
     {x == 1, u[x, t] == 0},
     {t == 0, u[x, t] == x (1 - x)}
    },
    {
     {{x, 0, 1}}
    }
  };
```

The list has two elements: the first element defines the PDE, the boundary conditions, and finally the initial condition, each as a list, and the second element defines the spatial domain as an iterator.

2.2 Setup

Once the problem is defined, the following command sets up the numerical code.

```
SetupMathPDE[diffusion1D, {}]
```

```
Execute:

SolvePDE[Nx, dt, m, n]

to obtain a solution to your PDE using C code generated by
   MathCode, if the C code successfully compiles. Here,
 {Nx} is a list of sizes for the variables {x}, and
 dt is the step-size for the variable t.

When you execute SolvePDE for the first time,
   you must set m=1, so that it first computes the
   initial conditions, and then evolves them up to
 t = (n+m-1)*dt. For subsequent calls of SolvePDE,
   choose any value of m other than 1, to evolve
   the solution by a further n steps.

The fields {u} are stored in U⟦2⟧, which has Nx elements (
 Nx for each field). Execute solutionAt[x, i, Nx]
  to obtain the value of the i-th field at the point
 {x}(these must be integers).
```

```
MathCode C++ 1.3.1 for mingw32 loaded from C:\MathCode
```

```
Successful compilation to C++: 13 function(s)
```

```
MathPDE is installed.
```

In this case, the last argument `{}` of `SetupMathPDE` is empty because there are no parameters in the input problem. If there were parameters, they could be specified in this list. After completing the symbolic part, the command loads *MathCode*, which then generates C++ code and compiles it. In the present problem, the C++ code is a set of 13 functions in addition to the main() function. The executable is also installed by *MathCode*, so that we can run it from the notebook interface.

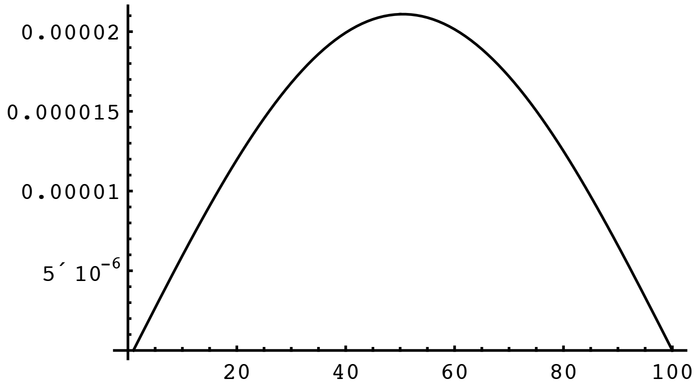
2.3 Numerical Computation

We can now execute the function `SolvePDE` to obtain a numerical solution.

```
SolvePDE[100, 0.01, 1, 100];
```

We can extract the solution computed by this command by invoking `solutionAt` and we can plot it.

```
ListLinePlot[Table[solutionAt[i, 1, 100], {i, 1, 100}]]
```



We can uninstall the C++ program and delete all the files related to *MathCode* if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before trying a new problem.

```
Quit
```

3. Design of *MathPDE*

3.1 Numerical Algorithms

3.1.1 Choosing an Appropriate Finite-Difference Method

In the finite-difference method, the independent variables are regarded as discrete and the domain becomes a grid. The derivatives of the dependent variables then automatically become differences between values at a combination of these grid points; the actual combination depends on the nature of the difference approximation. After such an approximation is performed, no derivatives are left and only the functions are evaluated at the grid points; the resulting system of relations between the values of dependent variables at a set of neighboring grid points is referred to as the *stencil* for the PDE system. Applying the stencil at all the grid points results in a system of coupled algebraic equations. In time-dependent problems, to which the present work is addressed, the solution at any time t_n (which is discretized as well) is determined, in general, from the solution up to the previous time instant t_{n-1}; the solution values at each grid point at the time instant t_n are the unknowns in the algebraic system.

Depending on the kind of discretization used, we may be able to explicitly solve for each unknown from one of the algebraic equations. In such a case, the difference scheme is called explicit. In cases where we cannot do this explicitly, we refer to the difference scheme as implicit.

Let us illustrate this with the example of the one-dimensional diffusion equation that we solved in Section 2. Suppose we replace $u_t(x, t)$ by a first-degree forward approximation, $(u(x, t+k) - u(x, t))/k$, and $u_{xx}(x, t)$ by a second-degree central approximation, $(u(x+h, t) - 2\,u(x, t) + u(x-h, t)) \big/ h^2$, where k and h are the step sizes along t and x. The result is the difference equation

$$(u(x, t+k) - u(x, t))/k = (u(x+h, t) - 2\,u(x, t) + u(x-h, t)) \big/ h^2 \tag{2}$$

that can be easily solved for $u(x, t+k)$ to get

$$u(x, k+t) = \frac{h^2\,u(x, t) - 2\,k\,u(x, t) + k\,u(-h+x, t) + k\,u(h+x, t)}{h^2}, \tag{3}$$

so we have an explicit finite-difference method. A simple but important observation is that the right-hand side of the above equation involves dependent variable values at time t that are already known, so we can use the right-hand side to recursively compute the dependent variable at any grid point and at any time, given the initial and boundary conditions. On the other hand, if we replace $u_{xx}(x, t)$ by a second-degree central approximation, $(u(x+h, t+k) - 2\,u(x, t+k) + u(x-h, t+k)) \big/ h^2$ at time $t+k$, rather than at time t, then we get a difference equation

$$(u(x, t+k) - u(x, t))/k = (u(x+h, t+k) - 2\,u(x, t+k) + u(x-h, t+k)) \big/ h^2 \tag{4}$$

that *cannot* be solved for the dependent variable $u(x, t+k)$ in terms of $u(x, t)$. In this case, we have an implicit finite-difference method, since the spatial derivative is advanced to the highest time $t+k$. In this case, since we have a linear system, we can state the problem in terms of matrices, and typically we have to solve a matrix problem of the kind $A \cdot X = B$. Nonlinear systems have to be handled differently; see Section 3.1.5.

In the symbolic part, *MathPDE* first extracts the list of all the derivatives of dependent variables with respect to independent variables from the equations specified by the user. This is straightforward using symbolic programming. Then an important decision to make is to choose the finite-difference approximation (FDA) to use for each derivative. This is based on the following simple algorithm.

Suppose we have to choose an FDA degree for the n^{th} derivative with respect to x of $u(x, t)$. We look for all the derivatives of u with respect to x in the given PDE, and find the highest order, $n_{\max}$; we use $n_{\max}$ as the approximation degree for all derivatives of u with respect to x, no matter what order. The same procedure works for derivatives with respect to other independent variables. The next thing to decide is which discretization to choose among the $2\, n_{\max} + 1$ possible choices for the FDA (see Section 3.1.2). We choose a central-difference approximation scheme for derivatives with respect to spatial variables and a forward scheme for time derivatives. Implicit schemes can be obtained by advancing spatial derivatives. It is easy to see that this algorithm leads to the approximation schemes discussed above for the diffusion equation. *MathPDE* generates both explicit and implicit schemes, and the choice between them is made at run time based on considerations of numerical stability (see Section 3.1.3).

For boundary conditions, we choose a forward or a backward scheme, depending on whether the condition applies at the left or the right boundary.

It is possible for the user to intervene and force a different scheme (see Section 3.1.4).

■ 3.1.2 Discretization of the Derivatives: Fornberg Algorithm

Once a choice has been made for the approximation degree for a derivative, we must obtain its finite-difference approximation. A straightforward way to do this is to use the Lagrange interpolation formula (see any text on numerical analysis, for example [7]). However, an elegant algorithm due to Fornberg is particularly suitable for implementation in *Mathematica* [15]. We have used his very simple code to discretize a derivative.

```
FD2[order_, degree_, kind_] :=
 Module[{t, x},
  t = PadeApproximant[x^kind * Log[x]^order, {x, 1, degree}];
  CoefficientList[Numerator[t], x]
 ]
```

Now, if we want a second-degree (`degree = 2`) central approximation (`kind = 2`) to a second-order (`order = 2`) derivative, we must use this.

```
FD2[2, 2, 2]
```

```
{1, -2, 1}
```

This gives us the coefficients in the resulting stencil (compare with the coefficients on the right-hand side of equation (2)). A *MathPDE* function `diffApprox[]` based on the above function `FD2[]` can then be used to obtain the FDA of any derivative. The following gives the second-degree (`degree = 2`) central approximation (`kind = 2`) to a second-order (`order = 2`) derivative.

```
diffApprox[u^(2,0)[x, t], 2, 2, h]
```

$$\frac{u[-1+x,\ t]-2\,u[x,\ t]+u[1+x,\ t]}{h^2}$$

This gives the first-degree forward approximation to a first-order time derivative; here `h` and `k` are the step sizes in `x` and `t`, respectively. Mixed derivatives can be easily handled with recursive calls to `diffApprox[]`, as done by the *MathPDE* function `discretize[]` that discretizes equations involving derivatives, as illustrated by the discretization of the following PDE.

```
diffApprox[u^(0,1)[x, t], 1, 1, k]
```

$$\frac{-u[x,\ t]+u[x,\ 1+t]}{k}$$

```
discretize[
 D[u[x, y, t], {x, 2}] + D[u[x, y, t], {y, 2}] +
   2 D[u[x, y, t], {x, 1}, {y, 1}] == D[u[x, y, t], {t, 2}],
 {u[x[{2, 2}, {2, 2}], y[{2, 2}, {2, 2}], t[{2, 1}, {2, 1}]]},
 {h1, h2, k}]
```

$$\frac{u[x,\ -1+y,\ t]-2\,u[x,\ y,\ t]+u[x,\ 1+y,\ t]}{h2^2}+\frac{u[-1+x,\ y,\ t]-2\,u[x,\ y,\ t]+u[1+x,\ y,\ t]}{h1^2}+\frac{1}{h1\,h2}\left(\frac{1}{2}u[-1+x,\ -1+y,\ t]-\frac{1}{2}u[-1+x,\ 1+y,\ t]-\frac{1}{2}u[1+x,\ -1+y,\ t]+\frac{1}{2}u[1+x,\ 1+y,\ t]\right)==\frac{u[x,\ y,\ t]-2\,u[x,\ y,\ 1+t]+u[x,\ y,\ 2+t]}{k^2}$$

The second argument to the function `discretize[]`, `{u[x[{2, 2}, {2, 2}], y[{2, 2}, {2, 2}], t[{2, 1}, {2, 1}]]}`, specifies the FDA to use for each derivative that appears in the PDE. It contains the two integers `degree` and `kind` to be passed as arguments to `diffApprox[]` for each derivative. Thus, the first argument `{2, 2}` of `x` means that the FDA must be second degree, central, for the first `x` derivative of `u`; the second argument of `x` specifies the integers required to approximate the *second* derivative of `u` with respect to `x`, and so on.

■ 3.1.3 Fourier Stability

MathPDE performs a Fourier stability test (see [6] for example) for simple explicit methods when the PDE problem is linear. To understand the underlying ideas, consider again the one-dimensional diffusion equation (3). If we use an explicit method described in Section 3.1.1, we get the stencil (equation (2)):

$$u(x, k+t) = \frac{u(x, t)\,h^2 - 2\,k\,u(x, t) + k\,u(x-h, t) + k\,u(h+x, t)}{h^2}. \tag{5}$$

To understand the behavior of the solution obtained by using this difference equation in the limit as $t \to \infty$, let us assume the solution for u is like a plane wave:

$$u(x, t) \sim e^{i\,(q\,x + t\,\omega)}, \tag{6}$$

where q and ω are the wave vector and frequency of the plane wave. We then obtain from equation (5),

$$e^{i\,\omega\,k} \sim \left(1 - 2\,k \big/ h^2\right) + \left(2\,k \big/ h^2\right)\cos(q\,h). \tag{7}$$

For the solution to be stable, it is necessary that the absolute value of the right-hand side of equation (7) is less than unity, or equivalently,

$$-1 \le \left(1 - 2\,k \big/ h^2\right) + \left(2\,k \big/ h^2\right)\cos(q\,h) \le 1. \tag{8}$$

From the inequality on the left, we get $k \big/ h^2 \le 1/(1-\cos(q\,h))$. Since the smallest value of $\cos(q\,h)$ is -1, we get

$$k \big/ h^2 \le 1/2 \tag{9}$$

as the criterion for the difference equation (5) to have a stable solution.

Based on this kind of analysis, *MathPDE* provides a function `stabineq[]` that returns the stability criterion for simple PDEs like diffusion and wave equations. This is used to help eliminate computational procedures that could be unstable when the step sizes chosen violate the criterion; in such cases, *MathPDE* selects an appropriate implicit scheme.

3.1.4 Higher-Degree Approximation

Finite-difference approximation of a derivative leads to truncation errors because the approximation is based on truncating the series expansion of a function to a certain number of terms determined by the approximation degree. To get a smaller truncation error, we can specify a higher-degree finite-difference approximation. We illustrate this with a simple example. Suppose we would like to solve the one-dimensional diffusion equation by using fourth-degree spatial discretization. The following sets up this problem.

```
SetupMathPDE[diffusion1D, {}, {{x, 4}}]
```

There is an important issue that arises in the case of a higher-degree FDA. Let us consider a fourth-degree central approximation of a spatial derivative.

```
diffApprox[u^(2,0)[x, t], 4, 3, h]
```

$$\frac{1}{h^2}\left(-\frac{1}{12}\,\mathrm{u}[-2+\mathrm{x},\,\mathrm{t}]+\frac{4}{3}\,\mathrm{u}[-1+\mathrm{x},\,\mathrm{t}]-\frac{5}{2}\,\mathrm{u}[\mathrm{x},\,\mathrm{t}]+\frac{4}{3}\,\mathrm{u}[1+\mathrm{x},\,\mathrm{t}]-\frac{1}{12}\,\mathrm{u}[2+\mathrm{x},\,\mathrm{t}]\right)$$

Note that this is a five-point scheme: it involves the grid points `x - 2`, `x - 1`, `x`, `x + 1`, and `x + 2`. Now suppose `x` varies from `1` to `N` on a grid. When `x = 2`, which is immediately next to the boundary `x = 1`, the above stencil would involve the point `-1` outside the grid. Therefore we use a *second*-degree FDA for the point `x = 2`, with the fourth-degree FDA taking over from `x = 3` onward. A similar device is used to eliminate the point `N + 1` at the other end. In general, the number of layers near the boundary where a lower-degree FDA is used depends on the degree of FDA desired.

3.1.5 Nonlinear Equations: Newton–Raphson Method

When the PDE is nonlinear, it is easy to see that the algebraic problem is nonlinear as well. Let us consider Burger's equation,

$$\frac{\partial u(x,t)}{\partial t}+\frac{\partial u(x,t)}{\partial x}\,u(x,t)=0. \tag{10}$$

When we discretize it using a difference method automatically selected by *MathPDE*, we get the following difference equation, which is nonlinear in u.

```
discretize[u^(0,1)[x, t] + u[x, t] u^(1,0)[x, t] == 0,
  {u[x[{1, 1}], t[{1, 1}]]}, {dx, dt}]
```

$$\frac{-\mathrm{u}[\mathrm{x},\,\mathrm{t}]+\mathrm{u}[\mathrm{x},\,1+\mathrm{t}]}{\mathrm{dt}}+\frac{\mathrm{u}[\mathrm{x},\,\mathrm{t}]\,(-\mathrm{u}[\mathrm{x},\,\mathrm{t}]+\mathrm{u}[1+\mathrm{x},\,\mathrm{t}])}{\mathrm{dx}}==0$$

We then have a system of N nonlinear algebraic equations to be solved at each point in time. *MathPDE* handles such a system using the multidimensional Newton–Raphson method (see [7]). We briefly summarize this method. Suppose we want to solve a system of N equations

$$f_i(x) = 0,\ i = 1, 2, \ldots, N, \tag{11}$$

where x is N-dimensional. The Newton–Raphson method is an iterative scheme in which the solution at the nth iteration is

$$x_n = x_{n-1} - G_n^{-1} F_n,\ n = 1, 2, 3, \ldots, \tag{12}$$

where $F_n^{\mathrm{T}} = (f_1(x_n), f_2(x_n), \ldots, f_N(x_n))$, and G_n is the Jacobian

$$\begin{pmatrix} f_1'(x_n) & f_1'(x_n) \ldots & f_1'(x_n) \\ f_2'(x_n) & f_2'(x_n) \ldots & f_2'(x_n) \\ \ldots & & \\ f_N'(x_n) & f_N'(x_n) \ldots & f_N'(x_n) \end{pmatrix} \tag{13}$$

When we attempt to perform the iteration equation (12), we must know the initial guess x_0. *MathPDE* uses the solution at the previous time step as the initial guess for iteratively solving the nonlinear system at every time instant. In particular, for $t = 1$, the initial conditions specified as part of the PDE problem are used as the initial guess. The iteration can be terminated when the difference between successive approximations x_n and x_{n-1} is less than a certain small number.

The numerical part involves a matrix problem for both linear and nonlinear PDEs: for the linear case, we have to solve a matrix equation of the kind $A \cdot x = B$; for the nonlinear case, we have to solve such a problem for each iteration. Further, it can be seen that the matrices involved are very sparse, the number of nonzero elements in each row being roughly equal to the degree of FDA. We can thus employ efficient numerical routines to handle sparse matrix systems.

■ 3.1.6 Solution of Sparse Matrix Systems

MathPDE uses the SuperLU numerical library for sparse matrix systems [13]. This optimized library is based on a variation of the Gaussian elimination algorithm adapted for sparse systems, and is actually a collection of three libraries: sequential, multithreaded, and distributed. *MathPDE* employs a few subroutines of sequential SuperLU to solve matrix equations of the kind $A \cdot X = B$ that we referred to in Section 3.1.5. The sequential library, implemented in C, supports real and complex data types in both single and double precision.

Here is the basic algorithm on which SuperLU is based, which is a sparse Gaussian elimination procedure to solve a matrix equation $A \cdot X = B$.

- Compute an LU decomposition of A, $P_r\, D_r\, A P_c\, D_c = L\, U$. Here, P_r and P_c are row and column permutation matrices and D_r and D_c are the so-called equilibration matrices, both diagonal. These four matrices are suitably chosen so as to enhance the sparsity of L and U, numerical stability, and parallelism. In a simple implementation offered in the subroutine `dgssv`, the equilibration matrices D_r and D_c are taken to be identity matrices.
- Once we have the LU decomposition, the solution vector X can be efficiently computed using $X = D_c\, P_c\, U^{-1}\, L^{-1}\, P_r\, D_r\, B$, multiplying from right to left.

The sparse matrix A must be provided in the Harwell–Boeing column-compressed format to save storage. A row-compressed format is also possible, that is, in which A^{T} is in column-compressed format, but involves some preprocessing to transform into the Harwell–Boeing format; this is followed in the data structure SuperMatrix. In this storage format, an $N \times N$ sparse matrix A, in which only nnz elements are nonzero, is specified in terms of three row vectors $a[0:\mathrm{nnz}-1]$, $\mathrm{asub}[0:\mathrm{nnz}-1]$, and $\mathrm{xa}[0:N]$ (that have, respectively, nnz, nnz, and $N+1$ elements).

- The successive elements of $a[0:\mathrm{nnz}-1]$ are obtained by sequentially going over the columns of A, running down each column, and picking the nonzero elements of A.
- The elements of $\mathrm{asub}[0:\mathrm{nnz}-1]$ are simply the row indices in A of the elements of $a[0:\mathrm{nnz}-1]$.
- The k^{th} element of $\mathrm{xa}[0:N]$, $\mathrm{xa}[k]$, for $1 \le k \le N$, is the total number of nonzero elements of A up to and including the k^{th} column, and $\mathrm{xa}[0] = 0$.

Let us illustrate this storage format for the following sparse matrix (taken from the SuperLU user guide [13]).

$$A = \begin{pmatrix} s & 0 & u & u & 0 \\ l & u & 0 & 0 & 0 \\ 0 & l & p & 0 & 0 \\ 0 & 0 & 0 & e & u \\ l & l & 0 & 0 & r \end{pmatrix}$$

For this choice of A, $N = 5$, $\mathrm{nnz} = 12$, and using the indexing conventions of C, $a = (s\, l\, l\, u\, l\, l\, u\, p\, u\, e\, u\, r)$, asub = (0 1 4 1 2 4 0 2 0 3 3 4), and xa = (0 3 6 8 10 12).

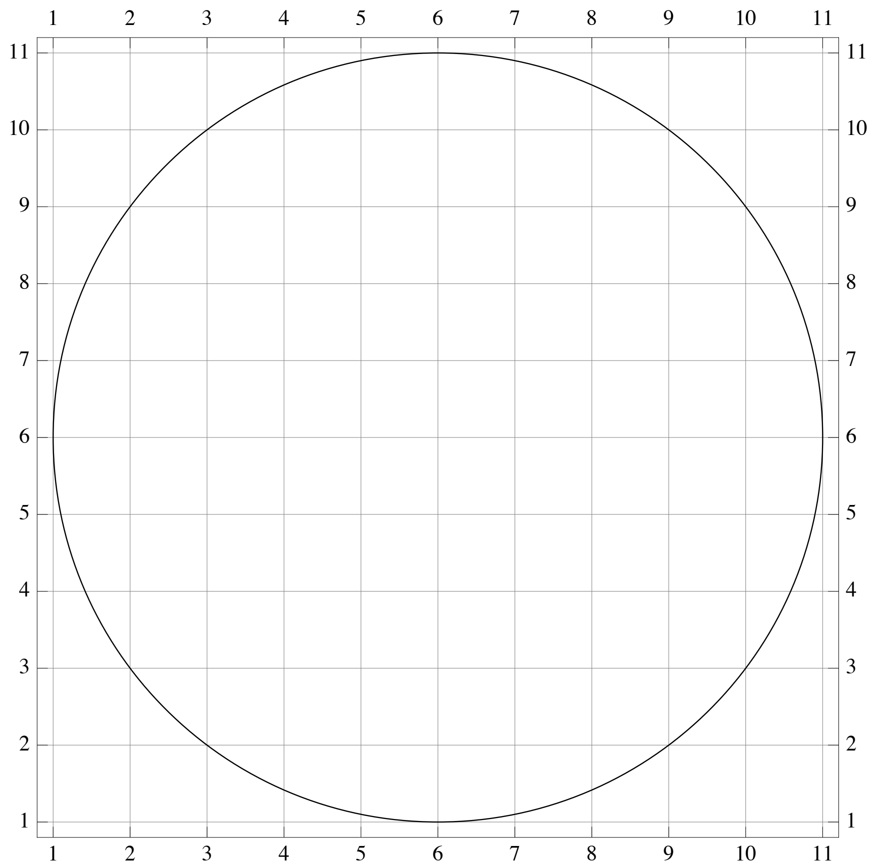
Matrices such as these, along with other information like N, nnz, and the matrix B (that can be specified as a simple row vector of length N) are required by the SuperLU subroutine `dgssv` that solves the matrix system $A \cdot X = B$.

◻ 3.2 Domains and Boundaries

■ *3.2.1 Domain Discretization*

An important aspect of *MathPDE* is its ability to handle a wide range of spatial domains. We explain the ideas with the example of a two-dimensional circular domain behind the way *MathPDE* discretizes domains. Let us therefore begin with a circular domain on which we overlay a grid with 11×11 grid points.

```
Graphics[{Circle[{6, 6}, 5]},
 GridLines → {Range[11], Range[11]}, Frame → True,
 FrameTicks → {Range[11], Range[11]}]
```



The domain is a circle of radius 1, centered at the origin (the point (6, 6) in the grid). The bounding box is a square with vertices at the points (1, 1), (11, 1), (11, 11), (1, 11).

In *MathPDE*, this domain is specified in the format `{{{x, -`$\sqrt{1 - y^2}$`,` $\sqrt{1 - y^2}$`}}, {{y, -1, 1}}}`. When we treat this domain descriptor as an iterator, it is clear that all the points inside the unit circle as well as those on its circumference are covered. Furthermore, only these points, and no other, are covered. In other words, the actual domain is a *subset* of the bounding box. For rectangular domains, the actual domain *is* the bounding box, whereas for other kinds of geometry, the actual domain is a *proper* subset of the bounding box. This is made possible because the limits of one of the independent variables, x, are treated as functions of the other independent variable, y.

For our example, here is the lower limit for x.

$$x_{\text{Left1}}[y] = -\sqrt{1 - y^2};$$

It is easy to discretize the domain now. Since we know the bounding box, we can obtain the discretized lower limit of x.

xLeft1[y_, Nx_, Ny_] :=

$$1 - \text{Round}\left[\frac{1}{2}(-1 + \text{Nx})\left(-1 + 2\sqrt{\text{Abs}\left[\frac{(\text{Ny} - y)(-1 + y)}{(-1 + \text{Ny})^2}\right]}\right)\right]$$

Here Nx and Ny are the number of grid points in the bounding box in the x and y directions, respectively. Arguing in the same manner, we have the following formula for the upper limit of x.

xRight1[y_, Nx_, Ny_] :=

$$1 + \text{Round}\left[\frac{1}{2}(-1 + \text{Nx})\left(1 + 2\sqrt{\text{Abs}\left[\frac{(\text{Ny} - y)(-1 + y)}{(-1 + \text{Ny})^2}\right]}\right)\right]$$

This is the overall algorithm we use for discretizing spatial domains. When the domain in question is nonrectangular, there are some issues that we need to be careful about. We now elaborate on some of these, continuing with the example of a circular domain.

Let us list the points lying on the left and right boundaries of the circle.

```
Table[{xLeft1[y, 11, 11], y}, {y, 1, 11}]
```

```
{{6, 1}, {3, 2}, {2, 3}, {1, 4}, {1, 5},
  {1, 6}, {1, 7}, {1, 8}, {2, 9}, {3, 10}, {6, 11}}
```

```
Table[{xRight1[y, 11, 11], y}, {y, 1, 11}]
```

```
{{6, 1}, {9, 2}, {10, 3}, {11, 4}, {11, 5},
  {11, 6}, {11, 7}, {11, 8}, {10, 9}, {9, 10}, {6, 11}}
```

Here, there is a danger if one is not careful. If the point `{6, 1}` alone is chosen as being on the circumference on the line `y == 1`, as is done above, then the lower `y` neighbor of `{5, 2}` (in the interior) is the point `{5, 1}`, which would be outside the domain. Therefore, it is necessary to keep all the points `{4, 1}`, `{5, 1}`, …, `{8, 1}` on the circumference. Here are improved versions of the definitions for `xLeft1` and `xRight1`.

```
xL1[y_, Nx_, Ny_] := Min[1 + xLeft1[-1 + y, Nx, Ny],
   xLeft1[y, Nx, Ny], 1 + xLeft1[1 + y, Nx, Ny]];

xR1[y_, Nx_, Ny_] := Max[-1 + xRight1[-1 + y, Nx, Ny],
   xRight1[y, Nx, Ny], -1 + xRight1[1 + y, Nx, Ny]];
```

We remark here that formulas `xL1` and `xR1` lead to extra computation compared with `xLeft1` and `xRight1`. However, since these computations are performed only on the domain boundaries, and not at all the points, the additional burden of this improvised algorithm is insignificant for large grids.

Let us once again look at the left and right boundaries of the circular domain.

```
Table[{xL1[y, 11, 11], y}, {y, 1, 11}]

{{4, 1}, {3, 2}, {2, 3}, {1, 4}, {1, 5},
 {1, 6}, {1, 7}, {1, 8}, {2, 9}, {3, 10}, {4, 11}}

Table[{xR1[y, 11, 11], y}, {y, 1, 11}]

{{8, 1}, {9, 2}, {10, 3}, {11, 4}, {11, 5},
 {11, 6}, {11, 7}, {11, 8}, {10, 9}, {9, 10}, {8, 11}}
```

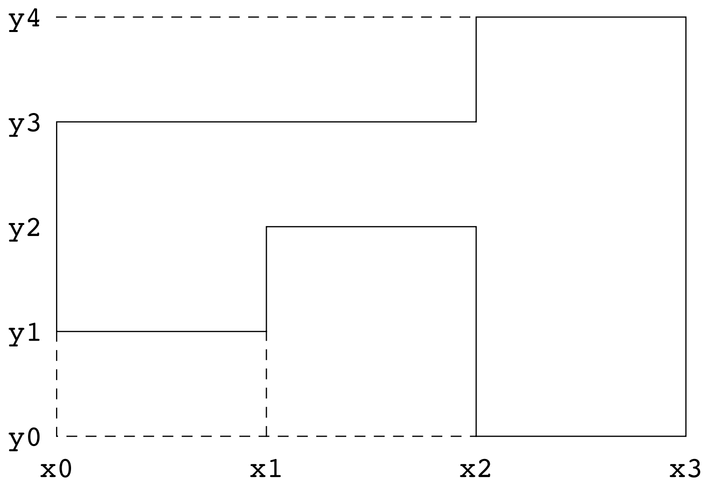
Although the differences are only on the lines `y = 1` and `y = Nx`, it is important that the neighbors of all interior points are now treated as being on the boundary, and not outside.

There is a shortcoming of this improvised algorithm: since the points `{4, 1}`, `{5, 1}`, …, `{8, 1}`are all treated as being on the circumference of the circular domain, the actual domain used in computations is therefore not exactly circular, but a distorted circle. This is a price we pay for describing nonrectangular domains using a Cartesian system of coordinates. It can be noted, however, that the extent of this distortion is negligible for large grids. In cases where such a distortion affects the quality of solution obtained, one needs to be more careful.

One last point we would like to discuss concerns the notion of *subdomains*. Above, we mentioned that the domain is specified as $\{\{\{x, -\sqrt{1-y^2}, \sqrt{1-y^2}\}\}, \{\{y, -1, 1\}\}\}$, so one could ask why this format is used instead of a simpler iterator like $\{\{x, -\sqrt{1-y^2}, \sqrt{1-y^2}\}\}, \{\{y, -1, 1\}\}$.

The reason is that we want to be able to describe more involved domains like this irregular domain (boundaries shown in solid lines).

```
Graphics[
 {Line[{{4, 0}, {6, 0}, {6, 4}, {4, 4}, {4, 3}, {0, 3},
    {0, 1}, {2, 1}, {2, 2}, {4, 2}, {4, 0}}], Dashed,
  Line[{{0, 1}, {0, 0}, {4, 0}}], Line[{{2, 0}, {2, 1}}],
  Line[{{0, 4}, {4, 4}}],
  Style[{Text["x0", {0, 0} - {0, .3}],
    Text["x1", {2, 0} - {0, .3}],
    Text["x2", {4, 0} - {0, .3}],
    Text["x3", {6, 0} - {0, .3}],
    Text["y0", {0, 0} - {.3, 0}],
    Text["y1", {0, 1} - {.3, 0}],
    Text["y2", {0, 2} - {.3, 0}],
    Text["y3", {0, 3} - {.3, 0}],
    Text["y4", {0, 4} - {.3, 0}]}, FontFamily → "Courier",
   12]
 }, ImageSize → {300, 200}]
```



This domain can be described using the following list, in which there are two sublists in the x part of the domain descriptor. We call these sublists *subdomains*. On the other hand, the y part has only one subdomain. It is clear that with this approach, we are able to describe and discretize a wide range of geometries, regular and irregular. The user needs to be able to describe the functional dependence (between independent variables) that defines the nature of geometry.

```
{{{x, If[And[y > y1, y < y3], x0, 1],
   If[And[y > y1, y < y3], x1, 0]},
  {x, If[And[y > y2, y < y3], x1, x2], x3}}, {{y, y0, y4}}}
```

One last point related to the notion of subdomains concerns the way to describe boundary conditions on a segment of the boundary. Suppose we want to apply a Dirichlet boundary condition `u[x, y, t] == c1` on the line joining the points `{x 1, y 2}` and `{x 2, y 2}`, which is a segment of the boundary of the domain shown. This is done by including a boundary condition like `{{{{x, x1, x2}}, {{y, y2, y2}}}, u[x, y, t] == c1}` in the problem list. The iterator corresponding to the boundary above generates the grid points that lie on this boundary segment.

3.2.2 Domain Decomposition: Boundary and Interior Regions

We now discuss the issue of decomposition of the domain into boundary and interior regions, and how to apply the discretized boundary conditions and the difference equations in the respective regions. Let us consider the example of a two-dimensional diffusion equation in a circular domain to illustrate the ideas.

```
diffusion2dcircle = {
   {
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]},
    {x == -Sqrt[1 - y^2], u[x, y, t] == 0.},
    {x == Sqrt[1 - y^2], u[x, y, t] == 0.},
    {t == 0, u[x, y, t] ==
      If[x^2 + y^2 < 0.5, 1. - x^2 - y^2, 0]}
   },
   {
    {{x, -Sqrt[1 - y^2], Sqrt[1 - y^2]}}, {{y, -1, 1}}
   }
  };
```

The domain `{{{x, -Sqrt[1 - y^2], Sqrt[1 - y^2]}}, {{y, -1, 1}}}` describes a circle that we discussed in the previous subsection. When discretized, it becomes `{{{x, xL1[y, Nx, Ny], xR1[y, Nx, Ny]}}, {{y, 1, Ny}}}`, where the functions `xL1` and `xR1` were defined earlier. Having discretized the domain, there now remains the problem of decomposing it into boundary and interior regions, where the boundary conditions and the difference equations (corresponding to the PDE) must be applied. We now describe the way in which this is done in *MathPDE*.

Let us begin by examining the boundary conditions. In the present example, we have two Dirichlet boundary conditions for `u[x, y, t] = 0` at the left and right branches of the circular domain, `x = xL1[y, Nx, Ny]` and `x = xR1[y, Nx, Ny]`.

Clearly, the interior region begins with the point `x = xL1[y, Nx, Ny] + 1` and ends with the point `x = xR1[y, Nx, Ny] - 1` at each value of y. Furthermore, the points on the circumference for $y = 1$, namely, $\{\{4, 1\}, \{5, 1\}, \{6, 1\}, \{7, 1\}, \{8, 1\}\}$ (for $Nx = Ny = 11$), as well as the points on the circumference for $y = Ny$, $\{\{4, 11\}, \{5, 11\}, \{6, 11\}, \{7, 11\}, \{8, 11\}\}$, must be part of the boundary. As a result, the boundaries are described by `list1` and `list2` and the interior region described by `list3`.

```
list1 = {{{x, xL1[y, Nx, Ny], xL1[y, Nx, Ny]}}, {{y, 1, Ny}}};

list2 = {{{x, xR1[y, Nx, Ny], xR1[y, Nx, Ny]}}, {{y, 1, Ny}}};

list3 = {{{x, xL1[y, Nx, Ny] + 1, xR1[y, Nx, Ny] - 1}},
   {{y, 2, Ny - 1}}};
```

This is the overall domain decomposition algorithm that *MathPDE* implements. We now elaborate on a few issues that arise in connection with matching the difference equations and the boundary conditions at the boundary.

When the lowest-degree difference approximation is used to generate the difference equations from the PDE (see Section 3.1.1), the domain for the circle above determines the set of grid points at which the difference equations apply. However, when we use a higher-degree difference approximation, matching the difference equations with the boundary conditions is a tricky issue. As we have already mentioned in Section 3.1.4, we use the lowest-degree stencil on a few grid layers near the boundary, with the higher-degree stencil taking over beyond this region into the interior. As a result, the interior region for the circle must be further decomposed into two subregions: a few layers near the boundary where the lowest-degree difference equations apply, and the rest of the interior region where higher-degree equations apply.

For example, here is the difference equation when a fourth-degree spatial scheme is used for the two-dimensional diffusion equation.

$$\frac{-u[x, y, t] + u[x, y, 1+t]}{dt} == \frac{1}{dy^2}\left(-\frac{1}{12}u[x, -2+y, t] + \frac{4}{3}u[x, -1+y, t] - \frac{5}{2}u[x, y, t] + \frac{4}{3}u[x, 1+y, t] - \frac{1}{12}u[x, 2+y, t]\right) + \frac{1}{dx^2}\left(-\frac{1}{12}u[-2+x, y, t] + \frac{4}{3}u[-1+x, y, t] - \frac{5}{2}u[x, y, t] + \frac{4}{3}u[1+x, y, t] - \frac{1}{12}u[2+x, y, t]\right)$$

This equation has the `y` values `{y - 2, y - 1, y, y + 1, y + 2}`. Since the lowest grid value of `y` is `1`, it is clear that this equation can be applied only for `y` beginning with `y = 3`. Similarly, the highest grid value of `y` at which the equation can be applied is `Ny - 2`. We can argue in the same way for the `x` limits, and as a result, the interior region where the fourth-degree stencil applies is `stencil4`.

```
stencil4 = {{{x, xL1[y, Nx, Ny] + 2, xR1[y, Nx, Ny] - 2}},
  {{y, 3, Ny - 2}}}
```

The region where the lowest-degree (i.e., second-degree in this case) stencil applies is this set of four regions, which is the difference between the regions `list3` and `stencil4`.

```
{{{y, 3, -2 + Ny}, {x, xL1[y, Nx, Ny] + 1, xL1[y, Nx, Ny] + 1}},
 {{y, 3, -2 + Ny}, {x, xR1[y, Nx, Ny] - 1, xR1[y, Nx, Ny] - 1}},
 {{y, 2, 2}, {x, xL1[y, Nx, Ny] + 1, xR1[y, Nx, Ny] - 1}},
 {{y, -1 + Ny, -1 + Ny},
  {x, xL1[y, Nx, Ny] + 1, xR1[y, Nx, Ny] - 1}}}
```

Finally, we comment on a correction required in the definitions of `xL1` and `xR1` that describe the boundary region. The left and right boundaries that make up the circular domain meet on the lines `y = 1` and `y = Ny`. As we argued in Section 3.2.1, the boundary intersects each of these lines at a *set* of points, and not just one point. As a result, all the points from `xL1[1, Nx, Ny]` to `xR1[1, Nx, Ny]` must be included in the boundary region; similarly, all the points from `xL1[Ny, Nx, Ny]` to `xR1[Ny, Nx, Ny]` must also be included in the boundary region. The original definitions of `xL1` and `xR1` do not include all these points, and we must therefore replace them by the set of regions that correctly describes the boundary region.

```
{
 {{x, xL1[y, Nx, Ny], xL1[y, Nx, Ny]}, {y, 2, Ny - 1}},
 {{x, xL1[y, Nx, Ny], xR1[y, Nx, Ny] - 1}, {y, 1, 1}},
 {{x, xL1[y, Nx, Ny], xR1[y, Nx, Ny] - 1}, {y, Ny, Ny}},
 {{x, xR1[y, Nx, Ny], xR1[y, Nx, Ny]}, {y, 1, Ny}},
}
```

This list also ensures that points where two boundaries intersect are included in only one of them, and not counted twice. The domain decomposition algorithm implemented by *MathPDE* takes this into account.

□ 3.3 Code Generation Algorithms

■ 3.3.1 Automatic Function Generation

In the previous two sections, 3.1 and 3.2, we discussed at length the numerical and domain-related algorithms that *MathPDE* implements. A third important component of *MathPDE* is its ability to generate a problem-specific program by stitching together the numerical and domain parts in an appropriate manner. This program is a set of independent and interrelated *Mathematica* functions that work to compute a numerical solution of the PDE problem. Each function performs a very specific task.

For example, consider the function `xL1[]` that we discussed in Section 3.2.1, which computes the lower limit of the iterator for `x` in a circular domain. This is one of the many functions that are automatically generated by *MathPDE* for the PDE problem `diffusion2dcircle`. The way we do this is by manipulating the `DownValues` of the function. Since `DownValues[xL1]` is a list of elements of the form `HoldPattern[lhs] :→ rhs`, we can define `lhs` and `rhs` suitably, and generate a definition of the function `xL1`. Let us briefly explain how we do it.

We first generate the list `deflist` after creating a context ``MathPDE` ``, whose elements are of the form {`head, lhs` → `rhs`}.

```
BeginPackage["MathPDE`"];

deflist =
  {{xL1, xL1[y, Nx, Ny] → Min[1 + xLeft1[-1 + y, Nx, Ny],
      xLeft1[y, Nx, Ny], 1 + xLeft1[1 + y, Nx, Ny]]},
   {xLeft1, xLeft1[y, Nx, Ny] →
```

$$1 - \text{Round}\left[\frac{1}{2}\,(-1+\text{Nx})\left(-1+2\sqrt{\frac{(\text{Ny}-\text{y})\,(-1+\text{y})}{(-1+\text{Ny})^2}}\right)\right]\}\};$$

This list is generated based on the domain-discretization algorithm as explained in Section 3.2.1. Here, `lhs` contains information about the input syntax (the function prototype, without type information) of the function `head`, and `rhs` contains what is going to be the body of the function definition. Note that the `Rule` format ensures delayed evaluation. Once we have a list like `deflist`, it is easy to generate function definitions. We can, for instance, define a function-defining function called `DefineFunction` that can generate the definition of a function based on information like `deflist`.

```
DefineFunction[head_, rule_] :=
 Module[{pat},
  DownValues[head] = {};
  pat = Map[ToExpression[StringJoin[ToString[#1], "_"]] &,
    rule[[1]]];
  DownValues[head] =
   {RuleDelayed @@ {HoldPattern @@ {pat}, rule[[2]]}};
 ]

Off[General::"spell1"];
Attributes[DefineFunction] = {HoldAll};
DefineFunction[head_, rule_] :=
 Module[{pat, min, max, rhs},
  DownValues[head] = {};
  pat = Map[ToExpression[StringJoin[ToString[#1], "_"]] &,
    rule[[1]]];
  rhs = rule[[2]] /. Min :→ min /. Max :→ max;
  DownValues[head] =
   {(RuleDelayed @@ {HoldPattern @@ {pat},
          rhs /. min[a_, b_, c_] :→ min[{a, b, c}] /.
           max[a_, b_, c_] :→ max[{a, b, c}]}) /. min :→ Min /.
     max :→ Max};
 ]
```

Now, we can execute the following commands.

```
Begin["MathPDE`Private`"];

Map[DefineFunction @@ # &, deflist];
```

After which, the functions `xL1` and `xLeft1` will be defined.

```
Map[Information, {xL1, xLeft1}]
```

```
MathPDE`xL1
```

```
xL1[y_, Nx_, Ny_] := Min[{1 + xLeft1[-1 + y, Nx, Ny], xLeft1[y, Nx, Ny]
```

```
MathPDE`xLeft1
```

$$\texttt{xLeft1[y\_, Nx\_, Ny\_] := 1 - Round}\left[\frac{1}{2}\,(-1+\texttt{Nx})\left(-1+2\sqrt{\frac{(\texttt{Ny}-\texttt{y})\,(-1+\texttt{y})}{(-1+\texttt{Ny})^2}}\right)\right]$$

```
End[];
EndPackage[];
```

We note one more thing here. Since the body of `xL1` depends explicitly on another function, `xLeft1`, it is important that the latter be still undefined when the former is defined. The sequence of function generations must therefore be arranged with proper regard for their interdependence. One more thing to note is that the body of each of the functions automatically generated must involve purely numerical operations and no symbolic ones. This is important, since we are finally interested in translating the program into a *compiled* language (like C++ or Fortran 90) by employing the code generator *MathCode* [14].

Some of the functions automatically generated do not have such a simple structure as `xL1`. These include, for instance, the function `SolvePDE` that we encountered in Section 2. Such functions perform more complicated operations, and involve local variables that must be declared in a `Module`. Such definitions are generated using special-purpose functions available in *MathPDE*, and functions like `DefineFunction` above will not do. However, the essential idea is still the same, and is based on delayed evaluation and the use of `DownValues`.

3.3.2 Translation into C++ Using MathCode

MathCode is a system for translating *Mathematica* functions into C++/Fortran 90 [14]. The functions must be purely numerical, as we mentioned in Section 3.3.1. We must declare the prototype information of the functions for *MathCode* to translate them. The numerical functions that *MathPDE* generates are then passed on to *MathCode*, which in turn translates them into C++ or Fortran 90.

The reason for translating the *Mathematica* functions is twofold. Firstly, it lends portability to the solver: the C++ code that is generated is completely self-contained. Secondly, *MathCode* enables us to run the generated code from the notebook as well. Typically, this leads to performance gains of several times compared with original *Mathematica* code.

We now illustrate the way *MathPDE* employs *MathCode* to generate C++ code.

```
Needs["MathCode`"];
```

```
MathCode version   1.2 loaded.
```

```
SetDirectory[$MCRoot <> "/Demos/SimplestExample"];
```

We have to start the context `MathPDE``, and mention `MathCodeContexts` within the path of the package.

```
BeginPackage["MathPDE`", {MathCodeContexts}];
```

However, since the functions `xL1` and `xLeft1` have already been defined, we simply end the package.

```
EndPackage[];
```

We next declare the function prototypes. This specifies the data types of all the arguments and the output. In our simple example, all data types involved are integers.

```
Declare[xL1[Integer y_, Integer Nx_, Integer Ny_] → Integer];
```

```
Declare[xLeft1[Integer y_, Integer Nx_, Integer Ny_] →
   Integer];
```

We then build the C++ code for the functions `xL1` and `xLeft1` with the following command.

```
BuildCode["MathPDE`"];
```

```
Successful compilation to C++: 2 function(s)
```

The next command runs the *Mathematica* code for the function `xL1`.

```
Timing[Do[xL1[2, 11, 11], {1000}]]
```

```
{0.691 Second, Null}
```

If we want to run the C++ executable instead, we must install it.

```
InstallCode[];
```

```
MathPDE is installed.
```

Now this runs the C++ executable.

```
Timing[Do[xL1[2, 11, 11], {1000}]]
```

```
{0.35 Second, Null}
```

It can be seen that there is a slight enhancement in speed; however, the enhancement factor depends on the problem. Here is the C++ code generated.

```
!! MathPDE.cc
```

```
#include "MathPDE.h"

#include "MathPDE.icc"

#include <math.h>
void MathPDE_TMathPDEInit ()
{
;
}

int MathPDE_TxL1 ( const int &y, const int &Nx, const int &Ny)
{
    return LightMin( make_lightN(3, 1+MathPDE_TxLeft1 (-1+y,
Nx, Ny),
        MathPDE_TxLeft1 (y, Nx, Ny), 1+MathPDE_TxLeft1 (1+y,
Nx, Ny) ));
}

int MathPDE_TxLeft1 ( const int &y, const int &Nx, const int
&Ny)
{
    return 1+-irint(((-1+Nx)*(-1+2*pow(pow(-1+Ny, -2)*(Ny+-
y)*(-1+y),
                               0.5)))/2);
}
```

We compiled just two functions, `xL1` and `xLeft1`. *MathPDE* automates this sequence of commands and generates, compiles, and installs code for all the numerical functions automatically generated for the given PDE problem (nine functions, for the example of the one-dimensional diffusion equation; see Section 2.2). The resulting code computes the solution to the PDE, as demonstrated in Section 2.3.

■ 4. Examples

This section presents a range of example PDE problems solved using *MathPDE*: the examples are chosen to illustrate the many features of *MathPDE*; for example, higher-degree approximation schemes, nonlinear problems, different kinds of boundary condition, non-rectangular geometries, and so on. All the problems are time-dependent PDE problems, mainly of the parabolic and hyperbolic types.

We now solve a variety of PDE problems using *MathPDE*. The solution steps involved are much like in Section 2, and are obvious from the context. If *MathPDE* is installed on your system, all the commands should work when executed in the sequence they are given in below.

□ 4.1 One-Dimensional Problems

■ 4.1.1 Diffusion Equation with Derivative Boundary Conditions

We must first load the package *MathPDE*.

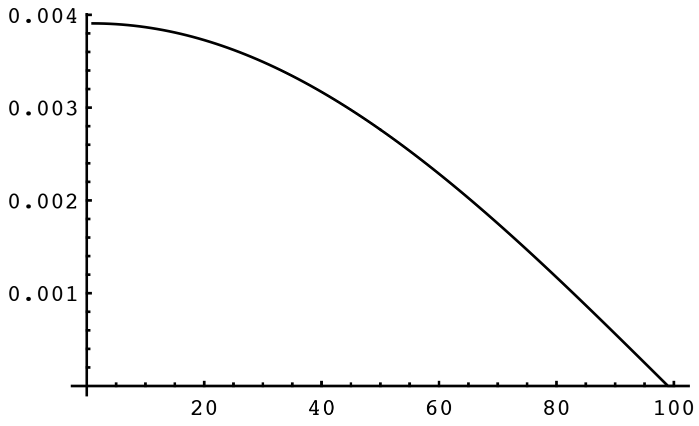
```
Needs["MathPDE`"];

diffusion2 = {
    {
     {∂{t,1} u[x, t] == ∂{x,2} u[x, t]},
     {x == 0, ∂{x,1} u[x, t] == 0.},
     {x == 1, u[x, t] == 0},
     {t == 0, u[x, t] == If[x < .15, x, 1 - x]}
    },
    {
     {{x, 0, 1}}
    }
   };

SetupMathPDE[diffusion2, {}]
```

To save space here as well as in the rest of Section 4, we have cut out the detailed comments returned by *MathPDE* when the function `SetupMathPDE` is executed.

```
AbsTime[SolvePDE[100, 0.01, 1, 200];]
```

```
{1.0715408 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```



The difference here from the example presented in Section 4 is that the derivative at the left end of the system is zero, which leads to a different solution profile. In both cases, however, the solutions decay to a uniform concentration distribution in the long time limit.

We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

■ 4.1.2 Wave Equation

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];
```

```
wave = {
   {
    {∂{t,2} u[x, t] == (1 / c^2) ∂{x,2} u[x, t]},
    {x == 0, u[x, t] == 0},
    {x == 1, u[x, t] == 0},
    {t == 0, u[x, t] == x (1 - x), ∂{t,1} u[x, t] == der1}
   },
   {
    {{x, 0, 1}}
   }
  };
```

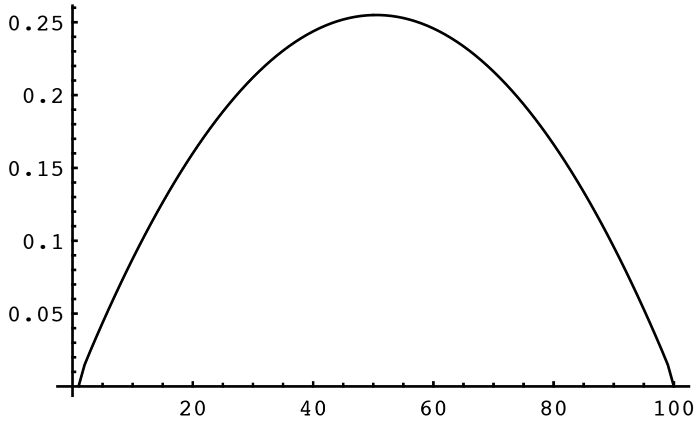
```
SetupMathPDE[wave, {c, der1}]
```

In this case, we can choose numerical values for `c` and `der1` at run time; the code generated for `SolvePDE` is such that these parameters can be passed as real arguments when we execute the program. We now run `SolvePDE` by choosing numerical values for these parameters.

```
AbsTime[SolvePDE[100, 0.01, 1000., 0.001, 1, 500];]
```

```
{1.5322032 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```



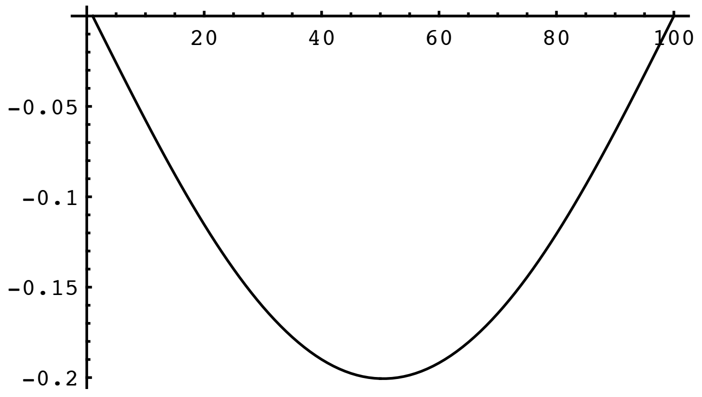
We can choose a different set of values for the parameters of the problem.

```
AbsTime[SolvePDE[100, 0.01, 1., 0.001, 1, 500];]
```

```
{2.4034560 Second, Null}
```

The solution is now different.

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```



We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

□ 4.2 Two-Dimensional Problems

■ 4.2.1 Diffusion Equation in a Rectangular Domain

We now take the example of a two-dimensional diffusion equation and compare *MathPDE* and `NDSolve`, the built-in *Mathematica* function to solve time-dependent PDE problems. We take the initial condition that is a Gaussian with its peak at the center of the domain, and Dirichlet boundary conditions.

`NDSolve` uses the numerical method of lines, while *MathPDE* uses a finite-difference time stepping. This could partly explain the differences in the solutions obtained by the two approaches.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];
```

```
diffusion2d = {
   {
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]},
    {x == 0, u[x, y, t] == 0},
    {x == 1, u[x, y, t] == 0},
    {y == 0, u[x, y, t] == 0},
    {y == 1, u[x, y, t] == 0},
    {t == 0, u[x, y, t] ==
      Exp[- ((x - 0.5) ^2 + (y - 0.5) ^2) / 0.02]}
   },
   {
    {{x, 0, 1}}, {{y, 0, 1}}
   }
  };
```

Let us set up the problem in *MathPDE*.

```
AbsTime[SetupMathPDE[diffusion2d, {}]]
```

```
{148.7438832 Second, Null}
```

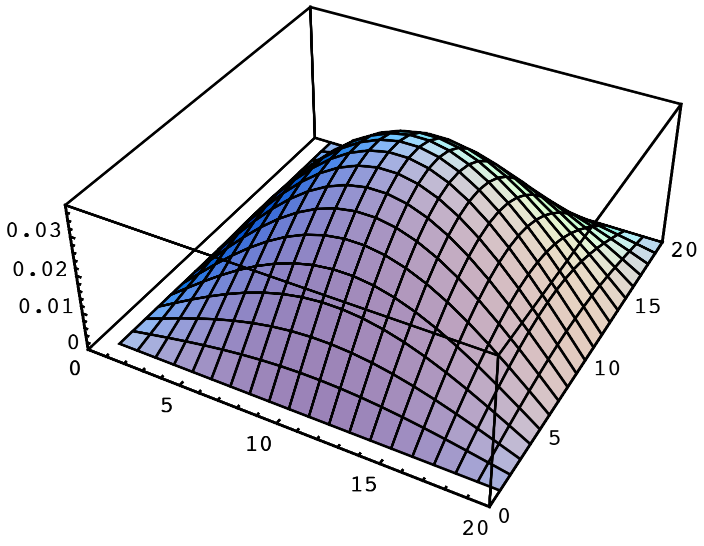
Now let us evolve the solution by 10 time steps.

```
{tot1, tot2} = {20, 20};
AbsTime[SolvePDE[tot1, tot2, 0.01, 1, 10];]
```

```
{1.9327792 Second, Null}
```

We plot the solution.

```
ListPlot3D[
 list = Table[solutionAt[i, j, 1, tot1, tot2], {i, 1, tot1},
   {j, 1, tot2}],
 PlotRange → {{0, tot1}, {0, tot2},
   {0, 0.0378126296236160`}}]
```



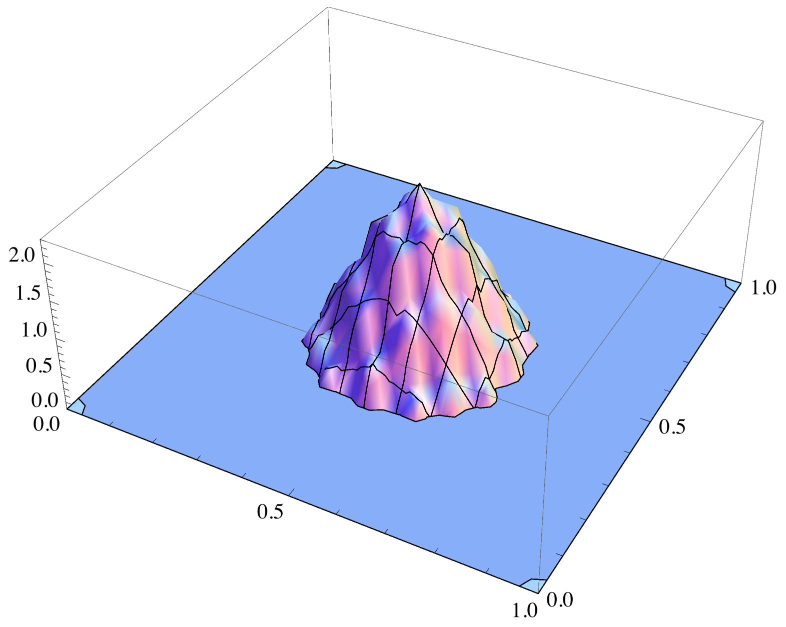
We can try this example using `NDSolve` as well. Let us choose the same step sizes as chosen above for *MathPDE* so as to compare the two solutions.

```
Timing[
 soln =
  Quiet@
   NDSolve[
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t],
     u[0, y, t] == 0, u[x, 0, t] == 0, u[1, y, t] == 0,
     u[x, 1, t] == 0,
     u[x, y, 0] == Exp[- ((x - 0.5) ^2 + (y - 0.5) ^2) / 0.02]},
    u, {x, 0, 1}, {y, 0, 1}, {t, 0, 0.5},
    MaxSteps → {20, 20, 50}]]
```

```
{0.132172, {{u → InterpolatingFunction[
     {{0., 1.}, {0., 1.}, {0., 0.0185968}}, <>]}}}
```

We plot the solution at `t = 0.1`.

```
Quiet@Plot3D[(u[x, y, 0.1] /.soln)⟦1⟧, {x, 0, 1},
   {y, 0, 1}, PlotRange → {{0, 1}, {0, 1}, {0, 2.3`}}]
```



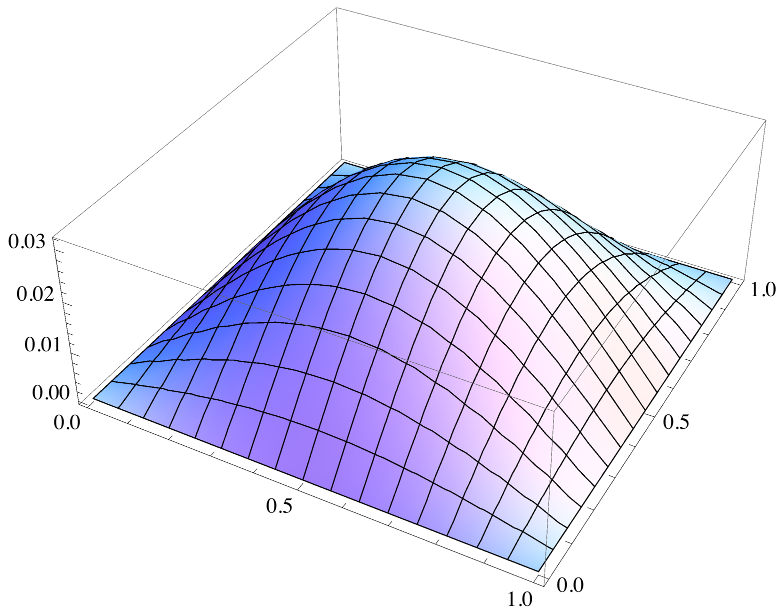
The solution has not homogenized yet, although the solution obtained using *MathPDE* was more spread out. However, suppose we let `NDSolve` automatically choose its step sizes.

```
Timing[
 soln =
  Quiet@
   NDSolve[
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t],
     u[0, y, t] == 0, u[x, 0, t] == 0, u[1, y, t] == 0,
     u[x, 1, t] == 0,
     u[x, y, 0] == Exp[-((x - 0.5)^2 + (y - 0.5)^2) / 0.02]},
    u, {x, 0, 1}, {y, 0, 1}, {t, 0, 5}]]
```

```
{9.51483, {{u → InterpolatingFunction[
      {{0., 1.}, {0., 1.}, {0., 5.}}, <>]}}}
```

We now plot the solution profile at `t = 0.1`.

```
Plot3D[(u[x, y, 0.1] /.soln)⟦1⟧, {x, 0, 1}, {y, 0, 1}]
```



Now the solution is more spread out, and agrees with the solution produced by *MathPDE*.

We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

■ 4.2.2 Diffusion Equation on a Circle

We now consider an example PDE problem with a circular domain; apart from that, the problem is essentially the same as in the previous example. This is a problem that `NDSolve` cannot handle, since the spatial geometry is not rectangular.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];

diffusion2dcircle = {
   {
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]},
    {x == -Sqrt[1 - y^2], u[x, y, t] == 0.},
    {x == Sqrt[1 - y^2], u[x, y, t] == 0.},
    {t == 0, u[x, y, t] ==
      Exp[-((x - 0.5)^2 + (y - 0.5)^2) / 0.02]}
   },
   {
    {{x, -Sqrt[1 - y^2], Sqrt[1 - y^2]}}, {{y, -1, 1}}
   }
  };
```

The boundary conditions are specified on the left and right semicircles. Since we describe a nonrectangular domain using Cartesian coordinates, one of the coordinates, `x`, has been made to depend on the other, `y`, in the domain description.

We now set up the problem using *MathPDE*.

```
AbsTime[SetupMathPDE[diffusion2dcircle, {}];]

{184.9960112 Second, Null}
```

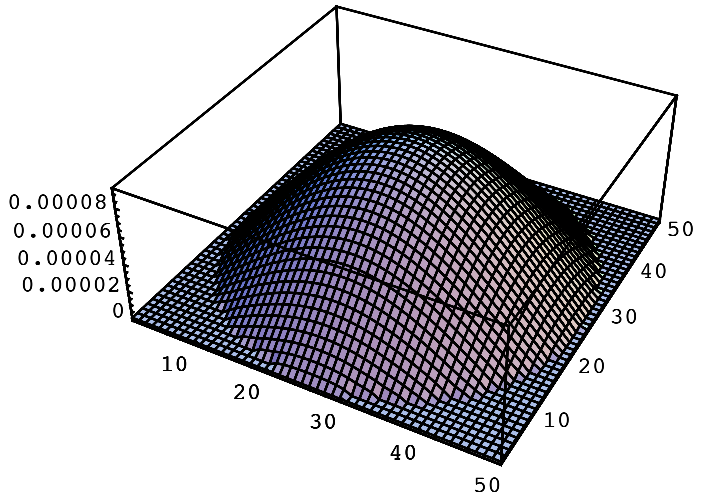
In this case *MathPDE* has generated 21 functions (as opposed to 13 in the previous example) that are translated into C++ code and compiled by *MathCode*. The extra functions in this case are required to compute the grid for the circular domain.

We compute the solution and plot it.

```
{tot1, tot2} = {50, 50};
AbsTime[SolvePDE[tot1, tot2, 0.01, 1, 100];]
```

```
{343.4338336 Second, Null}
```

```
ListPlot3D[Table[solutionAt[i, j, 1, tot1, tot2],
  {i, 1, tot1}, {j, 1, tot2}]]
```



Although we use Cartesian coordinates to describe a circular domain, the jagged nature of the circular boundary is not predominant at this level of granularity (50 grid steps in each of the two directions), and the solution appears fairly smooth.

We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

■ 4.2.3 A Two-Variable Problem: Wave Equation in a Circular Domain

We now study the example of the two-dimensional wave equation, in which there are two dependent variables. Let us first consider a rectangular domain.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];

wave2drectangle = {
   {
    {(1 / c^2) ∂{t,2} u[x, y, t] ==
      A1 (∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]) +
       A2 (∂{x,1} (∂{x,1} u[x, y, t] + ∂{y,1} v[x, y, t])),
     (1 / c^2) ∂{t,2} v[x, y, t] ==
      A1 (∂{x,2} v[x, y, t] + ∂{y,2} v[x, y, t]) +
       A2 (∂{y,1} (∂{x,1} u[x, y, t] + ∂{y,1} v[x, y, t]))},
    {x == 0, u[x, y, t] == 0, v[x, y, t] == 0},
    {x == 1, u[x, y, t] == 0, v[x, y, t] == 0},
    {y == 0, u[x, y, t] == 0, v[x, y, t] == 0},
    {y == 1, u[x, y, t] == 0, v[x, y, t] == 0},
    {t == 0, u[x, y, t] == (1 - x) x (1 - y) y,
     ∂{t,1} u[x, y, t] == 0.1, v[x, y, t] == (1 - x) x (1 - y) y,
     ∂{t,1} v[x, y, t] == 0.2}
   },
   {
    {{x, 0, 1}}, {{y, 0, 1}}
   }
  };
```

In this problem, there are some parameters, and so we have to input them in a list as an argument to the setup function.

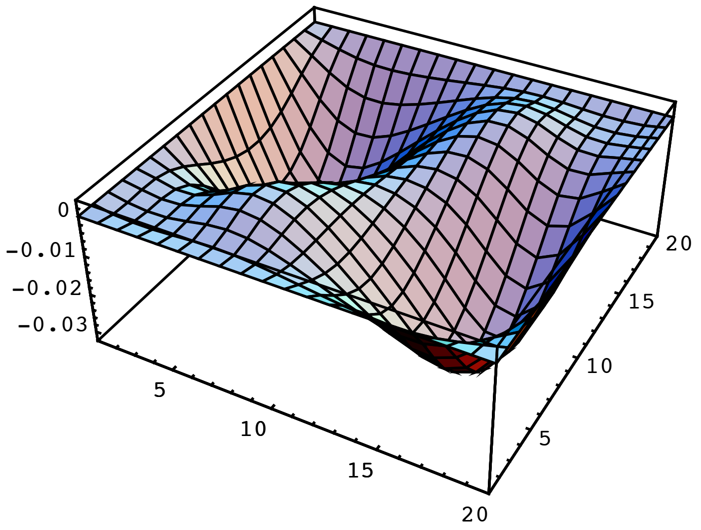
```
AbsTime[SetupMathPDE[wave2drectangle, {c, A1, A2}];]

{159.6295360 Second, Null}
```

Let us now compute the solution for a set of values of the parameters.

```
{tot1, tot2} = {20, 20};
AbsTime[SolvePDE[tot1, tot2, 0.1, 1., .1, 0.5, 1, 10];]

{72.3940976 Second, Null}
```

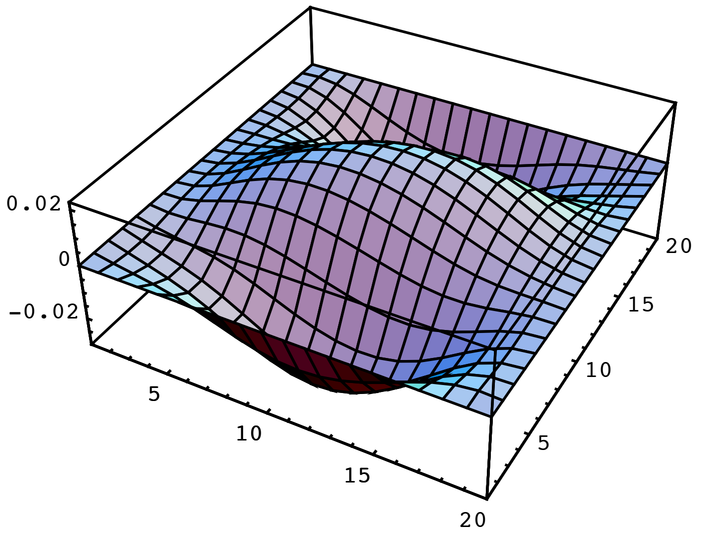
Here is the plot of the dependent variable u.

```
ListPlot3D[Table[solutionAt[i, j, 1, tot1, tot2],
  {i, 1, tot1}, {j, 1, tot2}]]
```



And here is the plot of the dependent variable v.

```
ListPlot3D[Table[solutionAt[i, j, 2, tot1, tot2],
  {i, 1, tot1}, {j, 1, tot2}]]
```



We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];

CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

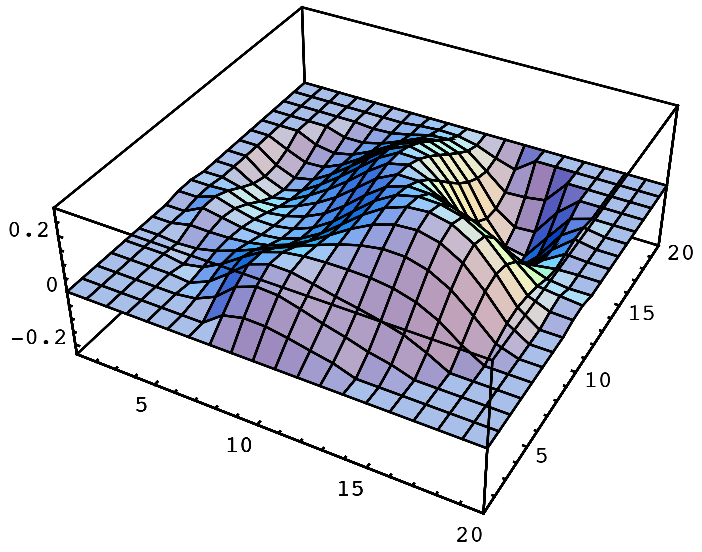
Now consider a similar problem in a circular domain.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];

wave2d2circle = {
   {
    {(1 / c^2) ∂{t,2} u[x, y, t] ==
      A1 (∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]) +
       A2 (∂{x,1} (∂{x,1} u[x, y, t] + ∂{y,1} v[x, y, t])),
     (1 / c^2) ∂{t,2} v[x, y, t] ==
      A1 (∂{x,2} v[x, y, t] + ∂{y,2} v[x, y, t]) +
       A2 (∂{y,1} (∂{x,1} u[x, y, t] + ∂{y,1} v[x, y, t]))},
    {x == -Sqrt[1 - y^2], u[x, y, t] == 0, v[x, y, t] == 0},
    {x == Sqrt[1 - y^2], u[x, y, t] == 0, v[x, y, t] == 0},
    {t == 0, u[x, y, t] == Exp[- ((x - 0.4) ^2 + (y - 0.4) ^2) / 0.1],
     ∂{t,1} u[x, y, t] == 0.1,
     v[x, y, t] == Exp[- ((x + 0.4) ^2 + (y + 0.4) ^2) / 0.1],
     ∂{t,1} v[x, y, t] == 0.2}
   },
   {
    {{x, -Sqrt[1 - y^2], Sqrt[1 - y^2]}}, {{y, -1, 1}}
   }
  };

AbsTime[SetupMathPDE[wave2d2circle, {c, A1, A2}];]

{162.5937984 Second, Null}

{tot1, tot2} = {20, 20};
AbsTime[SolvePDE[tot1, tot2, 0.1, 1., .1, 0.5, 1, 10];]

{29.2420480 Second, Null}
```

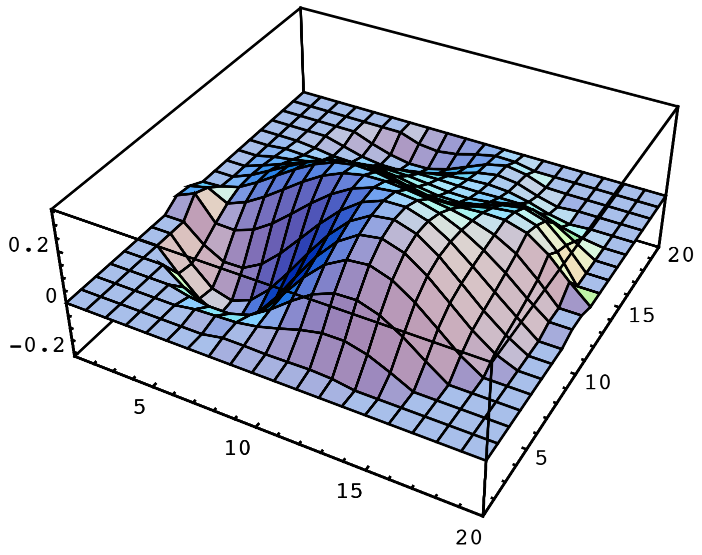
Here is the plot of the dependent variable `u`.

```
ListPlot3D[Table[solutionAt[i, j, 1, tot1, tot2],
  {i, 1, tot1}, {j, 1, tot2}],
 PlotRange → {{1, 20}, {1, 20}, Automatic}]
```



And here is the plot of the dependent variable `v`.

```
ListPlot3D[Table[solutionAt[i, j, 2, tot1, tot2],
  {i, 1, tot1}, {j, 1, tot2}],
 PlotRange → {{1, 20}, {1, 20}, Automatic}]
```



We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];

CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

□ 4.3 A Three-Dimensional Problem

■ 4.3.1 Three-Dimensional Wave Equation

Let us now consider the wave equation

$$\frac{1}{c^2}\,\frac{\partial^2\Phi}{\partial t^2} = A_1\,\nabla^2\Phi + A_2\,\nabla(\nabla.\Phi)$$

for the spatial and temporal variations of $\Phi = (u, v, w)$, the three-dimensional vector of displacements in a solid. Here, c is the speed of sound in the material, and A_1 and A_2 are constants that depend on the Poisson ratio. This equation arises in many contexts, for example when we have a shaft that rests on ball bearings and is connected to wheels at the ends. We can describe this problem in a cubic geometry in the form of the following list, in which we have expanded the vector equation into components.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];

wave3d =
  {
   {
    {(1 / c^2) ∂{t,2} u[x, y, z, t] ==
      A1 (∂{x,2} u[x, y, z, t] + ∂{y,2} u[x, y, z, t] +
          ∂{z,2} u[x, y, z, t]) +
       A2
        (∂{x,1} (∂{x,1} u[x, y, z, t] + ∂{y,1} v[x, y, z, t] +
             ∂{z,1} w[x, y, z, t])),
     (1 / c^2) ∂{t,2} v[x, y, z, t] ==
      A1 (∂{x,2} v[x, y, z, t] + ∂{y,2} v[x, y, z, t] +
          ∂{z,2} v[x, y, z, t]) +
     A2
      (∂{y,1} (∂{x,1}u[x, y, z, t] + ∂{y,1}v[x, y, z, t] +
          ∂{z,1}w[x, y, z, t])),
   (1 / c^2) ∂{t,2}w[x, y, z, t] ==
    A1 (∂{x,2}w[x, y, z, t] + ∂{y,2}w[x, y, z, t] +
        ∂{z,2}w[x, y, z, t]) +
     A2
      (∂{z,1} (∂{x,1}u[x, y, z, t] + ∂{y,1}v[x, y, z, t] +
          ∂{z,1}w[x, y, z, t]))
  },
  {x == -0.5, u[x, y, z, t] == 0, v[x, y, z, t] == 0,
   w[x, y, z, t] == 0},
  {x == 0.5, u[x, y, z, t] == 0, v[x, y, z, t] == 0,
   w[x, y, z, t] == 0},
  {y == -0.5, u[x, y, z, t] == 0, v[x, y, z, t] == 0,
   w[x, y, z, t] == 0},
  {y == 0.5, u[x, y, z, t] == 0, v[x, y, z, t] == 0,
   w[x, y, z, t] == 0},
  {z == -2, u[x, y, z, t] == 0, v[x, y, z, t] == 0,
   w[x, y, z, t] == 0},
  {z == 2, u[x, y, z, t] == 0, v[x, y, z, t] == 0,
   w[x, y, z, t] == 0},
  {t == 0, u[x, y, z, t] ==
    Exp[- ((x - 0.04) ^2 + (y - 0.04) ^2) / 0.01],
   ∂{t,1}u[x, y, z, t] == 0.1,
   v[x, y, z, t] == Exp[- ((x + 0.04) ^2 + (y + 0.04) ^2) / 0.01],
   ∂{t,1}v[x, y, z, t] == 0.2,
   w[x, y, z, t] == Exp[- (z^2) / 0.01],
   ∂{t,1}w[x, y, z, t] == 0.3}
 },
 {
  {{x, -0.5, 0.5}}, {{y, -0.5, 0.5}}, {{z, -2, 2}}
 }
};
```

Here we have taken simple Dirichlet boundary conditions; the practically interesting cases could have Robin boundary conditions involving normal derivatives of the fields, in which case we would need to do a little more work to transform the normal derivatives into suitable combinations of derivatives with respect to Cartesian coordinates.

For this problem, we use a slightly different version of the setup function that does not make *MathCode*-related declarations; it returns a list that has the prototype information about the functions for which C++ code is desired.

```
AbsTime[mcode = SetupMathPDE2[wave3d, {c, A1, A2}];]
```

```
{20.5996208 Second, Null}
```

We now apply the function `MathCodePart` to the list `mcode` to make declarations, generate C++ code, compile it, and install the executable.

```
AbsTime[MathCodePart @@ mcode]
```

```
{187.3093376 Second, Null}
```

Let us run `SolvePDE` by taking `Nx = Ny = Nz = 8`, and `△t = 0.1`. The parameter values are `{c, A1, A2} = {1., .1, 0.5}`.

```
{tot1, tot2, tot3} = {8, 8, 8};
AbsTime[SolvePDE[tot1, tot2, tot3, 0.1, 1., .1, 0.5, 1, 10];]
```

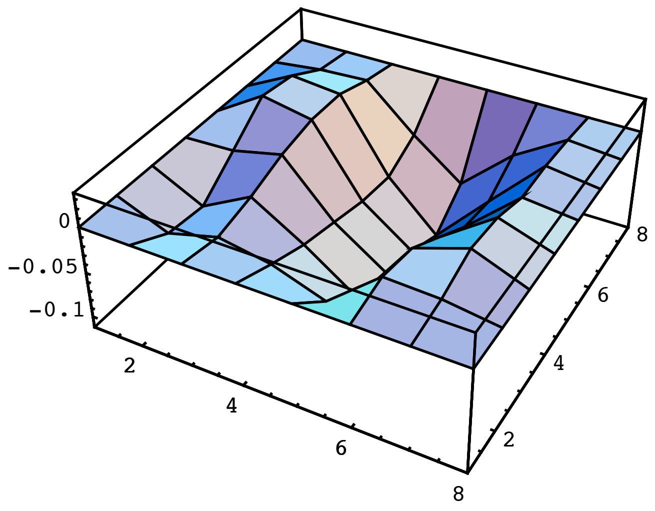
```
{275.7665328 Second, Null}
```

Let us define a function to extract function values (for `u`) over a two-dimensional cross section.

```
plotlistu[z_] :=
  Table[solutionAt[i, j, z, 1, tot1, tot2, tot3],
   {i, 1, tot1}, {j, 1, tot2}];
```

Here is the plot of the dependent variable `u` over a two-dimensional cross section for `z = (4 - 1) * (1.0 / 8.0) = 0.375`.

```
ListPlot3D[plotlistu[4]]
```

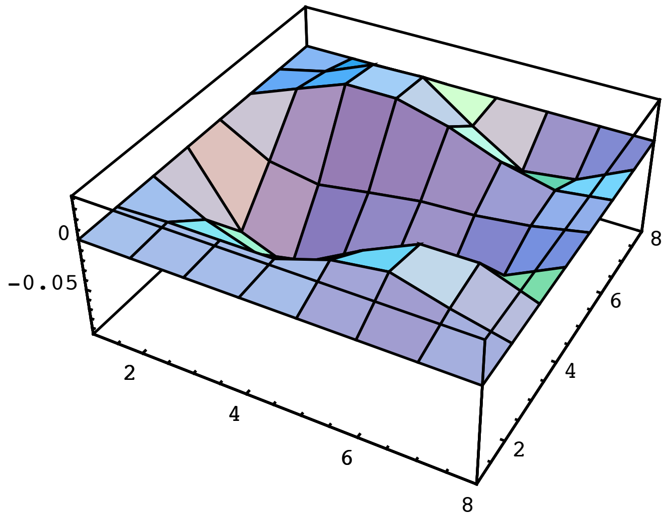


We can plot the other dependent variables v and w in the same manner. Let us define a function to extract function values for v over a two-dimensional cross section.

```
plotlistv[z_] :=
  Table[solutionAt[i, j, z, 2, tot1, tot2, tot3],
   {i, 1, tot1}, {j, 1, tot2}];
```

Here is the plot of the dependent variable v over a two-dimensional cross section for z = (4 - 1) * (1.0 / 8.0) = 0.375.

```
ListPlot3D[plotlistv[4]]
```

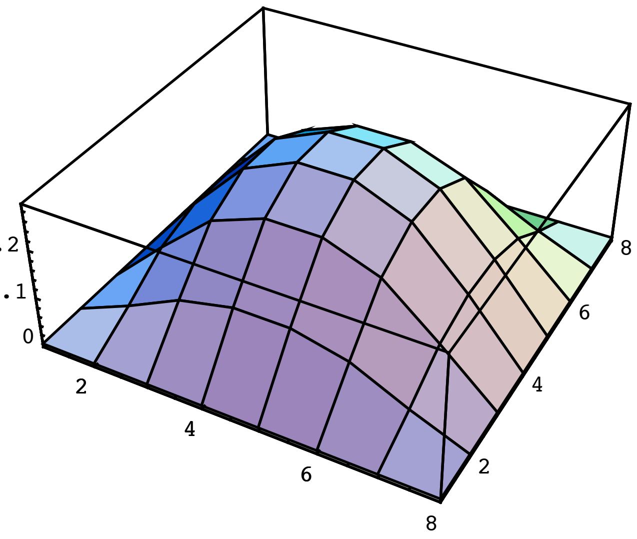


Finally, let us define a function to extract function values for w over a two-dimensional cross section.

```
plotlistw[z_] :=
  Table[solutionAt[i, j, z, 3, tot1, tot2, tot3],
   {i, 1, tot1}, {j, 1, tot2}];
```

Here is the plot of the dependent variable `w` over a two-dimensional cross section for `z = (4 - 1) * (1.0 / 8.0) = 0.375`.

```
ListPlot3D[plotlistw[4]]
```



We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

□ 4.4 Nonlinear Problems

■ 4.4.1 Advection-Diffusion and Inviscid Burger's Equations

Let us now consider the following PDE

$$u(x,t)\frac{\partial u(x,t)}{\partial t} + \frac{\partial u(x,t)}{\partial t} = c\,\frac{\partial^2 u(x,t)}{\partial x^2},$$

where c is a constant. When $c = 0$, the equation becomes the inviscid Burger's equation, and when $c = 1$, the equation becomes the advection-diffusion equation. Let us define this PDE problem with Dirichlet boundary conditions and a simple space-dependent initial condition.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];
```

```
nonlinear =
  {
   {
    {∂{t,1} u[x, t] + u[x, t] ∂{x,1} u[x, t] == c * ∂{x,2} u[x, t]},
    {x == 0, u[x, t] == u1},
    {x == 1, u[x, t] == u1},
    {t == 0, u[x, t] == u1 + x (1 - x)}
   },
   {
    {{x, 0, 1}}
   }
  };
```

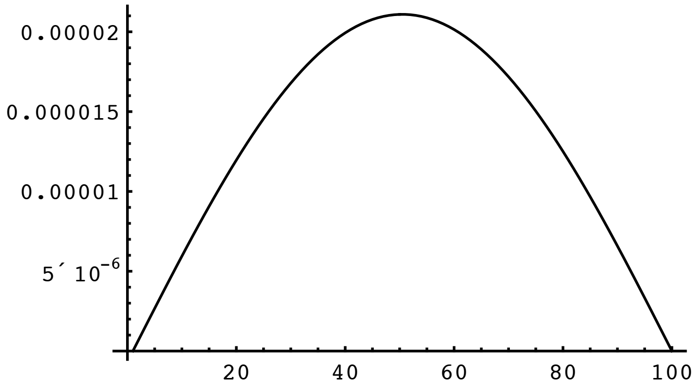
```
SetupMathPDE[nonlinear, {u1, c}]
```

In this case, the PDE leads to a nonlinear algebraic system that must be solved iteratively. Let us choose a small time step of 0.01 and evolve the solution by 100 time steps (i.e., up to `t = 1`) for a spatial size of 100 grid points. We first choose `c = 1`, so we have the advection-diffusion limit.

```
AbsTime[SolvePDE[100, 0.01, 0.0, 1.0, 1, 100, 0.001, 10];]
```

```
{2.4234848 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```

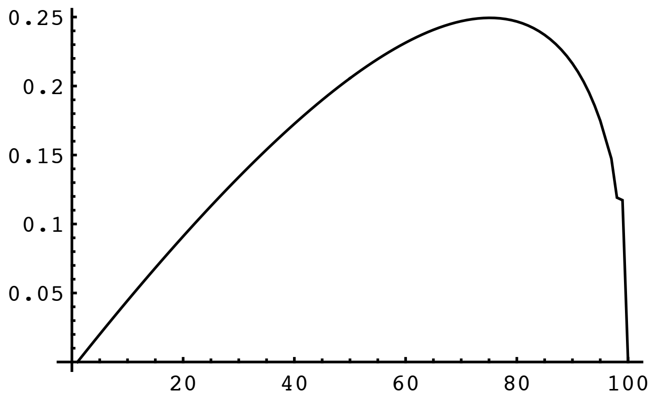


We now solve the problem with `c` = 0: in this limit, we have the inviscid Burger's equation.

```
AbsTime[SolvePDE[100, 0.01, .0, .0, 1, 100, 0.001, 10];]
```

```
{4.5865952 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```

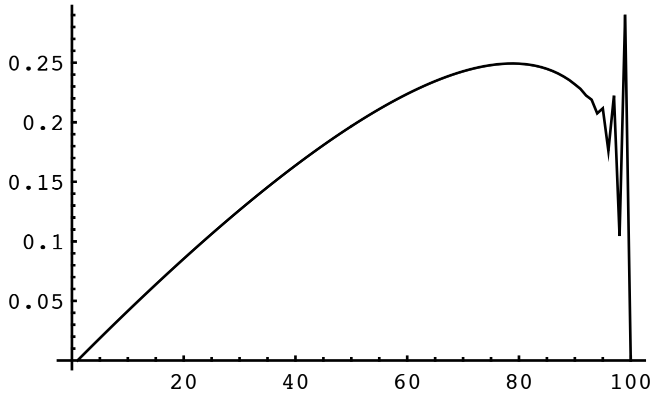


The solution varies very strongly in space near the right boundary. This indicates that finite-difference approximation is not very good for such problems. Indeed, if we evolve the solution by a further 15 time steps, we can see the quality of the solution deteriorate further.

```
AbsTime[SolvePDE[100, 0.01, .0, .0, 0, 15, 0.001, 10];]
```

```
{0.7110224 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```



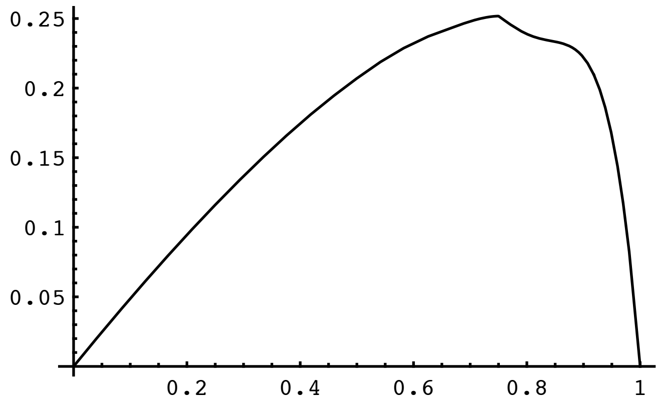
Now there are wild fluctuations near the right boundary, and the solution is not reliable. Let us solve the problem using `NDSolve`.

```
soln = NDSolve[{∂{t,1} u[x, t] + u[x, t] ∂{x,1} u[x, t] == 0,
    u[0, t] == 0, u[1, t] == 0, u[x, 0] == x (1 - x)}, u,
   {x, 0, 1}, {t, 0, 10}]
```

```
{{u → InterpolatingFunction[{{0., 1.}, {0., 10.}}, <>]}}
```

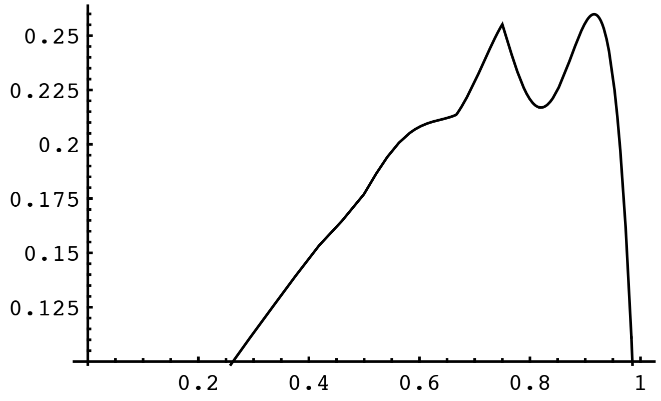
At `t = 1`, let us plot the solution.

```
Plot[(u[x, 1] /.soln)⟦1⟧, {x, 0, 1}]
```



This agrees well with the solution obtained using *MathPDE*, and the rapid fall near the right end of the system must be noticed. Let us now plot the solution at a later time, `t = 1.15`.

```
Plot[(u[x, 1.5`] /.soln)⟦1⟧, {x, 0, 1}]
```



Now we can see that there are regions where the solution shows strong fluctuations, although there are differences between the solutions obtained by *MathPDE* and `NDSolve`.

Examples such as these illustrate the limitations of *MathPDE*, based as it is on finite-difference discretization.

We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];

CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

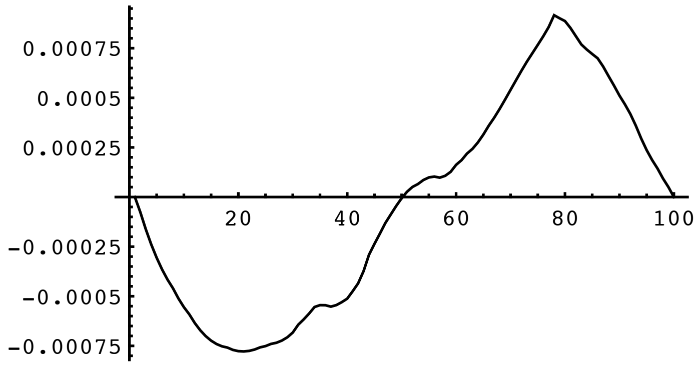
4.4.2 A Nonlinear Variant of the Diffusion Equation

Now let us try a nonlinear variant of the diffusion equation.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];

nonlinear1 =
  {
   {
    {u^(0,1)[x, t] == (u^(2,0)[x, t])^2 + c * u^(1,0)[x, t]},
    {x == 0, u[x, t] == 0},
    {x == 1, u[x, t] == 0},
    {t == 0, u[x, t] == x (1 - x)}
   },
   {
    {{x, 0, 1}}
   }
  };

SetupMathPDE[nonlinear1, {c}]

AbsTime[SolvePDE[100, 0.01, 1.0, 1, 100, 0.001, 10];]

{1.5221888 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```

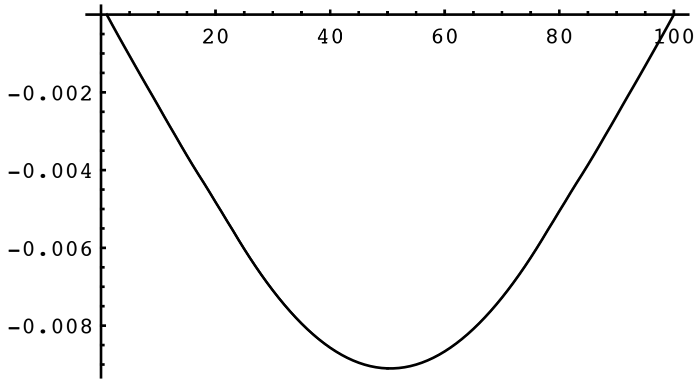


When we take `c = 0`, the solution is very different.

```
AbsTime[SolvePDE[100, 0.01, 0.0, 1, 100, 0.001, 10];]
```

```
AbsTime[Null]
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```



We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

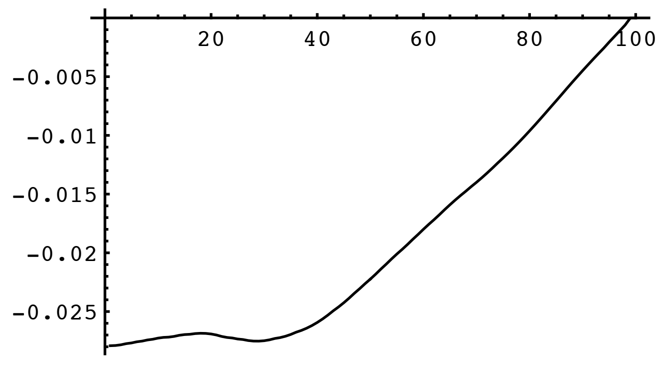
Consider the same PDE, but with a derivative boundary condition at x = 0.

We must first load the package *MathPDE*.

```
Needs["MathPDE`"];

nonlinear2 =
  {
   {
    {u^(0,1)[x, t] == (u^(2,0)[x, t])^2 + c * u^(1,0)[x, t]},
    {x == 0, u^(1,0)[x, t] == 0.},
    {x == 1, u[x, t] == 0},
    {t == 0, u[x, t] == x (1 - x)}
   },
   {
    {{x, 0, 1}}
   }
  };

SetupMathPDE[nonlinear2, {c}]
```

We solve it for c = 1.

```
AbsTime[SolvePDE[100, 0.01, 1.0, 1, 100, 0.001, 10];]

{8.4521536 Second, Null}

ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
 Joined → True]
```

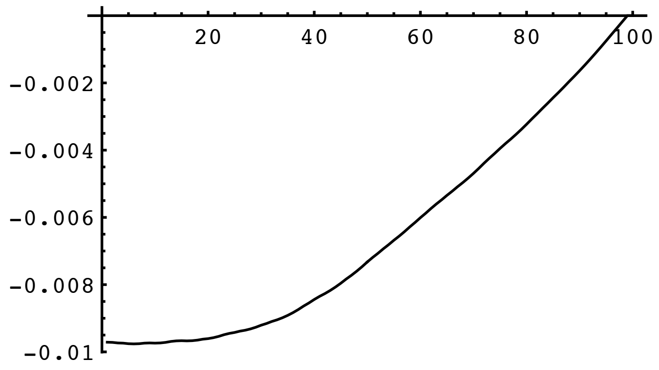


Now the solution profile is different, because the derivative at the left boundary must be zero, according to the boundary condition. Finally, we solve it for `c = 0`.

```
AbsTime[SolvePDE[100, 0.01, 0.0, 1, 100, 0.001, 10];]
```

```
{2.7439456 Second, Null}
```

```
ListPlot[Table[solutionAt[i, 1, 100], {i, 1, 100}],
  Joined → True]
```



We can uninstall the C++ program, and delete all the *MathCode*-related files if we do not need them.

```
UninstallCode[];
```

```
CleanMathCodeFiles[Confirm -> False, CleanAllBut -> {}];
```

We must quit the kernel before testing a new problem.

```
Quit
```

■ 5. Conclusion

In this article, we have presented a PDE solving system, *MathPDE*, that is based on *Mathematica*. We have demonstrated it for a variety of time-dependent PDE problems, and made comparisons with `NDSolve` in a few cases.

We have discussed at length the basic ideas underlying the design of *MathPDE*. We believe that there is scope for improvement in many areas, like the adaptive step sizes for space and time, the way in which higher-degree approximations are implemented, and so on. The limitations of *MathPDE* for nonlinear problems are obvious, based as it is on finite differences.

The main attractions of *MathPDE* are its ability to handle a wide range of spatial domains and the generation of a standalone C++ program for performing numerical computation.

■ References

[1] H. P. Langtangen, “Computational Partial Differential Equations: Numerical Methods and Diffpack Programming,” *Lecture Notes in Computational Science and Engineering*, Vol. 2, Berlin: Springer-Verlag, 1999.

[2] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, New York: Springer, 1985. www.cs.purdue.edu/ellpack/ellpack.html.

[3] E. Mossberg, K. Otto, and M. Thuné, “Object-Oriented Software Tools for the Construction of Preconditioners,” *Scientific Programming*, **6**(3), 1997 pp. 285–295. www.informatik.uni-trier.de/~ley/db/journals/sp/sp6.html.

[4] K. Åhlander, “An Object-Oriented Framework for PDE Solvers,” Ph.D. thesis, Department of Scientific Computing, Uppsala University, Sweden, 1999. uu.diva-portal.org/smash/record.jsf?pid=diva2:162116.

[5] W. D. Henshaw. “Overture Documentation, LLNL Overlapping Grid Project.” (May 23, 2011) www.llnl.gov/CASC/Overture/henshaw/overtureDocumentation/overtureDocumentation.html.

[6] W. F. Ames, *Numerical Methods for Partial Differential Equations*, 3rd ed., San Diego: Academic Press, 1992.

[7] C. F. Gerald and P. O. Wheatley, *Applied Numerical Analysis*, 5th ed., Reading, MA: Addison-Wesley, 1994.

[8] B. Gustafsson, H.-O. Kreiss, and J. Oliger, *Time Dependent Problems and Difference Methods*, New York: Wiley, 1995.

[9] M. Oh, “Modeling and Simulation of Combined Lumped and Distributed Processes,” Ph.D. thesis, University of London, 1995.

[10] K. Sheshadri and P. Fritzson, “A *Mathematica*-Based PDE Solver Generator,” *Proceedings of SIMS’99, Conference of the Scandinavian Simulation Society*, Linköping, Sweden, 1999 pp. 66–78.

[11] K. Sheshadri and P. Fritzson, “A General Symbolic PDE Solver Generator: Explicit Schemes,” *Scientific Programming*, **11**(1), 2003 pp. 39–55. dl.acm.org/citation.cfm?id=1240066&CFID=40011545&CFTOKEN=28188005.

[12] K. Sheshadri and P. Fritzson, “A General Symbolic PDE Solver Generator: Beyond Explicit Schemes,” *Scientific Programming*, **11**(3), 2003 pp. 225–235. dl.acm.org/citation.cfm?id=1240103.

[13] J. W. Demmel, J. R. Gilbert, and X. S. Li, *SuperLU Users’ Guide*. crd.lbl.gov/~xiaoye/SuperLU.

[14] P. Fritzson, *MathCode C++*, Linköping, Sweden: MathCore Engineering AB, 1998. www.mathcore.com.

[15] B. Fornberg, “Calculation of Weights in Finite Difference Formulas,” *SIAM Review*, **40**(3), 1998 pp. 685–691. amath.colorado.edu/faculty/fornberg/Docs/sirev_cl.pdf.

K. Sheshadri and P. Fritzson, “*MathPDE*: A Package to Solve PDEs by Finite Differences,” *The Mathematica Journal*, 2011. dx.doi.org/doi:10.3888/tmj.13-20.

About the Authors

K. Sheshadri works on computational algorithms for network biology problems. In the past he has worked on statistical mechanics of quantum condensed-matter systems. He was a guest researcher with Linköping University, Sweden in 1999, and had a collaboration with the university from 2000 to 2005, when the present work was done. Currently he is a lead scientist at Connexios Life Sciences, Bangalore, India.

Peter Fritzson is a professor and research director of the Programming Environment Laboratory at Linköping University. He is also director of the Open Source Modelica Consortium, director of the MODPROD center for model-based product development, and vice chairman of the Modelica Association, organizations he helped to establish. During 1999–2007 he served as chairman of the Scandinavian Simulation Society and secretary of the European simulation organization, EuroSim. Fritzson's current research interests are in software technology, especially programming languages, tools, and environments; parallel and multicore computing; compilers and compiler generators; and high-level specification and modeling languages, with special emphasis on tools for object-oriented modeling and simulation, where he is one of the main contributors and founders of the Modelica language. Fritzson has authored or coauthored more than 210 technical publications, including 16 books or proceedings.

K. Sheshadri
Peter Fritzson
Programming Environment Laboratory
Department of Computer and Information Science,
Linköping University, S-581 83 Linköping, Sweden
kshesh@gmail.com
peter.fritzson@liu.se