

Chapter One

COMSOL MULTIPHYSICS AND THE BASICS OF NUMERICAL ANALYSIS

W.B.J. ZIMMERMAN

*Department of Chemical and Process Engineering,
University of Sheffield,
Mappin Street, Sheffield S1 3JD, United Kingdom
E-mail: w.zimmerman@shef.ac.uk*

In this chapter, several key elements of numerical analysis are profiled in COMSOL Multiphysics with 0-D and 1-D models. These elements are root finding, numerical integration by marching, numerical integration of ordinary differential equations, and linear system analysis. These methods underly nearly all problem solving techniques by numerical analysis for chemical engineering applications. The use of these methods in COMSOL Multiphysics is illustrated with reference to some common applications in chemical engineering: flash distillation, tubular reactor design, diffusive-reactive systems, and heat conduction in solids.

1. Introduction

This chapter is rather busy, as it must accomplish several different goals. Primarily, it is intended to introduce key features of how COMSOL Multiphysics works. Secondly, it is to illustrate how these key features can be used to analyze simple enough chemical engineering problems that 0-D and 1-D spatial or spatial-temporal systems can describe them. The chapter is also intended to whet your interest to investigate modeling and simulation with COMSOL Multiphysics by presenting at least a glimpse of the power of the COMSOL Multiphysics and MATLAB tools when applied to chemical engineering analysis.

Because COMSOL Multiphysics is not intended to be a general tool for problem solving, some of these goals are achieved in a roundabout fashion. The author has previously taught courses in chemical engineering problem solving by numerical analysis using FORTRAN, *Mathematica*TM, and MATLABTM, and used all the examples implemented here with those tools. Furthermore, the most extensive compilations of chemical engineering

problem solving by numerical analysis have been done in POLYMATH [1], which only seems to be used by the chemical engineering community through the CACHE program.

In this book's predecessor, at this point we introduced the concept of a zero-dimensional domain to solve nonlinear algebraic equations and time-dependent ordinary differential equations. Conceptually, the concept of a zero-dimensional domain is simply a single finite element. Understanding the finite element method from the perspective of what happens in a single finite element is pedagogically very useful. COMSOL Multiphysics, however, has made it much simpler to solve 0-D algebraic and time-dependent ODEs by creating a separate dialogue box to specify them. So in this chapter, we will solve some examples both ways.

2. Method 1: Root finding

Typically, courses in numerical analysis go into great detail in the description of the algorithm classes used for root finding. From experience, there are only two algorithms that are really useful — the bisection method and Newton's method. Instead of presenting all the methods, here we will consider why root finding is one of the most useful numerical analysis tools. Finding roots in linear systems is fairly easy. Nonlinear systems are the challenge, and nearly all interesting dynamics stem from nonlinear systems. The interest in root finding in nonlinear systems results from its utility in describing inverse functions. Why? Because with most nonlinear functions, the "forward direction," $y = f(u)$, is well described, but the inverse function of $u = f^{-1}(y)$ may be analytically indescribable, multi-valued (nonunique), or even nonexistent. But if it exists, then the numerical description of an inverse function is identical to a root finding problem — find u such that $F(u) = 0$ is equivalent to $F(u) = f(u) - y = 0$. Since the goal of most analysis is to find a solution of a set of constraints on a system, this is equivalent to inverting the set of constraints. COMSOL Multiphysics has a core function for solving nonlinear systems, `femnlm`, and in this section its use to solve 0-D root finding problems will be illustrated.

`femnlm` uses Newton's method which with only one variable u uses the first derivative $F'(u)$ which is used iteratively to drive toward the root. The method takes a local estimate of the slope of the function and projects to the root. The slope can be computed either analytically (Newton-Raphson Method) or numerically (the secant method). If the slope can be computed either way, you can use Taylor's theorem to project to the root. The basic

idea is to use a Taylor expansion about the current guess u_0 :

$$f(u) = f(u_0) + (u - u_0)f'(u_0) + \dots, \quad (1)$$

which can be re-arranged, ignoring higher order terms in $(u - u_0)$ to estimate the root as

$$u = u_0 - \frac{f(u_0)}{f'(u_0)}. \quad (2)$$

This methodology is readily extendable to a multiple dimension solution space, i.e. u is a vector of unknowns, and division by $f'(u_0)$ represents multiplication by the inverse of the Jacobian of f . The next subsection illustrates root finding in COMSOL Multiphysics.

2.1. Root finding: A simple application of the COMSOL Multiphysics nonlinear solver

As implied in the previous section, root finding is a “0-D” activity, at least in terms of the spatial-temporal dependence of the solution vector of unknowns, u , which can be a multi-dimensional vector. COMSOL Multiphysics does not have a “0-D” application mode, so we improvise in 1-D. This has the undesirable feature that we will unnecessarily solve the problem redundantly at several points in space. Given the small size of the problem, the efficiency of COMSOL Multiphysics coding, and the speed of modern microprocessors, this causes no guilt whatsoever!

Start up MATLAB and type COMSOL Multiphysics in the command window. After several splash screens, you should be facing the Model Navigator window. Follow the steps in Table 1 to set up a 0-D application mode to solve the nonlinear polynomial equation:

$$u^3 + u^2 - 4u + 2 = 0. \quad (3)$$

Physics: Subdomain settings specifies the equation to be satisfied in each subdomain in Table 1. Notice the equation in the upper left given in vector notation. In 1-D, this equation can be simplified to

$$d_a \frac{\partial u}{\partial t} - \frac{\partial}{\partial x} \left(c \frac{\partial u}{\partial x} + \alpha u \gamma \right) + a u + \beta \frac{\partial u}{\partial x} = f. \quad (4)$$

Clearly, $\alpha \gamma$ and β are redundant with the simplification to 1-D. Since we want to find roots in 0-D, however, all the coefficients on the LHS of (4) can be set to zero. By rearranging the polynomial, we can readily see that $a = 4$ and $f = u^3 + u^2 + 2$. Note that we discretize the domain with a single element by specifying the maximum element size to be one, giving us 0-D!

Table 1. Root finding example in coefficient mode. File name: rootfinder.mph.

Model Navigator	Select 1-D. COMSOL Multiphysics:PDE modes:PDE, coefficient form
Draw Menu	Specify objects: Line. Coordinates pop-up menu. $x : 0 \ 1$ name: interval OK
Physics Menu: Boundary settings	Select domains: 1 and 2 (hold down Ctrl key) Select Neumann boundary conditions Leave defaults $q = 0 \ g = 0$ OK
Physics Menu: Sub-domain settings	Select domain: 1 Set $c = 0; a = 4; f = u^3 + u^2 + 2; d_a = 0$ Apply. Select init tab: set $u(t_0) = -2$ OK
Mesh menu: mesh parameters	Set maximum element size 1 Hit remesh. OK
Solve menu: solver parameters	Stationary nonlinear. Solve. OK
Post-processing: Point Evaluation	Boundary selection: 1. Expression: u . OK

By specifying the initial guess of as $u(t_0) = -2$, we find the root nearest to this value. If you are wondering why $a = 4$ was set, rather than all of the dependence put into f , it is so that the finite element discretization of the RHS of (4) does not result in a singular stiffness matrix.

The post-processing stage shows the result in the output window:

Value : -2.732051 , Expression : u , Boundary : 1.

The analytically determined root nearest to this is $-1 - \sqrt{3}$, showing the numerical solution in good agreement. According to the structure of the quadratic formula of algebra, clearly another root is $-1 + \sqrt{3}$, and by inspection, the third root is 1. Returning to the subdomain settings, set the initial guess to $u(t_0) = -0.5$ and COMSOL Multiphysics converges to $u = 0.732051$, again a good approximation. $u(t_0) = 1.2$ as an initial guess converges to $u = 1$.

This exercise clarifies two features of nonlinear solvers and problems — (i) nonlinear problems can have multiple solutions; (ii) the initial guess is key to convergence to a particular solution. With Newton’s method, it is usually the case that convergence is to the nearest solution, but overshoots in highly nonlinear problems may override this stereotype. These features persist in higher dimensional solution spaces and with spatial-temporal dependence.

The COMSOL Multiphysics model mph-file rootfinder.mph contains all the MATLAB source code with FEMLAB extensions to reproduce the current state of the FEMLAB GUI. This file is available from the website

Table 2. Root finding example in general mode. File name: rtfindgen.mph.

Model Navigator	1-D, COMSOL Multiphysics:PDE modes, general form
Options	Set Axes/Grid to [0,1]
Draw	Name: Interval; Start = 0; Stop = 1
Physics Menu/ Boundary Settings	Set both endpoints (domains) to Neumann BCs
Physics Menu/Subdomain Settings	set $\Gamma = 0$; $d_a = 0$; $F = u^3 + u^2 - 4 * u + 2$
Mesh mode	Set Max element size, general = 1; Remesh
Solve	Use default settings (nonlinear solver, exact Jacobian)
Post-process	After five iterations, the solution is found. Click on the graph to read out $u = 0.732051$. Play with the initial conditions to find the other two roots

<http://eyrie.shef.ac.uk/femlab>. Just pull down the file menu, select Open model m-file, and use the Open file dialog window to locate it. You can rapidly place your nonlinear function in the Subdomain settings, specify an initial guess, and use the stationary nonlinear solver to converge to a solution. But what if your function does not have a linear component to put on the LHS of (4)? For instance, $\tanh(u) - u^2 + 5 = 0$ results in a singular stiffness matrix when FEMLAB assembles the LHS of (4). The suggestion is to set the coefficient of the second derivative of u , $c = 1$ in the Subdomain settings. Coupled with the Neumann boundary conditions, this artificial diffusion cannot change the fact that the solution must be constant over the single element, yet it prevents the stiffness matrix from becoming singular.

Root finding in General Mode

The difficulty with a singular stiffness matrix assembly for $\tanh(u) - u^2 + 5 = 0$ can be averted by using General Mode, which solves

$$e_a \frac{\partial^2 u}{\partial t^2} + d_a \frac{\partial u}{\partial t} + \frac{\partial \Gamma}{\partial x} = F, \quad (5)$$

Table 3. Root finding in ODE settings.

Physics Menu: ODE settings	Name: v . Equation: $\tanh(v) - v^2 + 5$ OK
Solve menu: solver parameters	Stationary nonlinear. Solve. OK
Post-processing	Point evaluation. Boundary 2. Expression: v
Report window	Value: -2.008819 , Expression: v , Boundary: 2

where $\Gamma(u, ux)$ is in principle the same functionality as the coefficient form (4), but is treated differently by the Solver routines. In Coefficient Mode, the coefficients are treated as independent of u unless the numerical Jacobian is used, which brings out some of the nonlinear dependency — iteration does the rest. The exact Jacobian in General Mode differentiates both Γ and F with respect to u symbolically in assembling the stiffness matrix. Typically, General Mode requires fewer iterations for convergence than Coefficient Mode with the numerical Jacobian. The use of the exact Jacobian below does not require any special treatment to avoid a singular stiffness matrix in the treatment of the linear terms as the coefficient mode did. In general, General Mode is more robust at solving nonlinear problems than Coefficient Mode. It is my opinion that Coefficient Mode is a “legacy” feature of COMSOL Multiphysics — the PDE Toolbox of MATLAB, in many ways a precursor to FEMLAB and COMSOL Multiphysics, uses coefficient representations extensively. Further, the coefficient formulation with numerical Jacobian is a long standing FEM methodology, so for benchmarking against other codes, it is a useful formulation.

Table 2 holds the recipe for General Mode — a minor modification of what we just did.

Although setting up this template (`rtfindgen.mph`) for root finding of simple functions of one variable was rather involved, and in fact MATLAB has a simpler procedure for root finding using the built-in function `fzero` and inline declarations of functions, the COMSOL Multiphysics GUI now provides the utility to solve algebraic constraints as auxiliary equations in auxiliary variables, termed state variables. As an adjunct to our general mode root finder model, follow the steps in Table 3 to solve a nonlinear equation for a state variable v .

The next subsection applies our newly constructed nonlinear root finding scheme to a common chemical engineering application, flash distillation, which clarifies a few more features of the COMSOL Multiphysics GUI.

2.2. Root finding: Application to flash distillation

Chemical thermodynamics harbors many common applications of root finding, since the constraints of chemical equilibrium and mass conservation are frequently sufficient, along with constitutive models like equations of state, to provide the same number of constraints as unknowns in the problem. In this subsection, we will take flash distillation as an example of simple root

Table 4. Initial composition to the flash unit and partition coefficients K at equilibrium.

Component	X_f	K_i at 65°C and 3.4 bar
Ethane	0.0079	16.2
Propane	0.1281	5.2
<i>i</i> -Butane	0.0849	2.6
<i>n</i> -Butane	0.2690	1.98
<i>i</i> -Pentane	0.0589	0.91
<i>n</i> -Pentane	0.1361	0.72
Hexane	0.3151	0.28

finding for one degree of freedom of the system, which is conveniently taken as the phase fraction ϕ .

A liquid hydrocarbon mixture undergoes a flash to 3.4 bar and 65°C. The composition of the liquid feed stream and the “ K ” value of each component for the flash condition are given in the table. We want to determine composition of the vapor and liquid product streams in a flash distillation process and the fraction of feed leaving the flash as liquid. Table 4 gives the initial composition of the batch.

A material balance for component i gives the relation

$$X_i = (1 - \phi)y_i + \phi x_i, \quad (6)$$

where X_i is the mole fraction in the feed (liquid), x_i is the mole fraction in the liquid product stream, y_i is the mole fraction in the vapour product, and ϕ is the ratio of liquid product to feed molar flow rate. The definition of the equilibrium coefficient is $K_i = y_i/x_i$. Using this to eliminate x_i from the balance relation results in a single equation between y_i and X_i :

$$y_i = \frac{X_i}{1 - \phi(1 - \frac{1}{K_i})}. \quad (7)$$

Since the y_i must sum to 1, we have a nonlinear equation for ϕ :

$$f(\phi) = 1 - \sum_{i=1}^n \frac{X_i}{1 - \phi(1 - \frac{1}{K_i})} = 0, \quad (8)$$

where n is the number of components. This function $f(\phi)$ can be solved for the root(s) ϕ , which allows back-substitution to find all the mole fractions in the product stream. The Newton-Raphson method requires the derivative $f'(\phi_k)$ at the current estimate to determine the improved estimate, and COMSOL Multiphysics will compute this analytically as an option. It is

Table 5. Flash distillation example.

Model Navigator	Select 1-D. COMSOL Multiphysics:PDE modes:PDE, general form
Draw Menu	Specify objects: Line. Coordinates pop-up menu. x : 0 1 name: interval OK
Options: Constants	Enter the data from Table 4 X_1 , 0.0079, etc.
Options: Scalar Expressions	Define expression for the terms in the RHS of (8) $t_1 - X_1/(1 - u * (1 - 1/K_1))$ $t_2 - X_2/(1 - u * (1 - 1/K_2))$ etc.
Physics Menu: Boundary settings	Select domains: 1 and 2 (hold down Ctrl key) Select Neumann boundary conditions Leave defaults $q = 0$ $g = 0$ OK
Physics Menu: Sub-domain settings	Select domain: 1 Set $F = 1 + t_1 + t_2 + \dots + t_7$; $d_a = 0$ Select Init tab; set $u(t_0) = 0.5$
Mesh menu: mesh parameters	Set maximum element size 1 Hit remesh. OK
Solve menu: solver parameters	Stationary nonlinear. Solve. OK
Post-processing: Point Evaluation	Boundary selection: 1. Expression: u . OK Report window: Value : 0.458509, Expression: u

fairly straightforward to arrive at the Newton-Raphson iterate as

$$f'(\phi) = \sum_{i=1}^n \frac{X_i(1 - \frac{1}{K_i})}{[1 - \phi(1 - \frac{1}{K_i})]^2}. \quad (9)$$

Now onto the COMSOL Multiphysics solution for root finding. As an exercise, we will set up the solution using the general PDE mode. We could just load `rootfinder.mph` or `rtfindgen.mph` and customize it, but of course becoming familiar with COMSOL Multiphysics' features is an important goal.

Start up COMSOL Multiphysics and await the Model Navigator window. If you already have a COMSOL Multiphysics session started, save your workspace as a model MPH-file or the commands as a model m-file, and the pull down the file menu and select New. Follow the steps as arranged in Table 5 to set up the flash distillation example. Note that we have two additional stages in this example — Options: Constants and Options: Expressions. Constants can be defined and then used wherever a pure number might be legally used in a COMSOL Multiphysics data entry field. Expressions are similar to constants in that they can also be used wherever a COMSOL Multiphysics data entry field permits, but have the additional feature that they depend on the dependent variable(s). They

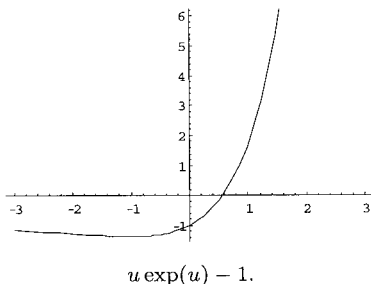
are also available for post-processing. Try putting t_1 and t_2 into the post-processing data display as expressions:

Value : -0.013865 , Expression : t_1 , Boundary : 1 ,
 Value : -0.203441 , Expression : t_2 , Boundary : 1 .

Another useful set of information comes from the solver log. Pull down the solver menu to the bottom and select View Log. The Solver Log dialogue window pops up. This shows when the solver ran, what solver command was executed (here femlin) and how circuitous the path to the solution from the initial condition was. Here it took three iterations to achieve absolute error of 10^{-9} . This information comes in particularly handy if your solution does not converge or is slowly convergent.

Exercises.

- 1.1. Find the roots of the equation $f(u) = u^3 - 3u^2 + \frac{5}{2}u - \frac{1}{2} = 0$. As this function is a cubic polynomial, there is an analytic solution in the irrational numbers, $u = 1$, $u = 1 - \frac{1}{\sqrt{2}}$, $u = 1 + \frac{1}{\sqrt{2}}$.



- 1.2. Find the root of the equation $f(u) = u e^u - 1 = 0$. This function is transcendental, which means that it has no analytic solution in the rational numbers. If you use Coefficient Mode, put $c = 1$ to aid convergence.

3. Method 2: Numerical integration by marching

Numerical integration is the mainstay of numerical analysis. The first duty of scientific computing before there were digital computers were to fill the handbooks with tables of special functions, nearly all of which were solutions to special classes of ordinary differential equations. And the computational methodology? One-dimensional numerical integration.

There are two classes of 1-D integration: initial value problems (IVP) and boundary value problems (BVP). The latter will be considered in the next section. The easiest to integrate are IVPs, as if all the initial conditions are all specified at a point, it is straightforward to step along by small increments according to the local first derivative. Clearly, if the ODE is

first order, i.e.

$$\frac{dy}{dt} = f(t), \quad y|_{t+\Delta t} = y|_t + \Delta t f(t). \quad (10)$$

The second statement in (10) is true exactly in the limit of $\Delta t \rightarrow 0$. It is termed the Euler method and is the most straight-forward way of integrating a first order ODE. In one dimension, you simply step forward according to the local value of the derivative of f at the point (x_n, y_n) , where n refers to the n -th discretization step of the interval upon which you are integrating. Thus,

$$\begin{aligned} y_{n+1} &= y_n + hf(x_n, y_n), \\ x_{n+1} &= x_n + h. \end{aligned} \quad (11)$$

This assumes that the derivative does not change over the step of size h , which is only actually true for a linear function. For any function with curvature, this is a lousy assumption. Consider, for instance, how far wrong we go with a large step size in Figure 1. So clearly, one important point in improving on Euler's Method is to be able to use big steps, since it requires small steps for good accuracy. Euler's method is called "first order" accurate, as the error only decreases as the first power of h .

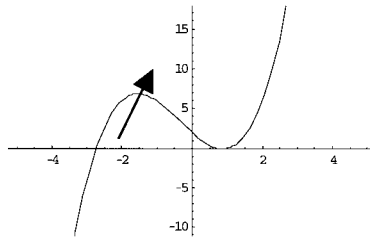


Figure 1. Euler step in numerical integration.

Runge-Kutta methods

So if we want to use big step sizes, we need a "higher order method," one that reduces the error faster as step size decreases. A k -th order method has error which diminishes as h^k . Given that it is curvature that we know we are neglecting, we can estimate the curvature of the graph $y(x)$ by evaluating the slope $f(x)$ at several intermediate points between x_n and x_{n+1} . Second order accuracy is obtained by using the initial derivative to estimate a point halfway across the interval, then using the midpoint derivative across the full width of the interval.

$$\begin{aligned}
 k_1 &= hf(x_n, y_n), \\
 k_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right), \\
 y_{n+1} &= y_n + k_2 + O(h^3).
 \end{aligned}
 \tag{12}$$

The upshot is that by making two function evaluations, we have saved a whole order in accuracy. So, for instance, with a first order method, N calculations gives us an error $O(1/N)$, but for a second order method, $2N$ calculations gives us error $O(1/4N^2)$. It would take N^2 calculations to do so well with a first order method.

Higher order Runge-Kutta methods

Can we do better? Clearly, we can use a three midpoint method to achieve third order accuracy, a four midpoint method for fourth order accuracy, etc. When should we stop? Well, there is more programming work for higher order methods, so our time is a consideration. But intrinsically, functions may not be very smooth in their k -th derivative that we are estimating. It is possible that in increasing the “accuracy of the approximation,” the round-off error of higher derivative terms so estimated becomes appreciable. If that is the case, with each successive step, the error may grow rapidly. This implies that higher order methods are less *stable* than lower order methods.

The common choice for integrating ODEs is to use a fourth order Runge-Kutta method. This is fairly compact to programme, gives good accuracy, and typically has good stability character.

Other methods

There are two other famous problems in numerical integration that need particular programming attention:

Numerical Instability. Suppose your integration diverges to be very far from known test-cases, even with a high order accuracy method. Then it is likely that your method is numerically unstable. You can cut down your step size and eventually achieve numerical stability. However, this means a longer calculation. If you are computing a great many such integrations and the slowness really bothers you, try a semi-implicit method like predictor-corrector schemes.

Stiff Systems. Stiff systems usually have two widely disparate length or time scales on which physical mechanisms occur. Stiff systems may have

“numerical instability” of the explosive sort mentioned above, or they may have nonphysical oscillations. Try the book of Gear [2] for a recipe to treat stiff systems.

3.1. Numerical integration: A simple example

Higher order ODEs are treatable by marching methods by reduction of order. Suppose you have an ODE:

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x). \quad (13)$$

Unless $q(x)$ and $r(x)$ are constants, then you are out of luck with most textbook analytic methods for finding a solution. There are special cases of $q(x)$ and $r(x)$ that lead to analytic solutions, but these days you are better off computing the numerical solution in nearly all cases anyway. Why? Because you need to plot the graph of the solution $y(x)$ to make sense of it, so you will need to harness some computing horse power for the graphics. How? First let’s reduce the order of the second order system above to two first order systems:

$$\begin{aligned} \frac{dy}{dx} &= z(x), \\ \frac{dz}{dx} &= r(x) - q(x)z(x). \end{aligned}$$

Each of these ODEs can be numerically integrated by time marching methods as in (11) or (12), *simultaneously*. A simple example is

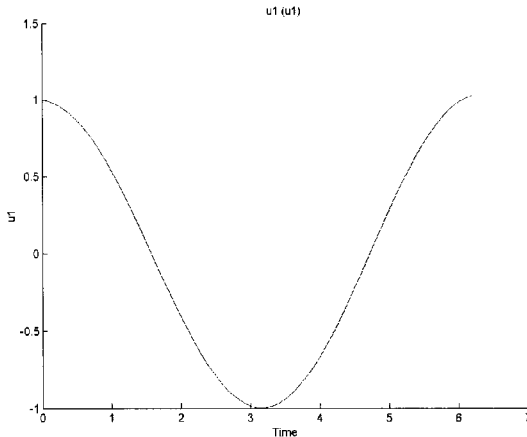


Figure 2. $u_1(t)$ over one period.

$$\frac{d^2u}{dt^2} + u = 0. \quad (14)$$

Reduction of order yields two first order ODEs:

$$\begin{aligned} \frac{du_1}{dt} &= -u_2, \\ \frac{du_2}{dt} &= u_1. \end{aligned} \quad (15)$$

Taking the initial condition to be $u_1 = 1$ and $u_2 = 0$, we can now set up a 0-D spatial system to integrate this coupled set of ODEs.

Start up COMSOL Multiphysics and await the Model Navigator window. If you already have a COMSOL Multiphysics session started, save your workspace as a model MPH-file or the commands as a model m-file, and then pull down the File menu and select New. Follow the set up information in Table 6.

This application mode gives us two dependent variables u_1 and u_2 and one space coordinate x . Notice the equation in the upper left in Subdomain Settings is given in vector notation. Because we have a vectorial set of variables, all the data entry tabs are for vectorial (F) or matrix (d_a) input.

`linspace(0,2*pi,50)` is the MATLAB command to create a vector of length 50 which uniformly goes from 0 to 2π . Data display gives $u_1(t = 2\pi) = 1.004414$. Given that the analytic solution is $u_1(t = 2\pi) = 1$, this is

Table 6. Time integration by marching of a simple example.

Model Navigator	Select 1-D. COMSOL Multiphysics:PDE modes:PDE, general form Insert dependant variables: $u_1 u_2$ Select Element: Lagrange-Linear
Draw Menu	Specify objects: Line. Coordinates pop-up menu. x : 0 1 name: interval OK
Physics Menu: Boundary settings	Select domains: 1 and 2 (hold down Ctrl key) Select Neumann boundary conditions Leave defaults $q = 0$ $g = 0$ OK
Physics Menu: Sub- domain settings	Select domain: 1 Set $F_1 = -u_2$, $F_2 = u_1$; Select Init tab; set $u_1(t_w) = 1$
Mesh menu: mesh parameters	Set maximum element size 1 Hit remesh. OK
Solve menu: solver parameters	Time-dependent solve. Enter on General tab Times: <code>linspace(0, 2*pi, 50)</code> Solve. OK
Post-processing: Cross-section plot parameters	Point tab. Accept the default of u_1 General tab. Accept the default of all times OK

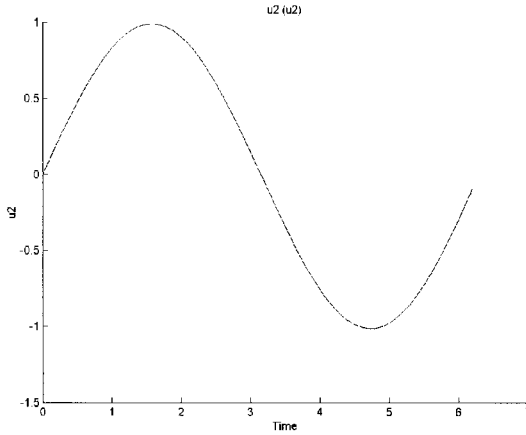


Figure 3. $u_2(t)$ over one period.

rather inaccurate (0.4%). Previously, FEMLAB permitted the user to select the time-integration scheme among several built-in solvers for MATLAB and FEMLAB. COMSOL Multiphysics does not permit this flexibility, but rather uses internal algorithm selection. It does, however, permit the user to adjust the local error tolerances (relative and absolute) on the General Tab of the Solver Parameters dialogue box. I changed the relative tolerance to 0.001 and absolute to 0.0001 to produce a somewhat better endpoint calculation of $u_1(t = 2\pi) = 0.998027$. Note that cumulative global error is of the order of the accuracy of the method 0.001.

These two figures (2 and 3) clarify that FEMLAB can reproduce the numerical integration of the cosine and sine functions with high fidelity if given a small enough time step. Although we think of sine and cosine as “analytic functions,” when tabulated this way, it is clear that the distinction between analytic functions and those that require numerical integration is specious — they are no more analytic than Bessel functions, elliptic functions, etc.

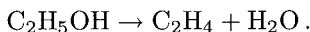
Exercise.

- 1.3. Try integrating equations (15) from the same initial conditions using the ODE Settings on the Physics Menu and naming v_1 and v_2 as state variables. The only particular difference for time dependent ODEs from algebraic equations is that you must use the notation for dv_1/dt , v_1t , etc.

4. Numerical integration: Tubular reactor design

In this section, a coupled set of first order nonlinear ODEs are solved simultaneously for the design of a tubular reactor undergoing a homogeneous chemical reaction. Typically, the key element in the design of a tubular reactor is the estimate of the length of the reactor.

A tubular reactor is used to dehydrate gaseous ethyl alcohol at 2 bar and 150°C. The formula for this chemical reaction is



Some experiments on this reaction have suggested the reaction rate expression at 2 bar pressure and 150°C, where C_A is the concentration of ethyl alcohol (mol/litre) and R is the rate of consumption of ethyl alcohol (mol/s/m³):

$$R = \frac{52.7C_A^2}{1 + \frac{0.013}{C_A}}.$$

The reactor is to have a 0.05 m diameter and the alcohol inlet flowrate is to be 10 g/s. The objective is to determine the reactor length to achieve various degrees of alcohol conversion. We wish to determine reactor length for the outlet alcohol mole fractions 0.5, 0.4, 0.3, 0.2, and 0.1.

Chemical engineering design theory

Assuming small heat of reaction, plug flow and ideal gas behaviour, it can be shown that the reacting flow is described by four ordinary differential equations in terms of the dependent variables C_A , C_W (the water concentration), V (the velocity) and x (the distance along the reactor from the inlet):

$$\begin{aligned} \frac{dC_A}{dt} &= -R \left(1 + \frac{C_A}{C} \right), \\ \frac{dC_W}{dt} &= R \left(1 - \frac{C_W}{C} \right), \\ \frac{dV}{dt} &= \frac{RV}{C}, \\ \frac{dx}{dt} &= V. \end{aligned} \tag{16}$$

The last equation states that the superficial velocity creates an equivalence between distance along the reactor and the residence time t that a fluid

element has to react. These equations are subject to the initial condition of the flow at the inlet ($t = 0$):

$$\begin{aligned} C_A(0) &= C & V(0) &= V_0, \\ C_W(0) &= 0 & x(0) &= 0. \end{aligned} \quad (17)$$

Approach

Clearly from the initial condition and stoichiometry, $C_W = C_E$ (the concentration of ethyl alcohol, and the value of C is constant as temperature and pressure are assumed constant. C can be found from the ideal gas law, with

$$C = \frac{p}{T(8314 \frac{\text{J}}{\text{kmolK}})}. \quad (18)$$

And the initial flow velocity V_0 can be determined from the flowrate given, the inlet density (the molecular weight of ethyl alcohol is 46 kg/kmol), and the tube cross-sectional area. The equations will need to be integrated numerically in space-time t until the required alcohol mole fractions have been reached. Use either simple Euler or Runge-Kutta numerical integration.

You may note that it is possible to solve for C_A without recourse to the other variables, but C_W , V , and x depend explicitly on t . But since the requirement is to find positions x where specific mole fractions occur, it is best to solve for all four variables simultaneously.

Partial results

A resolved numerical solution gives

$$\begin{aligned} \frac{C_A}{C} &= 0.1, \\ t &= 5.65225, \\ x &= 18.5435, \end{aligned} \quad (19)$$

with a profile for C_A/C as in Figure 4.

COMSOL Multiphysics implementation

We wish to create our pseudo-0-D simulation environment yet again, this time with four dependent variables. Start up COMSOL Multiphysics and await the Model Navigator window. Follow the steps in Table 7 to set up the tubular reactor design model. This application mode gives us four dependent variables $u_1 u_2 u_3 u_4$ and one space coordinate x . Since there are several parameters, it is useful to specify them with named constants. Furthermore, the rate law expression recurs, so it is convenient to define it as a scalar expression.

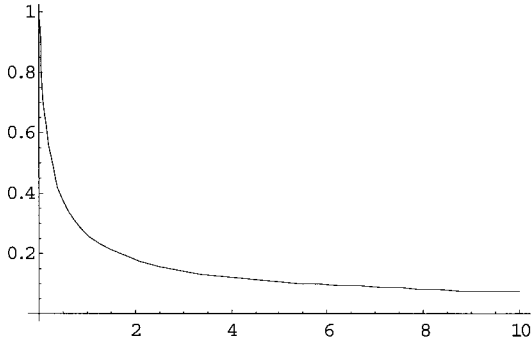


Figure 4. Profile of normalized alcohol concentration versus space time t .

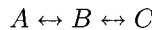
Try plotting point plots of u_1 , u_2 , u_3 and u_4 for the whole range of times. How good is the qualitative agreement with Figure 4? Does it agree numerically with the fully resolved solution?

Exercises.

- 1.4. Find the value of $y'(x = 1)$ from the system of equations below. Plot y' for x between 0 and 3.

$$\begin{aligned} y'' + y' + y^2 &= 0, \\ y(x = 0) &= 1, \\ y'(x = 0) &= 0. \end{aligned}$$

- 1.5. Linear systems of ODEs result from first order reversible reaction systems in a continuously stirred tank reactor. For instance, consider the isomerization reactions



with forward reaction rates k_1 and k_3 , respectively, as written; reverse reaction rates k_2 and k_4 , as written. First order kinetics leads to the following system of ODEs:

$$\begin{aligned} \frac{dc_A}{dt} &= -k_1 c_A + k_2 c_B, \\ \frac{dc_B}{dt} &= k_1 c_A - k_2 c_B - k_3 c_B + k_4 c_C, \\ \frac{dc_C}{dt} &= k_3 c_B - k_4 c_C. \end{aligned} \tag{20}$$

Table 7. Tubular reactor design modelling steps in COMSOL Multiphysics.

Model Navigator	Select 1-D. COMSOL Multiphysics:PDE modes:PDE, general form Set dependent variables: $u_1 u_2 u_3 u_4$ Select Element: Lagrange-Linear. OK
Draw Menu	Specify objects: Line. Coordinates pop-up menu. x : 0 1 name: interval OK
Options Menu: Constants	Fill out the table as below Name Expression P 200000 T 423 R 8314 MM 46 Flowrate 0.01 Dia 0.05 C $P/(RT)$ area $\pi * \text{Dia}^2 / 4$ rho $MM * C$ vel Flowrate/rho/area OK
Options Menu: Scalar Expressions	Define rate = $52.7 * u_1^2 / (1 + 0.013/u_1)$
Physics Menu: Boundary settings	Select domains: 1 and 2 (hold down Ctrl key) Select Neumann boundary conditions Leave defaults $q = 0$ $g = 0$ OK
Physics Menu: Sub- domain settings	Select domain 1 F tab; set $F_1 = -\text{rate} * (1 + u_1/C)$; $F_2 = \text{rate} * (1 - u_2/C)$; $F_3 = \text{rate} * u_3/C$; $F_4 = u_3/C$ Init tab; set $u_1(t_0) = C$; $u_3(t_0) = \text{vel}$. OK
Mesh menu: mesh parameters	Set maximum element size 1 Hit remesh. OK
Solve menu: solver parameters	Time-dependent solve. Enter on General tab Times: linspace (0, 10, 100) Solve. OK
Post-processing: Cross-section plot parameters	Point tab. Accept the default of u_1 General tab. Accept the default of all times OK

It may surprise you, but because the above system is linear, it has a general, analytic solution. Though general, it lends little insight into the dynamics of the system. Plot the graph of concentrations versus time for the initial value problem. Start with pure $C_A = 1$ with parametric values $k_1 = 1$ Hz, $k_2 = 0$ Hz, $k_3 = 2$ Hz, $k_4 = 3$ Hz and plot the graph versus time of concentrations.

5. Method 3: Numerical integration of ordinary differential equations

In the previous section, numerical integration was treated by marching methods, commonly referred to as “time-stepping,” although in the reactor

design application, it was clearly spatial integration. In marching methods, the unknowns are found *sequentially*. The other common method for integration is to approximate the ODE and solve *simultaneously* for the unknown dependent variables at the grid points. With marching methods, all solutions must be initial value problems (IVP). The number of initial conditions must match the order of the ODE system. But for second order and higher systems, a second type of boundary condition is possible — the boundary value problem (BVP), where in 1-D, there are conditions at the initial and final points of the domain. Hence, these are two point boundary value problems. Marching methods can laboriously treat BVPs by shooting — artificially prescribing an IVP and guessing the initial conditions that satisfy the actual BVP by trial and error. In higher dimensional PDEs, a BVP specifies conditions on the boundaries of the domain.

One of the major advantages of the finite element method is that it naturally solves two-point BVPs. As an example, the reaction and diffusion equation in 1-D is

$$\frac{D}{L^2} \frac{\partial^2 u}{\partial x^2} = R(u), \quad (21)$$

where u is the concentration of the species, D is the diffusivity, L is the length of the domain, $R(u)$ is the disappearance rate by reaction, and x is the dimensionless spatial coordinate. If the unknown function $u(x)$ is approximated by discrete values $u_j = u(x_j)$ at the grid points $x = x_j = j\Delta x$, then with central differences, the system of equations becomes

$$\sum_{j=1}^N M_{ij} u_j = \frac{L^2 \overline{\Delta x}^2}{D} R_i, \quad (22)$$

where M_{ij} is a tridiagonal matrix with the diagonal element -2 , and 1 on the super and subdiagonals:

$$M = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots \\ 1 & -2 & 1 & 0 & \cdots \\ 0 & 1 & \ddots & \ddots & \ddots \\ 0 & 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}, \quad (23)$$

and $R_j = R(u_j)$. This system can be solved by iteration for u_i^n by matrix inversion, where n refers to the n -th guess:

$$u_i^n = \frac{L^2 \overline{\Delta x}^2}{D} \sum_{j=1}^N M_{ij}^{-1} R_j, \quad (24)$$

and $R_j = R(u_i^{n-1})$. For either IVP or BVP, the appropriate rows of the matrix M in (23) can be altered to accommodate the boundary conditions. As written, (23) supposes $u = 0$ at both $x = 0$ and $x = 1$. This is a Dirichlet type boundary condition, and is the natural boundary condition for finite difference methods — natural because it occurs if no effort is made to overwrite rows of (23) with specified boundary conditions.

We will now illustrate the solution of (21) with COMSOL Multiphysics on a small 1-D domain with first order reaction $R(u) = ku$ and representative values for the resulting dimensionless parameter, the Damkohler number:

$$Da = \frac{kL^2}{D} = \frac{(10^{-3} \text{ s}^{-1})(10^{-3} \text{ m}^{-1})^2}{1.2 \times 10^{-9} \text{ m}^2/\text{s}} = 0.833, \quad (25)$$

and with boundary conditions $u = 1$ at $x = 0$ and no flux at $u = 1$.

This exercise interacts with MATLAB to explore the structure of a COMSOL Multiphysics representation of solution data and model layout. In windows, COMSOL Multiphysics with MATLAB is a desktop icon option, if you have a MATLAB licence. In UNIX/linux, the equivalent functionality can be launched with

```
comsol matlab path-ml nodesktop-ml nosplash
```

from linux command line. The “matlab” argument tells femlab to launch a matlab command window. The “path” argument sets up the matlab command window with access to the COMSOL library of commands.

First launch COMSOL Multiphysics and enter the Model Navigator. Follow the steps in Table 8. This application mode gives us one dependent variable u , but in a 1-D space with coordinate x . h and r are the two handles on Dirichlet BCs in Coefficient Mode. If you want to set u to a given value U_0 on a boundary, then it is accomplished with setting $h = 1$ and $r = U_0$. Clicking on the triangle symbol creates a default mesh (15 elements) and the triangle with the embedded upside-down triangle to refines the mesh (30 elements).

You should get a graph with the information as in Figure 5. Clearly the desired boundary conditions are met: $u = 1$ at $x = 0$ and the slope vanishes at $x = 1$. But did COMSOL Multiphysics solve the problem we thought we posed?

Now resolve with the stationary nonlinear solver. First note by View Log on the Solver Menu that COMSOL Multiphysics takes thirteen iterations to converge. Do you notice that the final value has dropped from 0.86 to 0.69? One might wonder why there is a difference. The linear (coefficient

Table 8. ODE example of a two-point boundary value problem in a reaction-diffusion system.

Model Navigator	Select 1-D. COMSOL Multiphysics:PDE modes:PDE, <i>coefficient</i> form Set dependent variable: u Select Element: Lagrange-Linear. OK
Draw Menu	Specify objects: Line. Coordinates pop-up menu. x : 0 1 name: interval OK
Physics Menu: Boundary settings	Select domains 1 Check Dirichlet and set $h = 1$; $r = 1$ Select domain 2 Select Neumann boundary conditions. OK
Physics Menu: Subdomain settings	Select domain 1 Set $C = -1$; $f = 0.833 * u$; $d_a = 0$ Select Init tab; set $u(t_0) = 1 - x$ OK
Meshing	Click on triangle symbol to mesh
Solve menu: solver parameters	Note stationary linear default. OK General tab. Set solution form to “Coefficient” Solve with button (=)
Post-processing: Point evaluation	Select Boundary 2 and expression u . Reports: Value: 0.861167, Expression: u , Boundary: 2

form) solver only evaluates $R(u)$ once at the initial condition $u(t_0) = 1 - x$ and thus only needs one iteration of (24). The nonlinear solver evaluates $R(u)$ for each iteration at the old estimate for u . Thus, the nonlinear solver might “forget” the initial guess completely after a number of iterations as it homes in on a converged solution.

Let’s test this explanation. Changing the initial condition should change the stationary linear solution. Return now to Subdomain settings and try the initial condition $u(t_0) = 1$. What final value do you get for $u(x = 1)$? Now try the stationary nonlinear solver. Do you get the same solution as with the other initial condition?

This example should illustrate the importance of selecting the right solver for your equations. If there is any dependence of f on the dependent variables, then the stationary nonlinear solver should be used. The linear solver is faster, but it also presumes that the coefficients of the PDE do not depend on the dependent variable u (else the problem would be nonlinear). When in doubt, use the nonlinear solver. After all, (21) with $R(u) = ku$, is a *linear* problem, but COMSOL Multiphysics only finds the correct steady state solution with the nonlinear solver! The slow convergence rate is also the consequence of the form of the model — general mode with the exact Jacobian solver option for the nonlinear solver converges in two iterations to the correct profile.

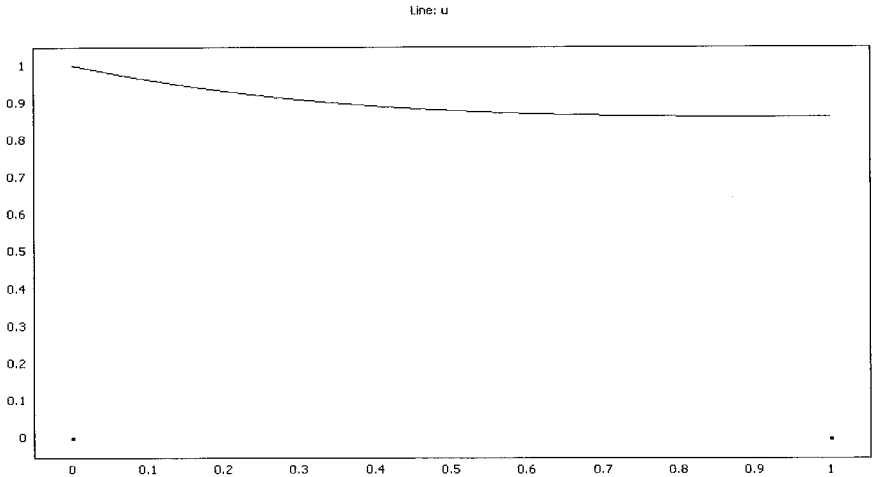


Figure 5. Concentration profile at steady state.

We argued that (22) is the finite difference matrix equation for this problem, yet later applied the argument that (24) should describe the COMSOL Multiphysics finite element problem. Because we used Lagrange linear elements, in this special case the finite element and finite difference matrix operators coincide, up to the boundary conditions. To see this, we will take a foray into the MATLAB representation of COMSOL Multiphysics problems.

Pull down the File Menu and select Export FEM structure as “fem.” This puts the current solution as a MATLAB data structure in the MATLAB workspace. We can then manipulate it using the built-in MATLAB functions and commands, as well as the special function set of COMSOL Multiphysics.

In your MATLAB workspace, try the commands

```
>>x=fem.mesh.p;
>>u=fem.sol.u;
>>plot(x,u)
```

This should pop up a MATLAB Figure plotting the solution u versus the array of mesh points. No doubt your plot looks scrambled. This is because COMSOL Multiphysics stores the mesh points and the associated solution variables so as to make the specification of the matrix equations sparse and compact. We can make sense of the solution by ordering the mesh points using the MATLAB sort command and the solution.

In your MATLAB workspace, try the commands

```
>>[xx, idx]=sort(x);  
>>plot (xx, u(idx))
```

The final MATLAB manipulation we will consider here is interrogation of the finite element matrix. The fem structure does not hold the finite element stiffness matrix, but rather contains the information necessary for FEMLAB functions to construct it. This activity is a vital part of the finite element method, and the FEMLAB function that does it is called `assemble`. Type in the command below:

```
>>[K,L,M,N]= assemble(fem);  
>>K'/15
```

This plot should resemble Figure 5, with the exception that it represents your last COMSOL Multiphysics solution. In fact, we can only make sense of the solution format of the `fem` structure so readily because this is a single dependent variable, one-dimensional problem. Otherwise, multiple variables and dimensions leave a mesh and solution structure that only COMSOL Multiphysics tools/functions can readily decode. Figure 6 shows the sparsity structure generated by the MATLAB command `spy(K')`. It is instructive to compare this with (23), since it is clear that 1-D Lagrange-linear elements with uniform grid are comparable to finite difference methods in the formulation of the matrix equations.

You should now see a MATLAB sparse representation of a matrix, all of the elements of which are 1, -2 , and 1, arranged on different diagonals. This is the stiffness matrix of the finite element method, and up to the ordering of the unknowns, is equivalent to (23). If you return to the Subdomain Settings, element tab, and select Lagrange quadratic elements, and repeat the solution, exporting FEM and `assemble K` as above, you will note that although sparse, the matrix is distinctly different from the Lagrange linear elements.

Exercise.

- 1.6. The coefficient form has a PDE term αu . Repeat the implementation of the reaction-diffusion example, but this time entering $\alpha = 0.833$ and $f = 0$ for the subdomain settings. Now compare the stationary linear and nonlinear solver solutions. Can you explain why this formulation leads to this result? What effect does this formulation of the problem have on the stiffness matrix K . Can you think of a difficulty that

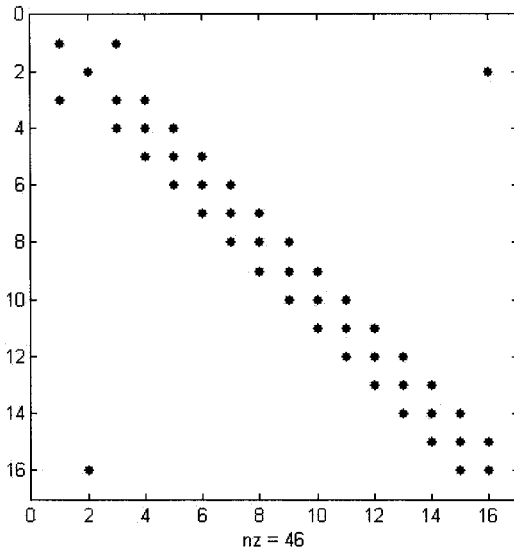


Figure 6. Sparsity structure of K' from the matlab command `spy(K')`.

might occur if the Da is chosen so that the diagonal element is nearly zero in magnitude, i.e. $Da\Delta x^2 = 2$?

6. Method 4: Linear systems analysis

Central to MATLAB, and hence to COMSOL Multiphysics, is linear systems analysis. In this section, we will briefly review the concepts of linear operator theory — typically lumped as “matrix equations” in undergraduate engineering mathematics modules. The good news is that it is not necessary to do any matrix manipulations yourself. That was the *raison d’être* for MATLAB: to serve as a user interface to libraries of subroutines for engineering matrix computations. It should be noted that COMSOL Script would equally well serve this purpose in the examples in this chapter. Much of the history of scientific computing is encapsulated in efficient and sparse methods for matrix computations. An excellent guide to matrix computations, but surely for experts, is the book of Golub and Van Loan [3]. However, at the introductory level to MATLAB, a good and readable survey can be found in the up-to-date book by Hanselman and Littlefield [4].

Briefly, the standard matrix equations look like this:

$$\begin{aligned}
 a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1, \\
 a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2, \\
 a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3, \\
 &\vdots \\
 a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M.
 \end{aligned} \tag{26}$$

Here there are N unknowns x_j which are related by M equations. The coefficients a_{ij} are known numbers, as are the constant terms on the right hand side, b_i . In engineering, models are frequently derived that satisfy such linear systems of equations. Mass and energy balances, for instance, commonly generate such sets of linear equations.

Solvability

When $N = M$, there are as many constraints as there are unknowns, so there is a good chance of solving the system for a unique solution set of x_j 's. There can fail to be a unique solution if one or more of the equations is a linear combination of the others (row degeneracy) or if all the equations contain only certain combinations of the variables (column degeneracy). For square matrices, row and column degeneracy are equivalent. A set of degenerate equations are termed **singular**. Numerically, however, at least two additional things can go wrong:

- While not exactly linear combinations of each other, some of the equations may be so close to linearly dependent that within round-off errors on the computer they are.
- Accumulated round-off errors in the solution process can swamp the true solution. This frequently occurs for large N . The procedure does not fail, but the computed solution does not satisfy the original equations all that well.

Guidelines for linear systems

There is no “typical” linear system of equations, but a rough idea is that round-off error becomes appreciable:

- N as large as 20–50 can be solved by normal methods in single precision without recourse to specialist correction of the two numerical pathologies.
- N as large as several hundred can be solved by double precision.

- N as large as several thousand can be solved when the coefficients are sparse (i.e. most are zero) by methods that take advantage of sparseness. MATLAB has a special data type for sparse matrices, and a suite of functions that exploit the sparseness.

However, in engineering and physical sciences, there are problems that by their very nature are singular or nearly singular. You might find difficulty with $N = 10$. Singular value decomposition is a technique which can sometimes treat singular problems by projecting onto nonsingular ones.

Common tasks in numerical linear algebra

Equation (26) can be succinctly written as a matrix equation (cf. equation (22)).

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad (27)$$

- Solution for the unknown vector \mathbf{x} , where \mathbf{A} is a square matrix of coefficients, and \mathbf{b} is a known vector.
- Solution with more than one \mathbf{b} vector with the matrix \mathbf{A} held constant.
- Calculation of the matrix \mathbf{A}^{-1} , which is the inverse of a square matrix \mathbf{A} .
- Calculation of the determinant of a square matrix \mathbf{A} .
- If $M < N$, or if $M = N$ but the equations are degenerate, then there are effectively fewer equations than unknowns — an underdetermined system. In this case, either there can be no solution, or there is more than one solution vector \mathbf{x} . The solution space consists of a particular solution \mathbf{x}_p plus any linear combination of typically $N-M$ vectors called the nullspace of \mathbf{A} . The task of finding this solution space is called *singularvalue decomposition*.
- If $M > N$, there is, in general, no solution vector x to (26). This overdetermined system happens frequently, and the best compromise solution that comes closest to satisfying the equations is sought. Usually, the closeness is “least-squares” difference between the right and left hand sides of (26).

Matrix computations in MATLAB

Matrix inversion is easily entered using the `inv(matrix)` command. Solution of matrix equations is represented by the matrix division `\` operator as here:

```
>> A=[ 3 -1 0; -1 6 -2; 0 -2 10];
>> B=[1; 5; 26];
```

```
>> X=A\B
X =
1.0000
2.0000
3.0000
```

Determinants

Determinants are used in stability theory and in assessing the degree of singularity of a matrix. Why do you need to know the determinant? Most of the time, you want to know when a determinant is zero. However, when the determinant is zero, or numerically close to zero, it is numerically difficult to compute due to “round-off” swamping effects mentioned earlier. This is yet another application for singular value decomposition.

MATLAB computes determinants by the simple function `det(A)`. Either enter by hand the matrix below at the MATLAB command line, or cut and paste from the file `matrix2.dat`:

```
>> A=[0.45, -0.244111, -0.0193373, 0.323972, -0.118829;
-0.244111, 0.684036, -0.103427, 0.205569, 0.00292382;
-0.0193373,-0.103427,0.8295, 0.0189674, -0.011169;
0.323972, 0.205569,0.0189674, 0.659479, 0.197388;
-0.118829,0.00292382,-0.011169, 0.197388, 0.776985]
```

The determinant is found from the `det` command

```
>> det(A)

ans =
-1.9682e-008
```

Principal axis theorem: Eigenvalues and eigenvectors

MATLAB has built-in functions for computing the eigenvalues and eigenvectors of a matrix:

```
>> eig(A)

ans = -0.0000
0.7000
0.8000
0.9000
1.0000
```

The `eig()` function can also return the eigenvectors as the columns of the matrix V when called as below:

```
>> [V,D]=eig(A)
```

V =

```
-0.6836  0.0000  -0.5469  -0.4785  -0.0684
-0.4181  0.6162   0.1831   0.4530  -0.4547
-0.0837  0.4003   0.6189  -0.6232   0.2479
 0.5409  0.2582  -0.2415  -0.4042  -0.6474
-0.2416  -0.6272   0.4755  -0.1190  -0.5550
```

D =

```
-0.0000  0  0  0  0
 0  0.7000  0  0  0
 0  0  0.8000  0  0
 0  0  0  0.9000  0
 0  0  0  0  1.0000
```

The `eigs()` function is a variant of `eig()` which computes a specific number of eigenvalues/eigenvector pairs for sparse matrices. Its use will be demonstrated in the next subsection in conjunction with COMSOL Multiphysics.

The matrix A has a determinant that is little different from zero and a single eigenvalue that is effectively zero. The eigenvector associated with it is effectively the null space of A — the direction that gets mapped to zero:

```
>> A*V(:,1)
```

```
ans =
1.0e-007 *
0.2669
0.1633
0.0327
-0.2112
0.0943
```

All the other eigenvectors can be verified by the property that they map onto themselves, scaled by the eigenvalue, for instance:

```
>> A*V(:,2) ./ V(:,2)
```

```
ans =
0.7000
0.7000
0.7000
0.7000
0.7000
```

In MATLAB, the ./ division operator is element-by-element division. The colon above refers to the whole of the column.

Because the system is nearly singular, we should not be surprised that the solutions to any matrix equation involving it are poorly conditioned. For instance,

```
>> B=[0; 1; 0; 1; 0];
>> A\B
```

```
ans =
1.0e+006 *
2.1487
1.3142
0.2631
-1.7001
0.7593
```

Since the elements of A are of order one, the forcing vector B is of order one, one would expect the solution to (27) to be order one, not order one million. For chemical engineers, this is like being told that a mass balance involves input flow rates of about 1 kg/hr, constraints on mass balances with appreciable fractions in the splitters (order one), and that the solution mass flow rates are about one million kg/hr for internal streams. Not likely. Yet this is the solution proposed by a nearly singular matrix.

Singular value decomposition (SVD)

SVD offers a better solution in many respects. All matrices have a unique decomposition, similar to the principal axis theorem for eigenvalues and eigenvectors

$$A = U \cdot \text{diag} \cdot V^T, \quad (28)$$

where U and V are square real and orthogonal. diag is a diagonal matrix which contains the singular values. In terms of U , V , and diag , the system (27) is readily solved

$$A^{-1} = V \cdot [1/\text{diag}_j] \cdot U^T, \quad (29)$$

U and V being orthogonal means that their transposes are also their inverses. The inverse of a diagonal matrix is just the reciprocal of the diagonal elements. So the only time we have a problem solving the system is when one or more of the singular values (diag_j), relative to the largest, is close to zero. It follows that $(1/\text{diag}_j)$ is a very large number, which distorts our numerical solution, sending it off to infinity along a direction which is

spurious. A good approximation is to throw these spurious directions away completely by setting $(1/diag_j)$ for the offending singular values to zero! The vector,

$$x = V \cdot [1/diag_j] \cdot U^T b \quad (30)$$

with this substitution for nearly zero elements, should be the smallest in magnitude to approximately satisfy the equations.

In the case of our example matrix A , the MATLAB command `svd()` gives the singular values if called with one output, and the three matrices U , $diag$, V if called with three:

```
>> [U,D,V]=svd(A)
```

```
U =
```

```
-0.0684 -0.4785 0.5469 0.0000 -0.6836
-0.4547 0.4530 -0.1831 -0.6162 -0.4181
0.2479 -0.6232 -0.6189 -0.4003 -0.0837
-0.6474 -0.4042 0.2415 -0.2582 0.5409
-0.5550 -0.1190 -0.4755 0.6272 -0.2416
```

```
D =
```

```
1.0000 0 0 0 0
0 0.9000 0 0 0
0 0 0.8000 0 0
0 0 0 0.7000 0
0 0 0 -0.0000 1.0000
```

```
V =
```

```
-0.0684 -0.4785 0.5469 0.0000 -0.6836
-0.4547 0.4530 -0.1831 -0.6162 -0.4181
0.2479 -0.6232 -0.6189 -0.4003 -0.0837
-0.6474 -0.4042 0.2415 -0.2582 0.5409
-0.5550 -0.1190 -0.4755 0.6272 -0.2416
```

Reassuringly, $U = V$ since the matrix A is symmetric.

The SVD prescription for solution with smallest magnitude is implemented as follows:

```
>> ss=[1. 1./0.9 1./0.8 1./0.7 0];
>> dinv=diag(ss);
>> V*dinv*U'*B
```

```
ans =
```

0.0893
 1.2820
 0.1479
 1.0317
 -0.2130

This is a far more physically acceptable solution, for instance, for internal mass flow rates in the hypothetical mass balance discussed above.

This excursion into linear systems theory is important for modeling with COMSOL Multiphysics because finite element methods are matrix based. When the generalized stiffness matrix becomes nearly singular, COMSOL Multiphysics may not be providing a satisfactory solution. These matrix computations and their sparse implementations in MATLAB can readily serve as diagnostics for the health of the COMSOL Multiphysics solution. They also provide an insight into the natural dynamics of the system through the eigen analysis of the operator. These ideas will be made concrete with an example computed as a COMSOL Multiphysics model in the next subsection.

6.1. Heat transfer in a nonuniform medium

The steady state heat transfer equation is commonly met in engineering studies as the simplest PDE that is analytically solvable: Poisson's equation. Nevertheless, series solutions for complicated geometries may be intractable. The author has recently shown that some series so derived are purely asymptotic and poorly convergent [5]. Consequently, numerical solutions are likely to be better behaved than series expansions. Furthermore, any variation on the processes of heat transfer may destroy the analytic structure. In this section, we will consider the typical one-dimensional heat transfer problem in a slab of nonuniform conductivity and a distributed source that is differentially heated on the ends:

$$-\frac{d}{dx} \left(k \frac{dT}{dx} \right) = f(x), \quad (31)$$

$$T|_{x=0} = 1 \quad T|_{x=1} = 0.$$

Launch COMSOL Multiphysics with MATLAB and enter the Model Navigator: Follow Table 9 for the instruction set to set up the heat transfer problem in a medium with a distributed source. The solution should be found fairly quickly resulting in a nearly linear profile with almost a slope of -1 . The Solver Log shows two step solution, which, since the problem

Table 9. Heat transfer in a nonuniform medium — Distributed source.

Model Navigator	Select 1-D. COMSOL Multiphysics: Heat Transfer:Conduction: Steady state analysis. Set dependent variable: u Select Element: Lagrange-Linear. OK
Draw Menu	Specify objects: Line. Coordinates pop-up menu. x : 0 1 name: interval OK
Physics Menu: Boundary settings	Select boundary selection 1 Set boundary condition: temperature; $T_0 = 1$ Select boundary selection 2 Set boundary condition; $T_1 = 0$. OK
Physics Menu: Subdomain settings	Select domain 1 Set $k = 1$; $Q = -x * (1 - x)$ Select Init tab; set $T(t_0) = 1 - x$. OK
Meshing	Click on triangle symbol to mesh
Solver	Click on the solve (=) button to solve
Post-processing: Data display	Specify $x = 0.5$ Value: 0.474097 [K], Expression: T , Position: (0.5)

is *linear*, is guaranteed. Verify that $T|_{x=0.5} = 0.474097$. This problem has an analytic solution with $T|_{x=0.5} \sim 0.475$:

$$T = 1 - \frac{13}{12}x + \frac{x^3}{6} - \frac{x^4}{12}. \quad (32)$$

Now try $k = 1 - x/2$. There is also an analytic solution in this case, but in the complex numbers requiring logarithms in the complex plane and a branch cut. The analytic solution gives $T|_{x=0.5} \sim 0.550$. How good is your solution?

Now for the linear systems theory. Pull down the File menu and select Export FEM structure as “fem.” This is the second time we have exported the FEM structure, so it might be useful to explain a bit more about it. As with any MATLAB variable, if you type the variable name on the command line, MATLAB will either provide the value of the variable or show its data structure. Try

```
>> fem
```

```
fem =
```

```
version: [1x1 struct]
  appl: {[1x1 struct]}
  geom: [1x1 geom1]
  mesh: [1x1 femmesh]
  border: 1
outform: 'general'
  form: 'general'
```

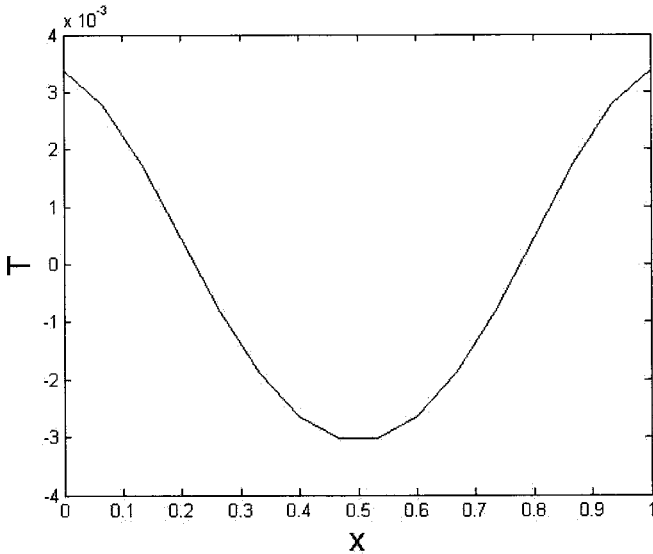


Figure 7. Projection solution for the purely Neumann solution to the nonuniform conductivity and distributed source heat transfer problem.

```

units: 'SI'
equ: [1x1 struct]
bnd: [1x1 struct]
draw: [1x1 struct]
xmesh: [1x1 com.femlab.xmesh.Xmesh]
sol: [1x1 femsol]

```

Of course, you can go further down the FEM structure and investigate branches, twigs, and leaves on the tree. We have already pruned

```

    fem.sol.u ,
    fem.mesh.pl .

```

The different branches contain a complete COMSOL Multiphysics model — specifying equations (`fem.appl{1}.equ`) and boundary conditions (`fem.appl{1}.bnd`) for the geometry held in `fem.geom`. Some of these fields are writable by MATLAB assignment statements, like any other variable. Interrogate the boundary conditions, for instance, with

```
>> fem.appl{1}.bnd
```

```
ans =
```

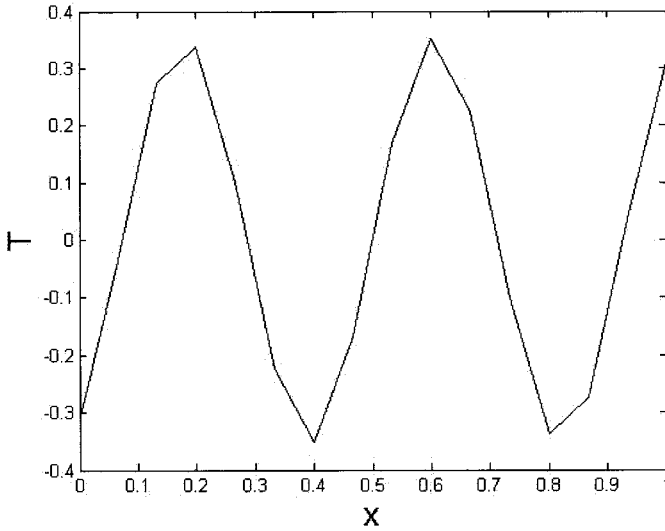


Figure 8. Smallest nonzero eigenvalue/eigenvector pair for the nonuniform conductivity and distributed source heat transfer problem.

```
type: 'T'  
T0: {[0] [1]}  
ind: [2 1]  
  
>> fem.appl{1}.bnd.T0{2}  
  
ans =  
  
1  
  
>> fem.appl{1}.bnd.T0{2}=10  
  
ans =  
  
10
```

It is now possible to upload this FEM structure to COMSOL Multiphysics using the FILE menu Import field. If you do this, check under Physics Menu — Boundary Settings, you should find that the boundary condition on boundary 1 is now $T = 10$. Clearly, we could make this change to the model far more readily by using the pull down menus in the

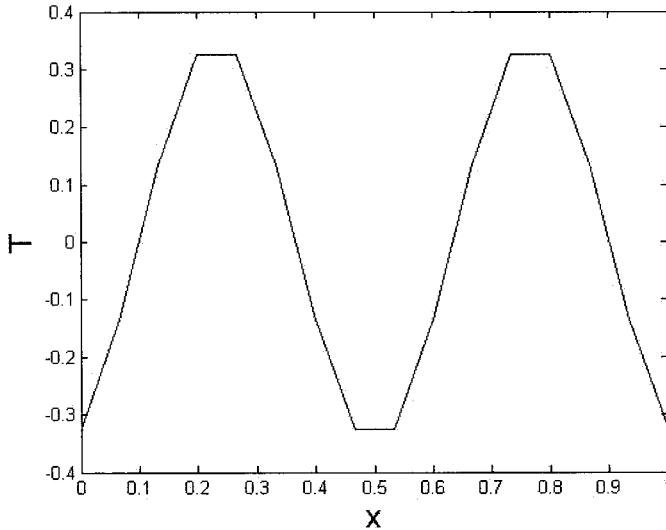


Figure 9. Next smallest nonzero eigenvalue/eigenvector pair for the nonuniform conductivity and distributed source heat transfer problem.

GUI. Nevertheless, altering FEM structures in MATLAB is a very powerful feature. In later chapters, we will use this feature to good effect. Given the complexity of the FEM data structure, however, it is best to keep our alterations to simple features.

You can now manipulate the solution in MATLAB. As in the last section, you can assemble the stiffness matrix and analyze it with `eigs`. As K is sparse, you can find the smallest six eigenvalues in magnitude with the `eigs` command.:

```
>>[K,L,M,N] = assemble(fem);
>>dd=eig(K)
>>dd=eigs(K, 6,-0.1)
>>[V,D]= eigs(K, 6, -0.1)
```

Note that K has one zero eigenvalue, and that all its eigenvalues are negative otherwise. This should not worry you, as COMSOL Multiphysics implements its boundary conditions through the block matrix N and auxiliary forcing vector M . It could replace rows of K and elements in L to approximate boundary conditions, but the structure of boundary conditions in COMSOL Multiphysics allows for more general types of boundary

conditions when augmenting the matrix equations with N and M . The fact that K is singular as a block matrix is a consequence of the natural boundary conditions for finite element methods being Neumann conditions (no flux). There are an infinity of solutions to the pure Neumann boundary conditions, as an arbitrary value can be added to any solution and it is still a solution. That K is singular naturally tripped up the author when he first used finite element methods as an undergraduate. Purely Neumann boundary conditions (zero heat flux in this example) are ill-posed. For instance, if you change the boundary conditions in our example to purely Neumann conditions, you should find that the steady state solution is not solvable — leads to a singular matrix. Some reference value (Dirichlet condition) must be specified for there to be a unique solution. Yet MATLAB can solve such a problem by SVD or by the principal axis theorem. Since the matrix K is positive-semi-definite, all its eigenvalues are real. So pseudo-inversion to eliminate the zero eigenvalue of K follows by following the recipe for the pseudo-inverse of the previous section:

```
>>ss=1./dd;
>>ss(6)=0;
>>dinv=diag(ss);
>>uneumann=V*dinv*V'*L
```

Finally, interpreting this solution must be done remembering that the structure of a FEMLAB mesh is not monotonic. These commands plot the solution:

```
>>[xs, idx]=sort(fem.mesh.p);
>>plot(xs, fem.sol.u(idx)) ;
```

Similarly, the approximate Neumann solution found from the projection onto the first five eigenvectors with smallest magnitude nonzero eigenvalues is found from

```
>>plot(xs, uneumann(idx));
```

Figure 7 shows the projection of the solution onto just the first six modes using the pseudo-inverse method. You can think of this plot as the deviation from the linear function satisfying the boundary conditions, i.e. $1 - x$. Furthermore, the eigenvectors can be interpreted the same way:

```
>>col1=V(:,1);
>>plot(xs,col1(idx));
>>col2=V(:,2);
>>plot(xs,col2(idx));
```

It is one of my pet complaints that eigenvalues and eigenvectors are not interpreted in mathematics classes, so the student does not learn why they are taught and thus dismisses them as “esoteric.” The eigenvectors in this problem represent the decaying “modes” as the solution approaches steady state at long times. The eigenvalues are the (exponential) decay rates, in this case, since the COMSOL Multiphysics sign is reversed from common practise. *Negative* eigenvalues would represent growth rates of unstable modes. Clearly, all the modes here are dissipative of energy. That not all analysis share this feature — some are unstable, can lead to physically interesting phenomena (pattern formation, explosion), but also to numerically difficult modelling — nonconvergent computational models.

Please note that in the case of the Neumann solution, any constant value can be added to the solution and it will remain a solution. The eigenvectors are not normalized, so they can be multiplied by any number and still be eigenvectors. Figures 8 and 9 show the two eigenpairs with smallest eigenvalues in magnitude. These are the slowest decaying modes, and therefore the pattern of the expected “standing waves” that disappear last as the steady-state is formed.

References

- [1] M. B. Cutlip and M. Shacham, *Problem Solving in Chemical Engineering with Numerical Methods* (Prentice-Hall, Upper Saddle River, NJ, 1999).
- [2] W. C. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (1971).
- [3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd edn. (Johns Hopkins University Press, Baltimore, London, 1996).
- [4] D. Hanselman and B. Littlefield, *Mastering MATLAB 7: A Comprehensive Tutorial and Reference* (Prentice-Hall, Saddle River, NJ, 2005).
- [5] W. B. Zimmerman, On the resistance of a spherical particle settling in a tube of viscous fluid, *International Journal of Engineering Science* **42**(17–18) (2004) 1753–1778.