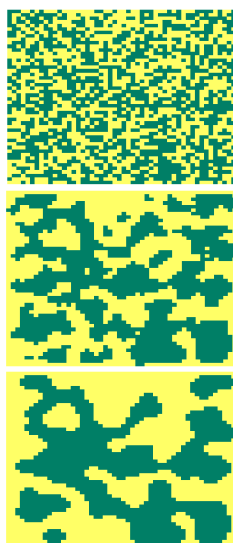


# Numerical Methods for Chemical Engineers: A MATLAB-based Approach



Raymond A. Adomaitis  
*Department of Chemical & Biomolecular Engineering and  
Institute for Systems Research  
University of Maryland  
College Park, MD 20742  
adomaiti@umd.edu – thinfilm.umd.edu*



This work is licensed under Creative Commons  
Attribution-NonCommercial-ShareAlike 4.0 International License  
<https://creativecommons.org/licenses/by-nc-sa/4.0/>

December 29, 2016

# Contents

<b>1</b>	<b>MATLAB overview</b>	<b>8</b>
1.1	An introduction to MATLAB	8
1.1.1	Installing and starting MATLAB	8
1.1.2	About MATLAB	9
1.1.3	What are typical MATLAB applications?	9
1.1.4	Getting started	10
1.1.5	Variables	10
1.1.6	Data types	11
1.1.7	The workspace	12
1.1.8	Help	12
1.2	Working with arrays	12
1.2.1	Arrays	12
1.2.2	The : operator	14
1.2.3	Matrix transpose	15
1.2.4	Array sums, differences, products	16
1.2.5	Other mathematical functions	17
1.3	Plotting	18
1.3.1	Multiple plot figures	19
1.3.2	Multiple plots per figure	19
1.4	Scripts, functions, and files	19
1.4.1	MATLAB and the file system	19
1.4.2	Addpath	20
1.4.3	Saving plots for webpages, documents, and more	20
1.4.4	Reading/writing data to the command window	20
1.4.5	Script files	21

<i>CONTENTS</i>	2
1.4.6 Functions . . . . .	22
1.4.7 Varying the number of input/output parameters . . . . .	24
1.4.8 Inline and anonymous functions . . . . .	25
1.5 Logical operations . . . . .	26
1.5.1 Relational operators . . . . .	26
1.5.2 Precedence . . . . .	27
1.5.3 Branching structures . . . . .	28
1.5.4 Multiple conditions per if statement . . . . .	29
1.5.5 A closer look at working with arrays . . . . .	29
1.5.6 Switch . . . . .	30
1.5.7 Find . . . . .	30
1.6 Looping constructs . . . . .	30
1.6.1 Nested loops . . . . .	31
1.6.2 While loops . . . . .	32
1.7 Elements of programming in MATLAB . . . . .	32
1.7.1 Combining decisions and mathematical operations . . . . .	33
1.7.2 Linear interpolation . . . . .	34
1.7.3 Search for a minimum value . . . . .	34
1.7.4 The quadratic equation solver . . . . .	35
1.7.5 Quadratic equation solver function . . . . .	36
1.7.6 Size selection . . . . .	37
1.8 Review problems . . . . .	38
<b>2 Object-oriented programming in MATLAB</b>	<b>43</b>
2.1 Object-oriented programming concepts . . . . .	43
2.1.1 The dimensional class . . . . .	44
2.1.2 The myline class . . . . .	45
2.2 Inheritance . . . . .	49
<b>3 Case Study: The associative array class</b>	<b>52</b>
3.1 Review problems . . . . .	53
<b>4 Linear systems</b>	<b>54</b>
4.1 Writing systems of linear equations in matrix form . . . . .	54
4.2 Matrix multiplication . . . . .	54

<i>CONTENTS</i>	3
4.3 Solving $Ax=b$	55
4.3.1 Factorization by Gaussian Elimination	55
4.3.2 An example of solving sets of linear equations	56
4.4 More equations than unknowns	57
4.5 Linear vector spaces	58
4.5.1 The Gram-Schmidt orthogonalization procedure	59
4.6 Least squares	60
4.6.1 Linear regression models	61
4.6.2 Computational example: Linear regression	63
4.7 The singular value decomposition	65
4.7.1 Computational example: orthogonalization using the SVD	66
4.7.2 Matrix condition number	67
4.7.3 Computational example: solving linear systems	68
4.8 Review problems	69
<b>5 Case Study: Nonadiabatic methane/octane flash drum</b>	<b>72</b>
5.1 Model derivation	72
5.2 Solution computed using partial pivoting	74
5.2.1 The solution without pivoting	75
5.3 Number of equations $\neq$ number of unknowns	76
5.4 Review problems	77
<b>6 Case Study: Underdetermined systems</b>	<b>78</b>
6.1 Diesel fuel blending	78
6.2 Chemical reaction stoichiometry	79
<b>7 Case Study: Enzyme kinetics</b>	<b>82</b>
7.1 Michaelis-Menten kinetics	83
7.1.1 Reaction data	84
7.2 Review Problems	84
<b>8 Case Study: Least squares fit of CVD reactor data</b>	<b>86</b>
8.1 The least squares fit	86
8.2 Using linearsystemsolver	87
8.3 Using the myline class	87

<b>9</b>	<b>Linear ordinary differential equations</b>	<b>89</b>
9.1	Scalar ordinary differential equations . . . . .	89
9.1.1	A non-homogeneous problem . . . . .	91
9.2	Sets of linear ODEs . . . . .	91
9.2.1	Computing eigenvalues/vectors . . . . .	92
9.2.2	Eigenvectors and the phase space . . . . .	94
9.2.3	Real and distinct eigenvalues ( $\Delta > 0$ ) . . . . .	95
9.2.4	Repeated eigenvalues ( $\Delta = 0$ ) . . . . .	96
9.2.5	Complex eigenvalues ( $\Delta < 0$ ) . . . . .	99
9.2.6	Symmetric systems . . . . .	100
9.2.7	Unsymmetric systems . . . . .	101
9.3	Summary of solutions to sets of linear ODEs . . . . .	101
9.4	Boundary-value problems . . . . .	102
9.4.1	Where are the eigenvalues? . . . . .	103
9.4.2	A nonhomogeneous BVP . . . . .	103
9.4.3	A diffusion-convection problem . . . . .	104
9.5	Review problems . . . . .	105
<b>10</b>	<b>Case Study: Quenching dynamics</b>	<b>108</b>
10.1	Transient solution . . . . .	109
10.2	Case $\alpha_c = 0$ . . . . .	110
10.2.1	Scalar ODE solution . . . . .	110
10.2.2	Solution in the phase plane . . . . .	111
10.2.3	Physical interpretation of eigenvectors . . . . .	111
<b>11</b>	<b>Case Study: Isothermal CVD reactor model</b>	<b>113</b>
11.1	CVD deposition rate . . . . .	114
11.2	CVD reactor dynamics . . . . .	115
11.3	Review problems . . . . .	116
<b>12</b>	<b>Nonlinear systems</b>	<b>117</b>
12.1	The generic characteristics of a nonlinear AE model . . . . .	117
12.2	Nonlinear model examples . . . . .	118
12.2.1	Multiple equation models . . . . .	119
12.3	Newton's method . . . . .	120

12.3.1 Quadratic convergence - numerical analysis . . . . .	120
12.3.2 Computational example: Quadratic convergence of Newton's method . . . . .	122
12.4 Multiple equations: The Newton-Raphson method . . . . .	124
12.4.1 Computational example: The Newton-Raphson method . . . . .	124
12.4.2 Jacobian arrays by finite differences . . . . .	125
12.4.3 Our basic Newton-Raphson function . . . . .	126
12.4.4 Jacobian-free formulations . . . . .	127
12.4.5 Krylov subspace methods . . . . .	128
12.5 The naemodel class . . . . .	129
12.6 Sensitivity analysis . . . . .	132
12.7 Continuation . . . . .	133
12.7.1 Computational example: Solutions by continuation . . . . .	134
12.8 Review problems . . . . .	136
<b>13 Case Study: A CVD reactor thermal dynamics model</b>	<b>138</b>
13.1 Steady-state solution using Newton's method . . . . .	141
13.2 Multiple equations: The Newton-Raphson method . . . . .	143
13.3 CVD thermal model derived from the neqmodel class . . . . .	145
13.4 Continuation . . . . .	147
13.5 Linearized system dynamics . . . . .	147
<b>14 Case Study: Adiabatic CSTR with first-order reaction</b>	<b>150</b>
14.1 Turning-point bifurcations . . . . .	152
14.2 Multistability . . . . .	153
14.3 Review problems . . . . .	153
<b>15 Case Study: Flash distillation for a non-ideal mixture</b>	<b>154</b>
15.1 The variables and modeling equations . . . . .	155
15.2 Definition of the model class meohwaterwilson . . . . .	155
15.3 Representative calculations . . . . .	157
15.3.1 Computing phase compositions for fixed temperature and pressure . . . . .	157
15.3.2 Flash drum material balance . . . . .	158
15.3.3 Computing temperature for specified liquid composition . . . . .	158
<b>16 Nonlinear ODE methods</b>	<b>160</b>
16.0.1 An isothermal chemical reactor model . . . . .	161

16.0.2 Inert species . . . . .	163
16.0.3 Time-varying feed composition . . . . .	164
16.1 Our Euler integrator . . . . .	164
16.2 Nonlinear pendulum example . . . . .	165
16.3 Computing time step sizes – the Runge-Kutta methods . . . . .	166
16.4 Numerical integration of ordinary differential equations . . . . .	168
16.4.1 A nonlinear problem . . . . .	170
16.5 Review problems . . . . .	170
<b>17 Orthogonal function sequences</b>	<b>172</b>
17.1 Norms, function spaces, and orthogonal functions . . . . .	172
17.1.1 The optimal projection . . . . .	173
17.1.2 Orthogonal function sequences . . . . .	174
17.1.3 The projection operator . . . . .	175
17.1.4 Gram-Schmidt orthogonalization . . . . .	175
17.1.5 SVD based orthogonalization . . . . .	176
17.2 Numerical representations of basis function sequences . . . . .	178
17.2.1 basisfun: basis function class . . . . .	178
17.3 Sturm-Liouville problems . . . . .	179
17.3.1 Orthogonality of eigenfunctions . . . . .	180
17.3.2 A simple example . . . . .	180
17.3.3 The case $q(x) = 0$ . . . . .	181
17.3.4 Completeness . . . . .	182
17.3.5 Basis functions computed as solutions to a Sturm-Liouville problem . . . . .	183
17.3.6 Another example . . . . .	184
17.4 The Galerkin projection . . . . .	185
17.4.1 Accuracy . . . . .	187
17.5 Assessing discretized solution accuracy . . . . .	187
17.6 The Gibb's phenomenon and filtering . . . . .	187
17.6.1 Filters . . . . .	189
17.6.2 Physical-space filters . . . . .	190
<b>18 Polynomial collocation for nonlinear/linear BVPs</b>	<b>192</b>
18.1 Polynomial interpolation and quadrature . . . . .	192
18.1.1 Uniqueness of the interpolating polynomial . . . . .	193

18.1.2 The Vandermonde array . . . . .	193
18.2 Lagrange interpolation . . . . .	193
18.2.1 Runge's divergence phenomenon . . . . .	194
18.3 Neville's algorithm . . . . .	194
18.4 Discrete differentiation operations . . . . .	196
18.5 Quadrature . . . . .	197
18.6 High-degree interpolation . . . . .	198
18.6.1 2-dimensional interpolation . . . . .	198
18.6.2 Collocation discretization . . . . .	199
18.6.3 Collocation solution error analysis . . . . .	199
18.7 Object classes for quadrature . . . . .	200
18.7.1 quadgrid: quadrature grid class . . . . .	200
18.7.2 Computational example: creating objects of quadgrid class . . . . .	200
18.7.3 scalarfield: scalar field class . . . . .	201
18.7.4 Computational example: creating objects of scalarfield class . . . . .	203
18.7.5 Computational example: using the linearoperator class . . . . .	204
<b>19 Case Study: Tubular reactor boundary-value problem</b>	<b>205</b>
19.1 Exact solution . . . . .	206
19.1.1 BVP collocation solution . . . . .	208
19.1.2 Collocation solution error analysis . . . . .	209
<b>20 Case Study: One-dimensional transient heat transfer</b>	<b>210</b>
20.1 Numerical solution . . . . .	211
20.2 Transport in more complicated geometries . . . . .	212



# Chapter 1

## MATLAB overview

### 1.1 An introduction to MATLAB


We will start by discussing how one accesses MATLAB, including any details particular to the University of Maryland. This will be followed by a quick check of the interactive environments of MATLAB before proceeding to introductory elements of the software. In the notes that follow, we denote MATLAB examples in the following manner

MATLAB examples

||

#### 1.1.1 Installing and starting MATLAB

If you have a personal copy of MATLAB, the installation instructions will come with the software. MATLAB can also be run using the campus Virtual Computer Laboratory (VCL) located at [eit.umd.edu/vcl/](http://eit.umd.edu/vcl/). This link will take you to the home page of the VCL from which you can log into the VCL using your university ID. Once done, click on the MATLAB icon; the MATLAB command line interface should appear after the MATLAB splash screen (these may open *under* the already open VCL windows!).

It is important to understand that the MATLAB application is not actually running on your local computer, and so the current directory (folder) that MATLAB will look into for your scripts and functions will not be found on your computer. To select a folder on your computer (e.g., your desktop), click on the  bottom above the command window next to the "Current Folder" window and select the Desktop folder from Client – *not the Local disk* – both of which are located in the Computer folder.

Among the commands MATLAB accepts are some Unix-like commands; therefore, we can change to our computer's desktop directory directly from the command line by

```
>> cd V:\Desktop % for Mac OS X
```

||

and so now MATLAB will first look to our desktop for .m script and function files and will deposit any plots or saved data files to the same. To make sure everything works, try the following commands

```
>> A = 2
>> b = 3;
>> A + b
>> plot([1,2,3,4],[10,-1,9,2],'-o')
>> print -djpeg matlabFig.jpg
>> quit
```

Much more will be said later regarding MATLAB and your computer's file system.

### 1.1.2 About MATLAB

MATLAB is a environment for scientific computing that is ideal for computations that require extensive use of arrays and graphical analysis of data. Features of MATLAB include the following:

1. MATLAB is a software package that operates as a problem-solving environment; it has a graphical user interface, in which the most prominent feature is a window that interprets user commands.
2. Typical MATLAB commands consist of those that manipulate data structures, and extensive set of mathematical functions, and numerous plotting and data visualization tools.
3. MATLAB commands can be assembled into a script that acts as an interpreted (not compiled) computer program; these scripts are saved as plain text files with a .m extension.
4. MATLAB is particularly good at handling arrays - in fact, the default data type is a double-precision array; array indices begin with 1 (compare to 0 in C, Java, and Python).
5. New functions in MATLAB can be defined to operate exactly as the built-in functions; all function parameters are passed by value to functions (no pointers).
6. MATLAB supports object-oriented programming approaches - classes are defined in terms of sub-directories with names beginning with the @ symbol; these directories typically contain the class constructor, display, accessor, and all other class methods.

### 1.1.3 What are typical MATLAB applications?

1. Solution and analysis of systems of linear equations;
2. Solving ordinary differential equations with numerical integration techniques and plotting the results;
3. Solving large sets of nonlinear equations, such as those resulting from a chemical process mathematical model;
4. Statistical analysis and graphical interpretation of experimental data;
5. Image processing;
6. Process control system synthesis and tuning;
7. Prototyping computer programs that will then be converted to a compiled language to improve its efficiency;

8. Much, much more.

In terms of comparing MATLAB to other computing languages,

- MATLAB is an interpreted language (no compiler); scripts can be saved as .m files;
- Array indices begin with 1 (compare to 0 in C or Java);
- Arrays are passed by value to functions (no pointers);
- Array elements are accessed with the format A(1,2) (compare to the format A[0][1] in C or Java);
- Powerful matrix mathematical functions are built-in (e.g., for Gaussian elimination or least-squares solution methods for linear systems);
- Numerous options for plotting and visualizing data are available in MATLAB.

### 1.1.4 Getting started

After starting MATLAB, the following prompt will appear in the command window:

```
>> ||
```

At this point, MATLAB is ready to begin a computational session. Hitting the return key generates a MATLAB prompt on a new line; valid MATLAB instructions have the following characteristics:

- MATLAB is case sensitive, so the command `plot` is different from `Plot`;
- MATLAB keywords cannot contain spaces; however, MATLAB ignores the number of spaces between keywords and mathematical operations;
- Comments are entered after the `%` symbol;
- `...` is used to continue a line after a line break;
- multiple commands can be entered on a single line by separating each command with a comma or semicolon.

### 1.1.5 Variables

The first important concept in understanding how MATLAB and programming languages work is to recognize that data are represented by **variables** and data are transformed by **functions** and **programs**. A variable has two important characteristics:

1. A name (e.g., *a*)
2. A value (e.g., 2)

In a MATLAB session, a new variable is defined using the *assignment operator* = (we intentionally not call this symbol the “equal sign”).

```
>> a = 2
a =
    2
```

Note how the name and value are echoed in this MATLAB session. Other important characteristics of variables include

- Recall that MATLAB is case sensitive, so the variable name “bob” is different from “Bob”
- Generally, try to avoid variable names that are the same as MATLAB functions (e.g., `sin = 2`); using keywords (e.g., `end`) is not allowed by MATLAB.
- The value can be much more than a single number - the differences are defined by the data types discussed in the following section.

### 1.1.6 Data types

The default data type in MATLAB is a double-precision array. While this data type tends to be used most often in computations, several other data types also are important:

```
>> a = 2 % define a scalar double
a =
    2
>> b = [ 2, 3] % a 1 by 2 array
b =
    2    3

>> s = 'a character string'
s =
a character string
>> s(3) % s is actually an array of single characters
ans =
c
```

as well as MATLAB cell arrays and structures for storing a set of values, possible of mixed data type, under a single variable name such as *cell* arrays

```
>> c = {'elements', 'in a cell array', 2}
c =
    'elements'    'in a cell array'    [2]
```

and *structures*

```
>> t.a = 1
t =
    a: 1
>> t.b = 'a structure'
t =
    a: 1
    b: 'a structure'
```

To check the *class* (the datatype) of some currently defined variable,

```
>> class(t) % returns the data type
ans =
struct
```

### 1.1.7 The workspace

In addition to its powerful library of mathematical and visualization functions, MATLAB also serves as a numerical problem solving environment. An important concept is that of the *workspace*, a virtual storage area that contains the current variable names and values. The command shell for each problem-solving session has its own workspace; the contents of a workspace can be saved using the `save` command and then retrieved with the `load` command.

During a MATLAB session, all variable names and values are stored in the session workspace. The current workspace variables can be listed using the `who` command; they can be cleared using the `clear varname` for the individual variables named `varname`; the entire workspace can be cleared using the `clear` command by itself. We will see that this notion of workspace will be extended by functions, because each function has its own workspace.

To investigate the workspace concept, try the following sequence of MATLAB commands in which two variables are defined and the `who` command is used to display the variables currently defined in the workspace.

```
>> a = 1;
>> b = 2;
>> a % echo A
>> who % show defined variables
Your variables are:
a  ans b
>> clear % clears all the variable definitions from the current workspace
```

**Question:** What will be displayed after this script is run if we use the `who` command again?

### 1.1.8 Help

MATLAB is an environment and programming language that contains numerous built-in functions and features and is a computational system that continues to evolve. Fortunately, all MATLAB documentation is available on-line; the starting point for any help is the keyword `help`.

## 1.2 Working with arrays

### 1.2.1 Arrays

*Array* and *matrix* are interchangeable terms that describe a set of numbers arranged in a grid-like fashion. A *vector* also is an array, however, vectors generally refer to one-dimensional arrays. In this book, the

default arrangement of a vector is a column, which differs from the row arrangement of MATLAB's default form for a one-dimensional array.

Arrays are defined and accessed based on the following concepts:

1. Array indices are nonzero positive integers beginning with 1;
2. Column elements are separated by a comma (,) and row elements are separated by a semicolon (;) when arrays are defined;
3. Blank spaces are equivalent to commas when defining arrays;
4. Arrays must have a rectangular shape;
5. When defined, array elements normally are contained between the [ and ] symbols (we will see that this is not always the case, especially when using the : operator, below).
6. The element in row  $i$  and column  $j$  of matrix  $\mathbf{A}$  is accessed by  $A(i,j)$

Now consider some basic MATLAB operations that define or access elements in array form. We begin with two, one-dimensional arrays each containing three elements:

```
>> a = [1, 2, 3] % row array
a =
     1     2     3
>> b = [1; 2; 3] % column array (vector)
b =
     1
     2
     3
>> a(1,2) % access element in 1st row, 2nd column
ans =
     2
```

Now we define two different two-dimensional arrays taking note of how the comma , and semicolon ; are used to separate column and row elements, respectively. We demonstrate how the size of a two-dimensional array is defined by its number of rows and columns.

```
>> A = [1, 2; 3, 4] % define a 2x2 array
A =
     1     2
     3     4
>> A(2,2) = -10 % reset the value of element (2,2)
A =
     1     2
     3    -10
```

```
>> a = [1, 2, 3; ... % line continuation
        4, 5, 6]
a =
     1     2     3
     4     5     6
>> size(a) % will return a row vector [no. rows, no. columns]
ans =
     2     3
```

**Question:** What is the size of  $c = [1, 2, -2]$ ?

### 1.2.2 The : operator

We can create arrays and access blocks of elements efficiently using the colon : operator. The : operator creates a sequence of numbers according to the following forms

initial value : increment : final value *general format*  
 initial value : final value *default increment value is 1*

and so sets of elements of an array can be accessed with the form  $A(\text{expression})$  where the expression defining the matrix indices takes the following forms:

( initial value : increment : final value ) *increment must be an integer*  
 ( initial value : final value ) *default increment value is 1*  
 ( : ) *all elements in a specified array dimension*

Note that the initial index of any dimension is 1 and the last index can be accessed with the end keyword.

The : operator is used to define new arrays in an efficient manner:

```
>> a = 1:0.5:4
a =
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000
>> b = 1:7
b =
     1     2     3     4     5     6     7
>> C = [1:5 ; 3, 4, 1, 0, 4] % creates a 2 by 5 array
C =
     1     2     3     4     5
     3     4     1     0     4
```

and to access blocks of elements in an existing array:

```
>> C(:,2) % return the second column of array C
ans =
     2
     4
>> C(2,:) % return the second row of C
ans =
     3     4     1     0     4
>> C(:,1:2:end) % return odd columns of C
ans =
     1     3     5
     3     1     4
>> C(1,3:5) % return row 1, columns 3 thru 5 of C
ans =
     3     4     5
```

### 1.2.3 Matrix transpose

The matrix transpose operation switches the order of the array indices, effectively flipping the array over its diagonal:

$$B = A^T \text{ such that } B_{j,i} = A_{i,j}$$

The MATLAB operator used to carry out the transpose operation is the single quote ('). Consider the one-dimensional array operations:

```
>> a = [1:4]
a =
     1     2     3     4
>> b = a' % transpose operation changes a from a row to column array
b =
     1
     2
     3
     4
```

and the two-dimensional examples of the transpose operator in action:

```
>> C = [1 2 3; 4 5 6]
C =
     1     2     3
     4     5     6
>> C'
ans =
     1     4
     2     5
     3     6
```



### 1.2.4 Array sums, differences, products

The arithmetic operations  $+$ ,  $-$ , and  $*$  can be used on arrays for addition, difference, and product operations; typically, we do not use the matrix division  $/$  operation (do not confuse this with the important operator to be discussed later; the  $./$  operator also is frequently used).

1.  $A*B$  is the matrix product of  $A$  and  $B$ ; in this case,  $A$  is a  $(I \times J)$  array while  $B$  is  $(J \times K)$  and the product is  $(I \times K)$ ;
2.  $A.*B$  is the element-by-element product operation; the size of arrays  $A$  and  $B$  must be identical;
3. There is no  $./$  or  $.-$  operation because these operations are by default element-by-element; each array must be identical in size;
4. Element-by-element operations such as division  $./$  and exponentiation are frequently used; most functions (e.g.,  $\sin()$ ,  $\exp()$ ,  $\log()$ , etc) operate as element-by-element operations.

Now consider some basic MATLAB array operations:

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
>> b = [2; 3]
b =
     2
     3
>> A*b % matrix multiplication
ans =
     8
    18
>> b*A % should generate an error message
??? Error using ==> *
Inner matrix dimensions must agree.
```

and now we consider examples of element-by-element operations

```
>> C = [ 3 4; -1 -2]
C =
     3     4
    -1    -2
>> A+C
ans =
     4     6
     2     2
>> A.*C
ans =
     3     8
    -3    -8
```

Element-by-element operations are particularly useful for emulating products and quotients of true mathematical functions; for example, if  $0 \leq x \leq 100$  and we wish to plot  $y = x \sin(\pi x)$ , we first represent  $x$  by a finely discretized array of values and then compute the products of  $x$  and  $\sin(\pi x)$ :

```
>> x = [1:0.1:100]; % note how ; suppresses echo
>> y = x.*sin(pi*x); % element-by-element product is used for x sin(pi x)
>> plot(x,y)
```

### 1.2.5 Other mathematical functions

In addition to the  $+$ ,  $-$ ,  $*$ , and  $/$  functions discussed previously, MATLAB also has built in many of the familiar mathematical functions. Those that operate on scalar variables or matrices on an element-by-element basis include

function	MATLAB equivalent
$\sin(x)$	<code>sin(x)</code>
$\cos(x)$	<code>cos(x)</code>
$\tan^{-1}(y/x)$	<code>atan2(y,x)</code>
$\ln(x)$	<code>log(x)</code>
$\log_{10}(x)$	<code>log10(x)</code>
$e^x$	<code>exp(x)</code>
$\sqrt{x}$	<code>sqrt(x)</code>

Exponentiation can be a bit more tricky; if we want to compute the element-by-element square of matrix **A** we use

```
>> A.^2
```

whereas the matrix product **AA** can be computed using

```
>> A^2
```

or

```
>> A*A
```

For vectors and matrices, MATLAB also features a number of useful functions; for example, for the

two-dimensional array  $\mathbf{A}^{I \times J}$

mathematical operation	MATLAB equivalent
$\sum_{i=1}^I A_{i,j}$	<code>sum(A)</code> or <code>sum(A,1)</code>
$\sum_{j=1}^J A_{i,j}$	<code>sum(A,2)</code>
$\frac{1}{I} \sum_{i=1}^I A_{i,j}$	<code>mean(A)</code> or <code>mean(A,1)</code>
$\frac{1}{J} \sum_{j=1}^J A_{i,j}$	<code>mean(A,2)</code>

and other matrix functions that operate similarly, such as `prod(A)`, `max(A)`, and `min(A)`.

## 1.3 Plotting

MATLAB has an extensive set of tools for plotting and data visualization. Consider, for example, the problem of plotting up  $\sin(t)$  and  $\cos(t)$  for  $0 \leq t \leq 4\pi$ . First, we define an array `t` of values over the defined domain and compute both function values; the sin function is plotted as a solid blue curve

```
>> t = [0:0.2:4*pi];
>> y = sin(t);
>> z = cos(t);
>> plot(t,y)
```

The plots are displayed in Fig. 1.1. A plot can contain multiple curves, and the curves are distinguished by different colors or line types

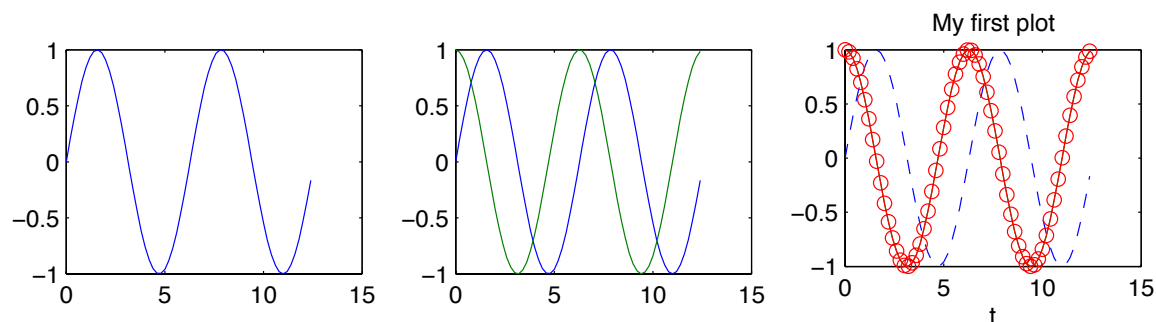


Figure 1.1: *Elementary sample plots.*

```
>> plot(t,y,t,z) % plots both curves on a single axis
>> plot(t,y,'--',t,z,t,z,'-o') % sin(t) dashed curve, cos curve -o
>> xlabel('t') % add a label to the "x" axis
>> title('My first plot') % adds a title to the plot
```

We can further modify plots using the following commands

```
>> grid % adds a grid
>> legend('sin(t)','cos(t)') % adds a legend
>> plot(t,y,'LineWidth',2) % increases the curve linewidth from 1 to 2 pts
>> axis([0 1 -1 2]) % sets x axis limits to 0 and 1, y to -1 and 2
```

### 1.3.1 Multiple plot figures

We can produce and view more than one plot at a time using the `figure` function.

```
>> figure(1) % brings figure 1 to the foreground
>> clf % clears current figure
```

### 1.3.2 Multiple plots per figure

Multiple plots can be placed on a in a single figure using the `subplot` function

```
subplot( M,N,P )
```

where  $M$  is the number of rows,  $N$  is the number of columns in the overall plot, and  $P$  is the plot number, where counting is done on a row-by row basis.

## 1.4 Scripts, functions, and files

Before we begin our discussion of programming in MATLAB, we examine the way MATLAB programs are stored and invoked using scripts and functions, and how MATLAB finds these files within your account's directory structure.

### 1.4.1 MATLAB and the file system

It is important to understand the relationship between the MATLAB operating environment and your computer's file system structure because

- MATLAB looks in the workspace for current variables and the file system for scripts and functions;
- The command shell looks first in the current directory for `.m` files, and then follows the path definitions;
- Understanding directory structures will help organize your scripts and functions into useful libraries;
- Object-oriented programming in MATLAB is implemented by defining object classes through user-created directories;
- Directory structures can be navigated through unix-like commands.

Try the following commands on your computer:

```
>> pwd      % present working directory
>> ls       % list directory contents
>> what     % list only MATLAB related files (.m, .mat, etc)

>> mkdir newdir % make a new directory (folder) named newdir
>> cd newdir  % change directories to newdir

>> diary on   % collect session output in text file "diary"
>> diary off  % turn off diary
```

We also can open a MATLAB script or function file in the M-file editor using `open filename`, where it is not necessary to include the `.m` extension in the file name.

### 1.4.2 Addpath

When MATLAB encounters a function name inside a statement, script, or function, it looks first in the present working directory for a `.m` file by that name, and then it follows the paths defined for that session. To add a new path, such as the directory `MyFuns`:

```
>> addpath MyFuns
```

### 1.4.3 Saving plots for webpages, documents, and more

Plots can be saved in a number of formats; for example, to save plots in jpeg format (suitable for Powerpoint, Word, and web-page documents)

```
>> print -djpeg myplot.jpg
```

### 1.4.4 Reading/writing data to the command window

Recall from the brief discussion of character strings, that a character string is (typically) a 1-row array of single characters (try the transpose operator on a string to see for yourself). Furthermore, scalar doubles can be converted to strings using the `num2str` function, and data can be read into the current workspace using the `input` function. To see some of these concepts in action, consider the following MATLAB session where two character strings are concatenated; note how the `disp` function removes the `ans =` output:

```
>> A='abc'; B='def';
>> C = [A,B] % concatenation at work
C =
abcdef
>> disp(C)
abcdef
```

We then concatenate character strings and numerical values by converting the latter to character strings with the `num2str` function:

```
>> F=3.2;
>> C=[A,num2str(F)]
C =
abc3.2
>> disp(['value of F: ',num2str(F)])
value of F: 3.2
```

We can use these facts to build up strings that are useful for writing out data in a more easily read format, or for figure titles and similar applications. For example, try writing to the command window using the `disp` command:

```
>> x = input('Enter the value of x: ');
Enter the value of x: 1.234
>> x
x =
1.234
>> disp(['value of x = ',num2str(x)])
value of x = 1.234
```

The `input` function can also be used to read in character strings from the command window; one can use the `input` function as-is, provided the user enter the character string in single quotes. To avoid the need for quotes, a second input parameter 's' can be added to specify the input as a character string, entered *without* quotes:

```
>> x = input('Enter you name: ','s')
Enter you name: Ray
x =
Ray
>>
```

### 1.4.5 Script files

An effective way to write programs in MATLAB that will be frequently re-used is to create a script file. A MATLAB script is a plain text file containing a sequence of MATLAB operations; when run, the commands in the script are interpreted sequentially from the top down. A key concept in script operation is that *the script operates within the workspace from which it is invoked*, and so scripts can use or modify variables already defined in that workspace, or the script can define new variables that will remain in that workspace after the script is run.

To demonstrate creating and running a script, using a text editor (such as the built-in MATLAB editor), create a file called `cpscript.m` with the following text:

```
% Cp is in J/mol/K and T is in K. The equation is valid only in the range
% 273 K < T < 373 K. Table C.3 p685, Smith, Van Ness, and Abbott, 7th ed.

Cp = 8.712 + 1.25e-3*T - 0.18e-6*T.^2;
T = 300
Cp = 8.712 + 1.25e-3*T - 0.18e-6*T^2
```

which computes the heat capacity of water  $C_p$  as a polynomial function of temperature  $T$ , where

$$C_p = 8.712 + 1.25 \times 10^{-3} T - 0.18 \times 10^{-6} T^2$$

and the coefficients are determined experimentally. In this small program, we fix the temperature to  $T = 300\text{K}$ . Try running the script by typing the filename on the command line:

```
>> cpscript
T =
    300
Cp =
    9.0708
```

Note that it is good practice to give scripts meaningful names and that the names should never contain blank spaces. Furthermore, script file names must always have the `.m` extension (suffix). We observe that the comments at the top of the script automatically become the information displayed when the `help` command is called:

```
>> help cpscript
% Cp is in J/mol/K and T is in K. The equation is valid only in the range
% 273 K < T < 373 K. Table C.3 p685, Smith, Van Ness, and Abbott, 7th ed.
```

As a practical application of this heat capacity equation

1. Plot the values of  $C_p$  over the valid range and mark the points at 300 and 373 K.
2. The amount of energy required to raise the temperature of water over the range  $[T_0, T_1]$  is given by

$$Q = \Delta H = \int_{T_0}^{T_1} C_p(T) dT$$

on a molar basis. Integrate the  $C_p$  equation by hand and add this calculation to our script to find  $Q$  given  $T_0 = 300\text{ K}$ ,  $T_1 = 373\text{ K}$ .

3. Estimate the *minimum* time required to heat a cup of water to its boiling point from  $T_0 = 300\text{ K}$  using a solar furnace. To simplify your calculations,

$$1 \text{ cup water} = 237 \text{ cm}^3 \frac{1 \text{ g}}{\text{cm}^3} \frac{1 \text{ mol}}{18 \text{ g}} = 13.17 \text{ mol}$$

## 1.4.6 Functions

Functions are similar to scripts in that they are plain text files containing a sequence of MATLAB operations and are a useful means of writing MATLAB programs that will be frequently re-used. The crucial

difference, however, between functions and scripts is that *functions define their own workspace*. This distinction is important because it allows functions to “hide” the details of their internal computations from the script, function, or MATLAB session from which the function is called.

MATLAB identifies functions from the first line of the function file, which must take the form

**function** [returned values] = funname(input parameter list)

consisting of the keyword `function`, followed by a list of variable that are to be returned from the function, followed by the function name itself `funname`, and the input parameter list - a list of variables that are sent to the function when it is called. MATLAB is quite flexible in calling functions in that not all input and returned parameters must be specified when the function is called. Generally, the function filename should be the same as the function name `funname`.

As an example of creating and using a function, if we would like to place the script we just wrote into a library and to increase its utility by allowing the script to take in any value of temperature  $T$  and return the corresponding heat capacity value  $C_p$ , we rewrite the function in the following form (named `cpair.m`):

```
function Cp = CpWater( T )
% Cp is in J/mol/K and T is in K. The equation is valid only in the range
% 273 K < T < 373 K. Table C.3 p685, Smith, Van Ness, and Abbott, 7th ed.
%
% Called as C = CpWater(T)

Cp = 8.712 + 1.25e-3*T - 0.18e-6*T.^2;
return
```

Note that the  $T^2$  term in this function allows for input temperature values in the form of vectors and arrays. Now, we define a temperature value  $T_s$  and call the function with the following commands:

```
>> Ts = 350;
>> D = CpWater(Ts)
D =
    9.1274
```

It is important to observe that *the variable names used to call the function can be different from those used internally by the function*. The way this works is that

1. variable  $T_s$  is defined in the workspace of the current MATLAB session;
2. a copy of  $T_s$  is passed to function `cpair.m` when the function is called and is copied to variable  $T$  inside the function;
3. the computations proceed inside the function, computing the value  $C_p$ , a copy of which is returned from the function by copying its value to variable  $D$  in the workspace from which the function was called.

Therefore, because of the way copies of variables are passes between the workspaces, the values of  $T$  and  $C_p$  remain inside the function, and so they disappear after the function is called. This behavior of functions is crucial to the operation of most computer programs, which would otherwise behave unpredictably if



the internal workings of a function disrupted variables in the workspace from which the function was called.

We see that our function can now be used to conveniently plot the heat capacity function by passing a temperature array to the function:

```
>> Tv = (-10:10:110) + 273;
>> Dv = CpWater(Tv);
>> plot(Tv,Dv,'-o')
>> grid on, xlabel('T (K)'), ylabel('Cp (J/mol/K)')
```

Results can be seen in Fig. 1.2.

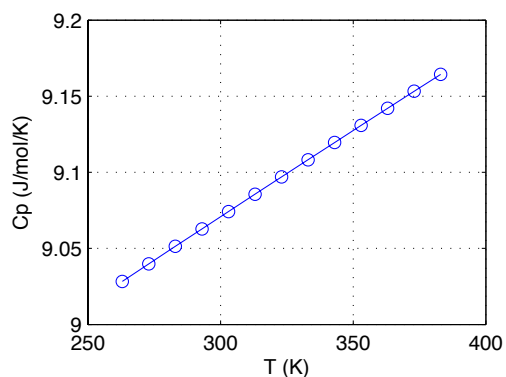


Figure 1.2: *Heat capacity of water as a function of temperature.*

Finally we note that the `return` statement included in the function is not actually needed in this case; however, when a `return` is reached inside a function, execution of the function stops immediately and control is returned to the statement, script, or function calling the function; this is useful for cases when we want to return from the function from a specific point within the function.

### 1.4.7 Varying the number of input/output parameters

As was stated in the previous section, not all input and output arguments of a MATLAB function must be specified when the function is called. For example, consider a function that begins with the statement

```
function [A,B,C] = myfun(X,Y,Z)
% remainder of function goes here
```

where A, B, and C are computed inside the function and returned. If we are only interested in the first output variable value, we can call the function using the following statement:

```
a = myfun(x,y,z)
```

Because of this flexibility in calling functions, we see there is value in ordering the output parameter list so that the most frequently needed output parameters are listed first. Inside the function, we can determine the number of output arguments used to call the function with the `nargout` keyword.

Because not all input parameters must be specified when calling a function, we must build in this flexibility when creating the function. Consider for example, if we would like to be able to call the function above by specifying only  $X$  and  $Y$ , where the default value of  $Z=0$  is used in cases when it is not specified. We rewrite the initial portion of the function to define this default value:

```
function [A,B,C] = myfun(X,Y,Z)
if nargin <= 2
    Z = 0;
end
% remainder of function goes here
```

Additionally,

```
function [A,B,C] = myfun(varargin)
```

allows a variable number of input arguments by having the input list be specified as the cell array `varargin`.

### 1.4.8 Inline and anonymous functions

Before closing this section on functions, let us describe two convenient approaches to defining relatively simple functions. An *inline* function is created using a character string `expr` corresponding to the function definition. The function name is defined when the function is created and arguments `arg1`, `arg2`, ... can be passed to the function. For example to create  $g(x) = A \sin(x)$

```
>> g=inline('A*sin(x)','x','A')
g =
    Inline function:
    g(x,A) = A*sin(x)
>> g(pi/2,10)
ans =
    10
```

We note that  $A$  and  $x$  must be explicitly passed to the function. An alternative that offers more flexibility is the *anonymous* function. Anonymous functions

1. are simple to pass to other functions, e.g., plotting, minimization, and numerical integration routines;
2. are created using the `@` symbol and are constructed without the single quotation marks used for inline functions;
3. can include variables currently defined in the workspace (c.f. inline functions); note that subsequent modifications to the variables do not affect anonymous functions already defined—parameters of the function are “frozen” at the time of their creation;
4. and are recommended for use over inline functions.

To see anonymous functions in action, try the following script

```
A = 10; % define A in workspace
g = @(x) A*sin(x) % note space location
ezplot(g)
```

## 1.5 Logical operations

### 1.5.1 Relational operators

Logical variables take on one of two values:

```
0 : false
1 : true
```

These values are produced by logical statements created using the relational operators listed below:

```
< : less than
> : greater than
<= : less than or equal to
>= : greater than or equal to
== : equal to
~= : not equal to
```

It is important to distinguish between the assignment (=) and equivalence operators; for example

```
>> A = 14 % assign the value of 14 to variable A
A =
    14
>> L = A == -10 % check if A equals -10
L =
     0
```

These relational operators can operate on scalars, arrays, and even character arrays (strings); for example, for double arrays

```
>> A = [1 2; 3 4]; B = [1 2; -10 7];
>> A == B
ans =
     1     1
     0     0
```

and for character strings

```

>> a = 'a'; b = 'b';
>> a == b
ans =
    0
>> a == 'a'
ans =
    1
>> strcmp(a,'a') % a better way to compare strings
ans =
    1

```

### 1.5.2 Precedence

Equations in MATLAB follow the typical precedence of mathematical operations:

1. operations in parentheses ( ) are evaluated first (this applies to functions)
2. exponentiation and matrix transpose are performed next
3. then unitary minus (-) and logical negation ~
4. \* and / are next
5. \ is next
6. + and - then are performed
7. then the : operator
8. the logical operators <, >, <=, >=, ==, ~=
9. logical AND (&)
10. logical OR (|)
11. short-circuit AND (&&)
12. short-circuit OR (||)
13. assignment = is performed last.

When two operations with equal precedence are encountered, MATLAB evaluates them from left to right.

Using this hierarchy, it is easy to see that in the following statement taken from `cpscript.m`, the exponents are evaluated first, followed by the product operations, then the addition operations, and finally the assignment operation:

```
Cp = 28.09 + 0.1965e-2*T + 0.4799e-5*T^2 - 1.965e-9*T^3
```

The rules of precedence also are critical to understanding important MATLAB operations that may, at first, seem to make no sense at all. Consider, for example, the following MATLAB statements:

```

n = 0
n = n + 1

```

In this sequence of statements, `n` is first assigned the value of 0; in the second statement, according to the rules of precedence, the addition operation on the right-side of the assignment operator `=` is performed first giving a numerical value of 1, which is then assigned to variable `n`, leaving it with the new value. This type of statement is extremely useful for counting operations inside computer programs.

### 1.5.3 Branching structures

Logical decision structures are program elements in which a sequence of decisions are made, producing some result dependent on the outcome of the decisions. The essential feature of such decision processes are the `if` structure, an example of which is included below.

```
if logical expression is true
    statements to be executed
end
```

and for multiple decisions

```
if logical expression
    statements to be executed
elseif logical expression
    statements to be executed
elseif logical expression
    statements to be executed
end
```

When one or more of the logical expressions are true, the first encountered is executed; for example, consider the script

```
a = -10;
if a < 0
    disp('RESULT: a<0')
elseif a < -5
    disp('RESULT: a<-5')
else
    disp('RESULT: a>=0')
end
```

Note the use of whitespace indentation to improve the readability of the script and to give a clearer picture of the logical structure of the statements. Running the script yields

```
RESULT: a<0
```

The potential ambiguity of the script above may be reduced using **nested** `if` statements

```

a = -10;
if a < 0
    disp('RESULT: a<0')
    if a < -5
        disp('RESULT: a<-5')
    end
else
    disp('RESULT: a>=0')
end

```

which results in

```

RESULT: a<0
RESULT: a<-5

```

### 1.5.4 Multiple conditions per if statement

It is possible to construct branching structures that include more than one condition per decision using the AND (&) and OR (|) operators:

```

if logical expression 1 & logical expression 2
    statements to be executed if both conditions are true
end

```

```

if logical expression 1 | logical expression 2
    statements to be executed if either or both conditions are true
end

```

The && and || are logical AND and OR operators that feature *short-circuiting* behavior, preventing the evaluation of subsequent logical operations when an expression evaluates to FALSE. A useful example of a short-circuit expression is

```

if b ~= 0 && a/b >= 1

```

which prevents division by zero in  $a/b$  when  $b=0$ .

### 1.5.5 A closer look at working with arrays

We saw earlier that the statement  $A == B$  produced an array of logical variables the same size as  $A$  and  $B$  provided the arrays  $A$ ,  $B$  are the same size. The relational operators also work when  $A$  or  $B$  is a scalar, returning an array the same size as the non-scalar. To summarize in a set of examples,

```

1 == [1, 2, 3]   evaluates to [1, 0, 0]
[1, 2] == [1, 2, 3] results in an array size error
[1, 3, 2] == [1, 2, 3] evaluates to [1, 0, 0]

```

We observe the general result of these operations is an array; using such an operation in an `if` statement, the expression is evaluated as TRUE if all the elements of the array produced by the relational operation

are true. If we would like to know whether *any* element of `[1,2,3]` is equal to 1, and to generate a scalar logical value describing the outcome of the evaluation, we use the MATLAB `any` function:

```
if any([1,2,3] == 1)
    disp('found a match')
end
```

Functions such as `any` are useful for comparing sets of values; for additional set operations, we may choose to use the MATLAB `intersect` and `union` functions.

### 1.5.6 Switch

One can use the `switch` statement as an alternative to the `if` structures discussed above. The general format is

```
switch scalar or string expression
    case case expression 1
        statements to be executed
    case case expression 2
        statements to be executed
    otherwise
        statements to be executed
end
```

### 1.5.7 Find

A function that finds the indices of all nonzero elements; this function is especially useful when used in conjunction with a logical operation; for example

```
>> x = [0:0.1:20];
>> y = sin(x);
>> I = find( y>0 ); % returns indices of all positive portions of the curve
>> plot(x,y,x(I),y(I),'o') % plots the positive segments
```

Results are plotted in Fig. 1.3.

## 1.6 Looping constructs

Loops for iterative and other calculations are created in a number of ways. The simplest form of a looping construct is:

```
for i = array of values
    statements to be executed
end
```

As an example, try the following script

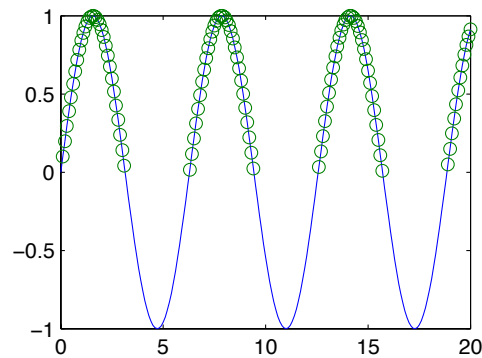


Figure 1.3: Use of the `find` function to identify the positive segments of  $\sin(x)$

```
% My first loop
for i = 1:6
    i
end
```

Now try running the script; note that the following is equivalent

```
% My first.1 loop
X = 1:6;
for i = X
    i
end
```

We can use a loop to create a table containing the cumulative sum of all integers from 1 to 10; the results are displayed in two columns using the MATLAB `disp.m` method.

```
% Table of cumulative sums
cumusum = 0;
for i = 1:10
    cumusum = cumusum + i;
    disp( [i,cumusum] )
end
```

Now try running the script; again, note the operation of the statement `cumusum = cumusum + i` and how its function depends on the precedence of the `+` and `=` operators.

### 1.6.1 Nested loops

Loops-within-loops frequently are used in matrix mathematical operations. For example, we can access and print the value of each element in an array with the following set of nested loops; we first set up a 2x3 array

```
>> A = [4 2 -3; 3 6 -10]
A =
     4     2    -3
     3     6   -10
```



and then extract the elements of the left-most 2x2 block of elements:

```
for row = 1:2
    for column = 1:2
        A(row,column)
    end
end
```

producing the output

```
ans =
     4
ans =
     2
ans =
     3
ans =
     6
```

For nicer output, try replacing `A(row,column)` with

```
disp(['A(',int2str(row),',',int2str(column),') = ',int2str(A(row,column))])
```

## 1.6.2 While loops

We now consider an alternate loop construct:

```
while logical expression
    statements to be executed
end
```

As an example, try the following script

```
% My second loop
i = 0;
while i < 6
    i = i+1
end
```

Note that an important difference between `if` and `while` loops is that a faulty choice of relational operation in the latter can result in an infinite loop.

## 1.7 Elements of programming in MATLAB

Now that we understand the basics of mathematical operations, logical operations, `if` and loop structures, and how to package MATLAB operations into scripts and functions, let us consider the problem of transforming a mathematical procedure or algorithm into a MATLAB program.

### 1.7.1 Combining decisions and mathematical operations

To illustrate the basic elements of programming, consider a case where we would like to modify the heat capacity function so that it checks to make sure the temperature  $T$  is in the proper range. In everyday language, an algorithm that would carry out this procedure might first check if the temperature is below the minimum valid temperature, then check if it too high, and if both of those conditions are not violated, compute the heat capacity using the empirical correlation. We carry out these operations in the following function using the `if` structure; note how multiple commands can be placed on one line by placing a comma between statements:

```
function Cp = CpWaterScalar(T)
% Cp is in J/mol/K and T is in K. The equation is valid only in the range
% 273 K < T < 373 K. Table C.3 p685, Smith, Van Ness, and Abbott, 7th ed.
%
% Called as C = CpWaterScalar(T) where T is a scalar

if T < 273
    disp('Temperature too low')
    Cp = NaN;
elseif T > 373
    disp('Temperature too high')
    Cp = NaN;
else
    Cp = 8.712 + 1.25e-3*T - 0.18e-6*T^2;
end
```

We test the function with the following script:

```
for T = (-10:20:110) + 273
    D = CpWaterScalar(T);
    disp(['T: ',num2str(T),',', Cp: ',num2str(D)])
end
```

which produces

```
Temperature too low
T: 263, Cp: NaN
T: 283, Cp: 9.0513
T: 303, Cp: 9.0742
T: 323, Cp: 9.097
T: 343, Cp: 9.1196
T: 363, Cp: 9.142
Temperature too high
T: 383, Cp: NaN
```

A more effective function makes use of the MATLAB `find` (or the `any`) function:

```

function Cp = CpWaterFinal(T)
% Cp is in J/mol/K and T is in K. The equation is valid only in the range
% 273 K < T < 373 K. Table C.3 p685, Smith, Van Ness, and Abbott, 7th ed.
%
% Called as C = CpWaterScalar(T) where T can be an array

Cp = 8.712 + 1.25e-3*T - 0.18e-6*T.^2;
I = find(T < 273); % temperature too low
J = find(T > 373); % too high

if length(I) > 0
    disp('Low temperatures found')
    Cp(I) = NaN;
end
if length(J) > 0
    disp('High temperatures found')
    Cp(J) = NaN;
end

```

Try this function with the input

```
T = (-10:20:110) + 273;
```

## 1.7.2 Linear interpolation

Given two data points  $(x_{d1}, y_{d1})$  and  $(x_{d2}, y_{d2})$ , we can define a line

$$y = \frac{y_{d2} - y_{d1}}{x_{d2} - x_{d1}} (x - x_{d1}) + y_{d1}$$

We translate this to the MATLAB function below:

```

function y = linearinterp(xdata,ydata,x)

slope = (ydata(2)-ydata(1))/(xdata(2)-xdata(1));
y = slope*(x-xdata(1)) + ydata(1);

```

If we try this with

```

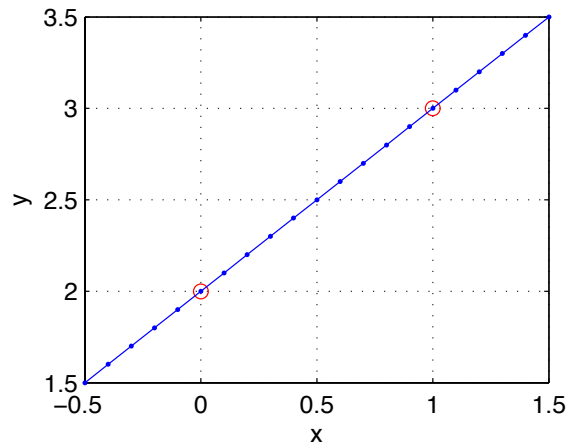
>> xd=[0 1]; yd=[2,3]; x=[-0.5:0.1:1.5];
>> y = linearinterp(xd,yd,x);
>> plot(xd,yd,'ro',x,y,'.-')
>> grid on; xlabel('x'); ylabel('y')

```

we obtain the plot in Fig. 1.4, showing the two data points as red circles, and the interpolated points as the blue dots connected by a blue line.

## 1.7.3 Search for a minimum value

In this algorithm, we make use of logical operations and loop structures to develop a function that scans a column vector of data, searching for and returning the minimum value and corresponding index.

Figure 1.4: *Demonstration of linear interpolation.*

```
function [minval, I] = mymin(X)
% Search a column vector for its minimum value
I = 1;
minval = X(I);
for i = 2:length(X)
    if X(i) <= minval
        minval = X(i);
        I = i;
    end
end
```

We test this with

```
>> X = [1,-5,3,10];
>> [minval, I] = mymin(X)
```

### 1.7.4 The quadratic equation solver

We make use of branching concepts in developing a script for solving the following quadratic equation

$$ax^2 + bx + c = 0$$

Of course, the quadratic equation has two solutions:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We present the following script as a program that takes as inputs the values of the three coefficients, checks the discriminant for the form of the roots (two real and distinct, two real identical, two complex roots), and prints the results:

```
% Quadratic equation solver script
a = input('Enter a: '); b = input('Enter b: '); c = input('Enter c: ');
disp(['The equation: ', ...
      num2str(a),'x^2 + ',num2str(b),'x + ',num2str(c),' = 0'])

discr = b^2 - 4*a*c;
if discr > 0
    x(1) = (-b + sqrt(discr))/(2*a);
    x(2) = (-b - sqrt(discr))/(2*a);
    disp(['has two real distinct roots: ', ...
          num2str(x(1)),' , ',num2str(x(2))])
elseif discr == 0
    x = -b/(2*a);
    disp(['has two identical real roots: ',num2str(x)])
else
    xreal = -b/(2*a); ximag = sqrt(-discr)/(2*a);
    disp(['has complex roots: ',num2str(xreal),' +/- i',num2str(ximag)])
end
```

We test this script with two cases, the first which generates two real distinct roots and the second complex roots:

```
>> quadscript
Enter a: 1
Enter b: 5
Enter c: 1
The equation: 1x^2 + 5x + 1 = 0
has two real distinct roots: -0.20871 , -4.7913
>> quadscript
Enter a: 5
Enter b: 1
Enter c: 1
The equation: 5x^2 + 1x + 1 = 0
has complex roots: -0.1 +/- i0.43589
```

### 1.7.5 Quadratic equation solver function

To increase the utility of the quadratic equation solver, we can rewrite it in the form of a function:

```

function [xreal, ximag] = quadfun(a,b,c)
% Quadratic equation solver function
disp(['The equation: ', ...
      num2str(a),'x^2 + ',num2str(b),'x + ',num2str(c),' = 0'])

ximag(1) = 0; ximag(2) = 0;
discr = b^2 - 4*a*c;
if discr > 0
    xreal(1) = (-b + sqrt(discr))/(2*a);
    xreal(2) = (-b - sqrt(discr))/(2*a);
    disp(['has two real distinct roots: ', ...
          num2str(xreal(1)),' , ',num2str(xreal(2))])
elseif discr == 0
    xreal(1) = -b/(2*a); xreal(2) = -xreal(1);
    disp(['has two identical real roots: ',num2str(xreal(1))])
else
    xreal(1) = -b/(2*a); xreal(2) = xreal(1);
    ximag(1) = sqrt(-discr)/(2*a); ximag(2) = -ximag(1);
    disp(['has complex roots: ', ...
          num2str(xreal(1)),' +/- i',num2str(ximag(1))])
end

```

Test the function above with the following:

```

>> x = quadfun(1,5,1)
>> [x,y] = quadfun(5,1,1)
>> [x,y] = quadfun(1,5,1)

```

### 1.7.6 Size selection

We now make use of logical operations and loop structures to develop a function than sorts a column vector into discrete size ranges (bins). We leave it as an exercise to modify this function so that it works for cases where one or more data points coincides with the upper value of the variable binrange.

```

function [ninbin, binmin, binmax] = binsort(X, N, binrange)

% Sort a column vector of data points into N evenly spaced bins

binmin = ( 0:1/N:(N-1)/N )*(binrange(2)-binrange(1)) + binrange(1);
binmax = binmin + (1/N)*(binrange(2)-binrange(1));
ninbin = zeros(N,1);
for i = 1:length(X)
    for n = 1:N
        if X(i) >= binmin(n) & X(i) < binmax(n)
            ninbin(n) = ninbin(n) + 1;
        end
    end
end
end

```

The MATLAB function `randn` generates normally distributed random numbers with zero mean and standard deviation of 1; in the example below, we generate 10,000 of these random numbers and sort them into 10 bins between -3 and 3, then plot the results with the MATLAB bar chart function to reveal the classic bell curve in Fig. 1.5

```
>> y = randn(10000,1);
>> [ninbin, binmin, binmax] = binsort(y, 10, [-3,3])
>> bar((binmax+binmin)/2,ninbin)
```

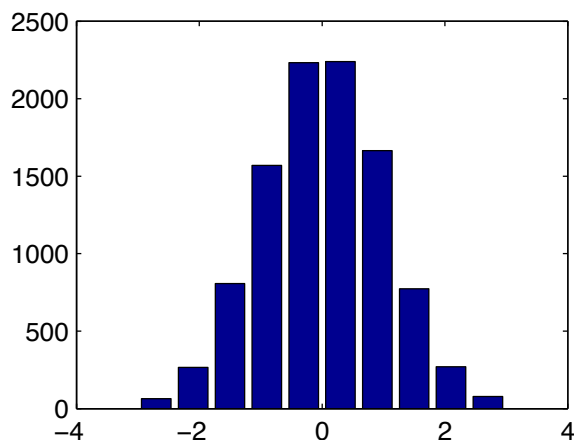


Figure 1.5: *Demonstration of the bin sorting algorithm using a normally distributed random number generator.*

## 1.8 Review problems

1. Numerically demonstrate that MATLAB's built-in `log` function corresponds to the natural logarithm, first by computing `log(2.7183)` and then by using MATLAB's `help` function. Record your results using the `diary` function, saving the `diary.m` file to your computer's desktop.
2. Briefly describe the differences between
  - (a) procedural and object-oriented programming
3. One day this week, look up the high and low temperatures recorded at 5 major cities in the United States. Arrange these data in a MATLAB struct data type and compute the difference between the high and low temperatures, the averages of temperatures and differences, and the standard deviation of the same. Then, construct a plot which graphically illustrates as much of this information as possible.
4. A point on a circular wheel traces a curve called the cycloid as the wheel rolls in a straight line. The cycloid is defined by the parametric equation

$$\begin{aligned}x &= r(t - \sin t) \\ y &= r(1 - \cos t)\end{aligned}$$

Plot the cycloid for  $r = 1.5$  and  $0 \leq t \leq 7\pi$  by defining a finely spaced vector of points  $\mathbf{t}$  and using the vector to compute and plot  $\mathbf{x}$  and  $\mathbf{y}$ .

5. Create the following arrays

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

by first creating the array

$$\mathbf{C} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and then forming  $\mathbf{A}$  and  $\mathbf{B}$  from an array of  $\mathbf{C}$  arrays in a tiling pattern.

6. Consider the arrays

$$\begin{aligned} \mathbf{a} &= [1 \ 2] \\ \mathbf{b} &= [2; \ 3; \ 3] \\ \mathbf{c} &= [-1 \ 2; \ -1 \ 0] \\ \mathbf{d} &= [1 \ 2 \ 3; \ -2 \ 0 \ 1] \end{aligned}$$

Calculate by hand the products (if any) of the following operations; compare your results to MATLAB

$$\mathbf{a}*\mathbf{b}, \mathbf{b}*\mathbf{a}, \mathbf{d}*\mathbf{b}, \mathbf{c}*\mathbf{a}'$$

7. Generate each of the following sequences using the `:` operator

$$\begin{aligned} &0, 1, 2, 3, 4, 5, 6 \\ &-1.5, -1.0, -0.5, 0, 0.5, 1.0, 1.5, 2.0 \\ &-10, -8, -6, -4, -2, 0 \\ &3, 5, 7, 9, 11, 13, 15, 17 \end{aligned}$$

8. Using the `:` operator, write a two-line MATLAB script to first extract the odd rows from a matrix  $\mathbf{A}$ ; assigning that result to matrix  $\mathbf{B}$ , then extract the even columns from  $\mathbf{B}$  to form  $\mathbf{C}$ ; test your example with `A=round(10*rand(5))`.
9. The two-column array **february** has days listed in the first column and the daily average temperature (in °C) in the second. Using a **for**-loop, check each temperature and write out those dates corresponding to below-zero temperatures. Test your algorithm with `february = [1 -2; 2 3; 3 0; 5 -5; 6 10]`.
10. The factorial of  $n$  is defined as

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdots n \text{ for } n > 0 \\ &= 1 \text{ for } n = 0 \end{aligned}$$

for non-negative integers  $n$  (the Gamma function can be used for non-integer and negative values, but we do not consider those cases in this problem). Using this explicit definition, write a MATLAB function that returns the value of  $n!$  for input parameter  $n$ . Test the function for input values of 3, 0, 3.5, and -3.



11. Create a MATLAB function that computes the molecular masses of molecules composed of hydrogen, nitrogen, oxygen, and carbon; the function should have as its input parameters the number of each of these species and should return the molecular mass of the input molecule. Test your function on water, propane, benzene, nitric acid, and dimethyl ether.
12. Heat capacity for many materials can be approximated by the polynomial curve fit

$$C_p(T) = a + bT + cT^2 + dT^3$$

where  $T$  is given in  $^{\circ}\text{C}$  and  $C_p$  in  $\text{J/mol/K}$ . Data for several chemical compounds are given below; these curve fits are valid over the temperature range  $0\text{--}1200^{\circ}\text{C}$

Substance	$a$	$b$	$c$	$d$
acetone	71.96	$20.10 \times 10^{-2}$	$-12.78 \times 10^{-5}$	$34.76 \times 10^{-9}$
air	28.94	$0.4147 \times 10^{-2}$	$0.3191 \times 10^{-5}$	$-1.965 \times 10^{-9}$
methane	34.31	$5.496 \times 10^{-2}$	$0.3661 \times 10^{-5}$	$-11.00 \times 10^{-9}$

- (a) Write a MATLAB function that takes in the values of these coefficients and a given temperature  $T_o$  and computes the molar heat capacity.
- (b) Then, after integrating the general formula for heat capacity by hand, modify the function to compute the total amount of energy required to increase the temperature of one mole of each substance by  $500^{\circ}\text{C}$  over the initial temperature  $T_o$ .
- (c) Plot the values of  $C_p$  for all three chemical species on the sample plot over the temperature range each correlation is valid. Use a legend and different line types on this plot to distinguish between the species.
13. Create three functions called `cpacetone.m`, `cpair.m`, and `cpmethane.m` that take as input a single value of temperature and return the value of  $C_p$ ; demonstrate the use all three for  $T = 200^{\circ}\text{C}$  using a single MATLAB script.

Modify the functions above so that they take arrays as inputs; check each by plotting  $C_p$  for each component for

$$t = [100:5:600] .$$

||

14. A 1 mol/sec stoichiometric mixture of air and methane available at  $20^{\circ}\text{C}$  is to be fed to a furnace at  $500^{\circ}\text{C}$ ; using your functions for computing heat capacity of these species (developed in the previous problem), compute the power required using a constant heat capacity evaluated at  $250^{\circ}\text{C}$ .
15. What are the results of

$$\begin{aligned} I &= 2 == 2 > 1 \\ &= 2 : 5 + 2 \\ &= 2 * 2 : (5 - 1) \\ &= 3 > 2 \& 1 \\ &= 3 < 2 \& \& 1 / (3 < 2) \end{aligned}$$

16. What is the result of

```
J = [1, 3];
B = [2, 4; -6, 9];
y = B(J)
```

17. Look up the special matrices ones, zeros, eye and briefly describe the arrays they produce.
18. What are the elements of the following arrays:

```
A = 1:3
B = [A; -2:1:0]
C = [B; -A]
```

Which arrays can be multiplied?

19. A 4 m<sup>2</sup> roof-top solar water heater theoretically can collect up to 4000 W of power under optimal conditions. Using the heat capacity functions for water described in the text, compute the efficiency of a system that is found to heat 5 mol/s of water from 25° C to 55° C.
20. A map of our geographical region denotes  $x > 0$  as the number of miles east of the origin and  $y > 0$  north, with the map centered at Cumberland, MD. In this coordinate system, the depth  $d$  of the Marcellus geological formation is approximated as

$$d(x, y) = 40x - 45y - 0.25xy + 9000 \quad \text{feet}$$

Plot the depth of the formation for  $-120 \leq x \leq 20$  and  $-20 \leq y \leq 140$ .

21. Modify the quadratic equation solving function so that
  - (a) the single root  $x = -c/b$  is computed and returned for the case  $a = 0$  and  $b$  nonzero;
  - (b) reports no solution exists if  $a$  and  $b$  both are zero.
22. Modify the bin-sorting algorithm binsort.m discussed in the notes, to include the following features:
  - (a) take into account the potential of data points at the uppermost bin range limit;
  - (b) check if the binrange is valid;
  - (c) create a related function which first calls binsort.m and then plots the output as a step like plot with ninbin as the ordinate, and the binmin and max values as the abscissa;

Test your algorithm and plotting routine for the following data sets

- (a) 1000 random numbers generated with the MATLAB rand.m function with  $N = 10$  and binrange = [0 1]
  - (b) same as above, but with binrange = [0 1.5]
  - (c)  $X = [0 \ 1 \ 2 \ 3 \ 2 \ 3 \ 2 \ 1 \ 4 \ 8 \ 6 \ 5 \ 3 \ 4 \ 5 \ 6 \ 3 \ 9 \ 2 \ 1 \ 4 \ 5 \ 6 \ 2 \ 3 \ 4 \ 5]$  with  $N = 3$  and binrange = [0 9]
23. Create a function that sorts three inputs in ascending order using an algorithm that exhaustively searches through all combinations of the three inputs for the correct ordering. Test your function with random sets of 3 numbers, including some combinations where two of the inputs are identical.

24. Using the function `mymin.m`, create a function that sorts a vector in ascending order, called as

`[V, I] = mysort(X)`

producing the sorted vector  $V$  and a vector  $I$  of the same size corresponding to the index of those values in the original data. Test your algorithm with the  $X$  data of the previous problem. **Hint:** create a vector containing the indices of the initial data and iteratively determine the current minimum value in the data vector using `mymin.m`, removing both the identified value and the corresponding index element during each iteration.

25. Look up the definition of the two sorting algorithms: *bubble sort* and *insertion sort*. Write individual MATLAB functions that implement the two algorithms. The input to each function must be a row or column vector, and the two returned variables are the sorted vector (in ascending order) and the vector of indices for the sorted array indicating the original positions of the elements in the sorted array with respect to the input array. Test your functions with row array  $a = [2 \ 4 \ -1 \ -1 \ 0 \ 11 \ -3 \ 2 \ 8 \ 7 \ 6]$ , printing the results of each iteration as the sorting algorithms proceed. Which algorithm requires fewer iterations?

26. Consider the following vapor-liquid equilibrium data

$T(^{\circ}C)$	138	75	120	90	69
$x_h$	0	0.79	0.11	0.40	1
$y_h$	0	0.97	0.50	0.84	1

Create a function which takes as input a temperature between 69 and 138°C, performs a linear interpolation to find the corresponding values of  $x_h$  and  $y_h$ , and plots the tie-line connecting the two, along with all of the thermodynamic data (temperature on the ordinate, mole fractions on the abscissa). Test your function for  $T = 60:10:140^{\circ}C$ . Be sure to sort all thermodynamic data (using one of the algorithms discussed above) before doing any other calculations.

## Chapter 2

# Object-oriented programming in MATLAB

Up to now we have focused on procedural programming techniques: translating numerical algorithms into source code in a very "top-down" approach. Now we shift our focus to the data being processed, defining new data types and the functions (methods) that operate on them. Why? Because this approach results in more reliable and reusable software, especially for large projects.

### 2.1 Object-oriented programming concepts

Object-oriented programming concepts can significantly reduce the complexity of computationally implementing applied mathematics and numerical analysis methods. The object-oriented programming in the context of scientific computations proceeds by identifying those data structures that remain unchanged during a solution procedure and creating a corresponding set of methods that operate on these new objects.

Our approach to achieving these goals is to define a set of new MATLAB object classes. New object classes in MATLAB are defined by extensions of an existing MATLAB class; the new and overloaded methods of the class are placed in a separate directory dedicated to that class.

In MATLAB, we have encountered such data types as

- double (the default data type)
- char

We now examine the struct data type which allows storage of set of variables of possibly different type in one variable:

```
A.val = 1.3  
A.units = 'm'
```

||

**Definition:** A class defines the data fields of particular structures (objects) of that class and the methods (functions) that can be used on objects of that class

### 2.1.1 The dimensional class

Example: We consider developing the dimensional class, with two data fields:

```
numerical value : A.val
units : A.units
```

and the following methods

```
constructor method (dimensional.m)
display method (display.m)
addition + (plus.m)
multiplication * (mtimes.m)
```

Because `plus.m` and `mtimes.m` are already defined in MATLAB as functions that normally operate on variables of type `double`, we will be overloading these methods by our new definitions.

We define our new class in MATLAB by creating a new directory named `@dimensional`. Inside this directory (folder), we place the constructor function `dimensional.m`

```
function B = dimensional(val,units)
% dimensional class constructor method
A.val = val;
A.units = units;
B = class(A,'dimensional');
```

a display method `display.m`

```
function display(A)
% dimensional class display method
disp([num2str(A.val), ' ', A.units])
```

the overloaded addition function `plus.m`

```
function C = plus(A,B)
% Dimensional class plus (+) method
if strcmp(A.units,B.units)
    val = A.val+B.val;
    units = A.units;
    C = dimensional(val,units);
    return
end
error('Units mismatch')
```

and the overloaded matrix product method `mtimes.m`

```

function C = mtimes(A,B)
% dimensional class mtimes (*) method
val = A.val*B.val;
if strcmp(A.units,B.units)
    units = [A.units,'^2'];
else
    units = [A.units,' ',B.units];
end
C = dimensional(val,units);

```

Consider then, defining

```

x = 1.3 m
y = -2.0 m
z = 5.0 sec

```

using the calls to the constructor method `dimensional.m`:

```

>> x = dimensional(1.3,'m')
1.3 m
>> y = dimensional(-2,'m')
-2 m
>> z = dimensional(5,'sec')
5 sec

```

and noting the format of the echoed output, defined by the `display.m` method. We then try out representative calculations with our overloaded methods:

```

>> x + y
-0.7 m
>> x * y
-2.6 m^2
>> x * z
6.5 m sec
>> x + z
??? Error using ==> dimensional/plus
Units mismatch
>> x==y
??? Error using ==> ==
Function '==' not defined for variables of class 'dimensional'.

```

Note that MATLAB returns an error as a result of the final command because the definition of equivalence (`eq.m`) has not yet been defined for objects of the `dimensional` class.

### 2.1.2 The myline class

We consider developing the `myline` class, with two data fields:

```

slope : A.a
intercept : A.b

```

so that we can conveniently work with scalar linear algebraic equations of the form

$$y = ax + b$$

Therefore, objects of class `myline` will contain the constant coefficients of the linear equation; the methods we will develop to create these objects and to manipulate the objects consist of:

1. constructor method (`myline.m`)
2. display method (`display.m`)
3. addition `+` (`plus.m`)
4. a method to evaluate  $y = ax + b$  (`yval.m`)
5. a simple plotting method (`plot.m`)
6. accessor methods (`slope.m`, `intercept.m`)

Because `plus.m` and `plot.m` are already defined in MATLAB as functions that normally operate on variables of type `double`, we will be overloading these methods by our new definitions.

We define our new class in MATLAB by creating a new directory named `@myline`. Inside this directory (folder), we place the constructor function `myline.m`

```
function A = myline(x,y)
% myline.m Constructor method for myline class. Called as
%
%           A = myline(x,y)
%
% INPUT PARAMETERS
%           x : x values or a (slope)
%           y : y values or b (intercept)
if nargin == 0; x = 0; y = 0; end % default values
if prod(size(x)) == 1
    A.a = x; A.b = y;
else
    B(:,1) = x(:); B(:,2) = 1;
    % Note how the line is computed by least squares, equivalent
    % to coeff = (B'*B) \ (B'*y)
    coeff = B\y(:);
    A.a = coeff(1); A.b = coeff(2);
end
A = class(A,'myline');
```

We note that the line may be computed by a least squares procedure when more than two data points are used as input to define the line; more details on the least-squares procedure was provided in the previous Chapter. A display method now is defined:

```
function display(A)
% display.m Display method for myline class
disp('myline class object, with y = ax+b')
disp(['a = ',num2str(A.a)])
disp(['b = ',num2str(A.b)])
```

the overloaded addition function `plus.m`

```
function A = plus(A,c)
% plus.m Addition method for myline class. Called as
%
%      B = A+c
%
% INPUT PARAMETERS
%      A : a myline object
%      c : a scalar value
A.b = A.b + c;
```

and the overloaded plotting method `plot.m`

```
function plot(A,x0,x1,linesty)
% plot.m Plot method for myline class. Called as
%
%      plot(A,x0,x1,linesty)
%
% INPUT PARAMETERS
%      A : a myline object
%      x0,x1 : domain over which the line will be plotted
%      linesty : linestyle (optional)
x = [x0,x1];
if nargin == 3
    plot(x,yval(A,x))
elseif nargin == 4
    plot(x,yval(A,x),linesty)
end
```

the linear equation evaluation method `yval.m`

```
function y = yval(A,x)
% yval.m Evaluate y = ax+b method for myline class. Called as
%
%      y = yval(A,x)
%
% INPUT PARAMETERS
%      A : myline object
%      x : x values
y = A.a*x + A.b;
```

and finally, the accessor methods `slope.m` and `intercept.m`, needed because *encapsulation* prevents any methods outside the class `myline` from having access to the data fields of a `myline` object:



```
function a = slope(A)
% slope.m An accessor method for myline class. Called as
%
%           a = slope(A)
%
% INPUT PARAMETERS
%           A : a myline object
a = A.a;
```

and

```
function b = intercept(A)
% intercept.m An accessor method for myline class. Called as
%
%           b = intercept(A)
%
% INPUT PARAMETERS
%           A : a myline object
b = A.b;
```

Consider then, defining a line that passes through the points

$$\begin{aligned}(x_0, y_0) &= (0, 2) \\ (x_1, y_1) &= (1, 3)\end{aligned}$$

using the class constructor method; note that the display method is called in the statement below because the semicolon normally used to suppress echo is not included; the object A is created and we determine the methods available for use on this object:

```
>> A = myline([0 1],[2 3])
myline class object, with y = ax+b
a = 1
b = 2

>> methods myline

Methods for class myline:
display   intercept myline   plot       plus       slope     yval
```

Now we display the values of the linear equation coefficients using the slope and intercept methods:

```
>> a = slope(A)
a =
    1

>> b = intercept(A)
b =
    2
```

We now try the remainder of the methods that have been defined, first observing the results of the plot method in Fig. 2.1.

```

>> plot(A,-1,3)
>> hold on, grid, xlabel('x'), ylabel('y')
>> x = 2;
>> y = yval(A,x);
>> plot(x,y,'o'), hold off
>> B = A+1
myline class object, with y = ax+b
a = 1
b = 3
>> plot(B,-1,3,'--')
hold off

```

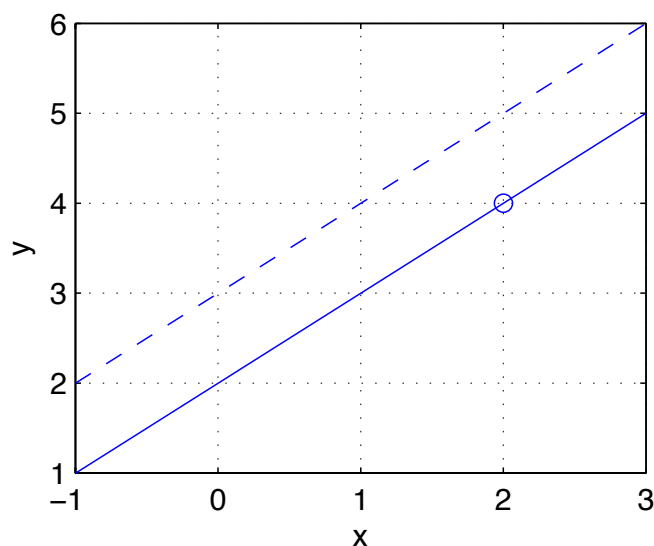


Figure 2.1: Plot of the original myline object *A* (solid line), and evaluation of a point on that line (circle), and the new line *B* found after adding the scalar value of 2 to *A*.

## 2.2 Inheritance

The myline class was constructed from scratch by defining a new data type and its associated methods. We can also build new classes from existing classes, where the new (derived or child) class inherits all the data fields and methods of the parent class; new data fields can be added to the child as well as methods. The new methods either overload some or all of the parent's methods, and entirely new methods can be defined. In either case, *the methods defined for the child class can only access data fields defined specifically for the child class; fields that were inherited from the parent can only be accessed through the methods defined for the parent class.*

Consider, for example, making the myline class more specific: a new class will be derived from myline to include all the methods and data fields of the parent, and will add two new datafields describing the limits of validity of the linear equation - think of this as defining the class of all linear models defined

from empirical data with the domain of validity of the correlation being  $x_{min} \leq x \leq x_{max}$ . Data fields of this new class are

A.myline (all data fields of the parent class)  
 A.xmin (Lower limit of validity)  
 A.xmax (Upper limit of validity))

The new constructor for this class is

```
function A = mylinelim(x,y,xmin,xmax)
% mylinelim.m Constructor method for mylinelim class. Called as
%
%           A = mylinelim(x,y,xmin,xmax)
%
% INPUT PARAMETERS
%       x : x values or a (slope)
%       y : y values or b (intercept)
%       xmin : x value minimum limit (optional)
%       xmax : x value maximum limit (optional)
if nargin == 4
    A.xmin = xmin; A.xmax = xmax;
elseif nargin == 2
    A.xmin = min(x); A.xmax = max(x);
else
    x = 0; y = 0; A.xmin = 0; A.xmax = 1;
end
B = myline(x,y);
A = class(A,'mylinelim',B);
```

Notice how the constructor of the derived class actually can be simpler than the same for the parent class. We now *overload* the display method of the myline class:

```
function display(A)
% display.m Display method for mylinelim class
disp('mylinelim class object, with y = ax+b')
disp(['a = ',num2str(slope(A))])
disp(['b = ',num2str(intercept(A))])
disp(['limits of validity, xmin: ', ...
      num2str(A.xmin), ' xmax: ',num2str(A.xmax)])
```

and, most importantly, the yval.m method:

```

function y = yval(A,x)
% yval.m Evaluate y = ax+b method for mylinelim class. Called as
%       y = yval(A,x)
%
% INPUT PARAMETERS
%       A : myline object
%       x : x values
y = yval(A.myline,x);
I = find(x<A.xmin | x>A.xmax);
if prod(size(I)) > 0
    warning('Some input values are outside the limits')
end

```

It is critical to note that the `slope.m`, `intercept.m`, `plot.m`, and `plus.m` methods operate only on the parent class data fields and so are inherited from the parent class and so do not have to be redefined for the new subclass. We use the new classes in the following simple example:

```

>> A = mylinelim([0 1],[2 3],0,1)
mylinelim class object, with y = ax+b
a = 1
b = 2
limits of validity, xmin: 0 xmax: 1
>> x = [-1 0 1 2];
>> y = yval(A,x)
yval: some input values are outside the limits
y =
     1     2     3     4

```

## Chapter 3

# Case Study: The associative array class

A data structure that stores a sequence of variable values *and their associated names* is called an associative array. In computer science terminology, these data structures are sometimes called hash tables. We define the `assocarray` class in MATLAB to perform this function; our motivation is that objects of this type are good for storing the sets of variables and parameters that define a nonlinear modeling problem, especially when these objects are unpack-ed into the workspace of a function.

The class is defined in the table below; the table form is a standard method to define a class and it takes the general form

class name
class attributes (variables)
class methods (functions)

and so for the `assocarray` class:

<b>assocarray</b>
key : char cell array val : double cell array
assocarray(key1,val1,key2,val2,...) delete(A,key) display(A) getval(A,key) insert(A,newval,newkey,l) keyindex(A,key) keys(A) setval(A,newval,key,l) sort(A) unpack(A) valdiff(A,B) values(A)

We now demonstrate several of the basic `assocarray` methods by creating an `assocarray` object containing the keys 'p' and 'q' and their associated values 5 and -2:

```
>> A = assocarray({'p',5,'q',-2})
assocarray object "A":
p:
    5
q:
   -2

>> keys(A)
ans =
    'p'    'q'

>> getval(A,'q')
ans =
   -2
```

Now we insert a new key 'r' and its associated value of 13, noting how the object A must be passed as input to the method and how the method returns the updated version of object A:

```
>> A = insert(A,12,'r',2)
assocarray object "A":
p : 2
r : 12
q : -2

>> keyindex(A,'r')
ans =
    2
```

We can sort the object A according to the keys using the `sort` method:

```
>> sort(A)
assocarray object "ans":
p : 2
q : -2
r : 12
```

### 3.1 Review problems

1. Create an `assocarray` object containing the symbols and atomic weights of the first 18 elements, arranged by atomic number. Then, using the `assocarray/sort` method, print the symbols and associated atomic weights in alphabetical order. Finally, repeat problem 5 of Chapter 1, but extract the molecular weight information from the `assocarray` object you created (pass the object through the function's input parameter list).

## Chapter 4

# Linear systems

### 4.1 Writing systems of linear equations in matrix form

Systems of linear equations themselves are extremely common and important models; for example, consider a set of coupled, linear equations written in the form

$$\mathbf{Ax} = \mathbf{b}. \quad (4.1)$$

A model of this form might arise from a material balance over a chemical reactor in which the following reaction takes place



If  $F$  represents the total molar flow rate of reactants to the reactor,  $k$  the reaction rate,  $x_{Pin}$  and  $x_{Qin}$  the feed mole fractions of species P and Q, respectively, and we assume the reactor contents are well mixed, we can write a material balance for species P and Q as

$$\begin{aligned} F(x_{Pin} - x_P) &= kx_P \\ F(x_{Qin} - x_Q) &= -kx_P \end{aligned}$$

Given these simple material balance equations, we can write them in matrix form (4.1) with

$$\mathbf{A} = \begin{bmatrix} k + F & 0 \\ -k & F \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_P \\ x_Q \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} Fx_{Pin} \\ Fx_{Qin} \end{bmatrix}$$

### 4.2 Matrix multiplication

The product  $\mathbf{Ax}$  in (4.1) requires us to define the matrix multiplication operation. For systems where  $\mathbf{x}$  is a column vector, we can write

$$b_i = \sum_{j=1}^J A_{ij}x_j \quad i = 1, \dots, I$$

where the first index of  $A_{ij}$  refers to the row and the second to the column of matrix  $\mathbf{A}$ . This operation fixes the sizes of the arrays as

$$\mathbf{A}^{I \times J} \mathbf{x}^{J \times 1} = \mathbf{b}^{I \times 1}.$$

In general, we can premultiply the array  $\mathbf{B}$  by array  $\mathbf{A}$  if the maximum column index of  $\mathbf{A}$  is identical to the maximum row index of  $\mathbf{B}$ :

$$\mathbf{A}^{I \times J} \mathbf{B}^{J \times K} = \mathbf{C}^{I \times K}$$

where

$$C_{i,k} = \sum_{j=1}^J A_{i,j} B_{j,k}, \quad i = 1, \dots, I, \quad k = 1, \dots, K.$$

It is not difficult to see that in general

$$\mathbf{AB} \neq \mathbf{BA}.$$

Finally, an important fact we will make extensive use of is that the transpose of a matrix product is equal to the product of the transpose of each term in reverse order:

$$[\mathbf{AB}]^T = \mathbf{B}^T \mathbf{A}^T.$$

### 4.3 Solving $\mathbf{Ax}=\mathbf{b}$

Our goal is to solve linear algebra problems of the form

$$\mathbf{A}^{I \times J} \mathbf{x}^{J \times 1} = \mathbf{b}^{I \times 1} \quad (4.2)$$

using numerical methods that are computationally reliable and give the most information about the structure of the modeling equations themselves. Algebraically, it is possible to write the solution to (4.2) as

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$$

if  $\mathbf{A}^{-1}$  exists, but computing  $\mathbf{A}^{-1}$  directly satisfies neither of the stated goals. Therefore, we wish to consider numerical methods that transform the hard problem to solve (4.2) into the easier problem

$$\mathbf{Ux} = \mathbf{d} \quad (4.3)$$

where  $\mathbf{U}$  has a structure that makes the solution to (4.3) simpler to find and gives insight into the the problem we are solving, such as whether there are a sufficient number of equations to determine a unique solution.

#### 4.3.1 Factorization by Gaussian Elimination

There are an infinite number of such transformations; we will discuss *Gaussian elimination with partial pivoting*, a numerical technique which reduces computational inaccuracies. Gaussian elimination applied to (4.2) consists of a  $J$ -step forward elimination procedure to obtain the system (4.3) followed by back substitution to compute the values  $\mathbf{x}$ . Note that  $J$  is the number of columns of  $\mathbf{A}$  and the number of variables in  $\mathbf{x}$ . The forward elimination procedure at each step  $j = 1, \dots, J$  consists of the following three main operations:

1. In the  $j$ th step of the elimination procedure, the **pivot** element is identified by finding the largest (in absolute value) element  $A_{k,j}$  of those found in the subcolumn below (and including) the diagonal element  $A_{j,j}$ ;



2. The pivot row elements  $A_{k,m=1,\dots,J}$  and  $b_k$  then are interchanged with the  $j$ th row (this leaves the arrays unchanged if the pivot point is the largest-magnitude element);
3. For each nonzero  $A_{m,j}$ ,  $m = j + 1, \dots, I$  in the rearranged system, a multiplier is computed as  $-A_{m,j}/A_{j,j}$  and the product of the multiplier and the pivot row is added to row  $m$ ; this procedure is used to remove all nonzero elements in each element below the pivot.

The main idea behind partial pivoting is that it tends to make the solution more resistant to small numerical errors. Generally, large multipliers (small pivots) contribute to numerical errors: in general it is a poor idea to multiply the pivot row with a large number and then add it to another, because the first might wash out any contribution of the second through round off error.

The array permutations (pivoting) and other forward-elimination operations can be represented as array operations. The coefficient array of (4.2) can be **factored** into the three distinct arrays

$$\mathbf{A}^{I \times J} = \mathbf{P}^{I \times I} \mathbf{L}^{I \times J} \mathbf{U}^{J \times J}$$

where  $\mathbf{L}$  and  $\mathbf{U}$  have a lower and upper triangular structure, respectively, such that the solution of (4.3) can be found by simple back substitution.  $\mathbf{P}$  contains row permutation (exchange) information. We note that the product  $\mathbf{PL}$  may not have a lower triangular structure. The elements of  $\mathbf{d}$  are also computed from the back substitution problem

$$\begin{aligned} \mathbf{PLd} &= \mathbf{b} \\ \text{or } \mathbf{Ld} &= \mathbf{Pb} \quad \text{because } \mathbf{P}^{-1} = \mathbf{P}. \end{aligned}$$

It is easy to prove that  $\mathbf{P}^{-1} = \mathbf{P}$ : because  $\mathbf{P}$  interchanges rows, an additional product operation will return the array to its original form. Therefore,  $\mathbf{PP} = \mathbf{I}$  and  $\mathbf{P}^{-1} = \mathbf{P}$  follows.

### 4.3.2 An example of solving sets of linear equations

Consider the following set of linear equations

$$\begin{aligned} x_M + x_O &= 1 \\ y_M - 220x_M &= 0 \\ y_0 - 0.5x_O &= 0 \\ y_M + y_O &= 1 \end{aligned}$$

and the problem of solving the set of linear equations  $\mathbf{Aw} = \mathbf{b}$  where  $w = [x_M, x_O, y_M, y_O]^T$ ; first, the coefficient arrays are constructed

```
>> A = [1 1 0 0; -220 0 1 0; 0 -0.3 0 1; 0 0 1 1] % coefficient array
A =
    1.0000    1.0000         0         0
   -220.0000         0    1.0000         0
         0   -0.3000         0    1.0000
         0         0    1.0000    1.0000
```

```
>> b = [1; 0; 0; 1] % nonhomogeneous terms
b =
     1
     0
     0
     1
```

and the the solution is computed in two different ways; first by inverting the matrix **A** and computing the product of the array inverse with the **b** array:

```
>> w = inv(A)*b; % solution found by matrix inversion
>> disp(w')
    0.0032    0.9968    0.7010    0.2990
```

It is simple to use the Gaussian elimination technique in MATLAB because factorization and back substitution are combined in the \ operation.

```
>> w = A\b; % Gaussian Elimination -- generally better
>> disp(w')
    0.0032    0.9968    0.7010    0.2990
```

Factorization alone is done with the lu.m routine:

```
>> [L,U,P] = lu(A);
```

The factorization routine lu returns three arrays

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -0.0045 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -0.3 & 0.0014 & 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} -220 & 0 & 1 & 0 \\ 0 & 1 & 0.0045 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0.9986 \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

where the original coefficient array can be recovered as  $\mathbf{P} \cdot \mathbf{L} \cdot \mathbf{U}$  and where the effect of the row interchanges can be observed in  $\mathbf{L} \cdot \mathbf{U}$ .

## 4.4 More equations than unknowns

To this point, the linear equation sets contained equal numbers of equations and unknowns, giving the square systems

$$\mathbf{A}^{I \times J} \mathbf{x}^{J \times 1} = \mathbf{b}^{I \times 1}$$

where  $I = J$ . For the case of  $I = J = 4$ , the Gaussian elimination procedure resulted in a system in upper-triangular form (where the \*'s denote potentially non-zero elements):

$$\begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix} \mathbf{x} = \mathbf{d} \quad \text{with} \quad \mathbf{d} = \begin{bmatrix} * \\ * \\ * \\ * \end{bmatrix}$$

and  $\mathbf{d} \neq \mathbf{b}$  in general.

In this chapter we mainly consider the cases of having more equations than unknowns – the opposite case normally can only be solved satisfactorily by returning to the modeling stage and finding the missing modeling equations or by computing the space in which the solutions exist. A more formal discussion of the latter concept situation will take place in the following section when the null space is defined.

Cases of  $I > J$  can be split into two distinctly different classes of computational problems

1. **Redundant** equations, where one or more equations is not linearly independent from the rest;
2. **Overspecified** problems, where all of the equations are linearly independent and no unique solution exists.

The distinction between these cases is important for understanding the meaning of the modeling equations and for selecting the proper numerical solution procedure.

The basic matrix operations used in the Gaussian elimination procedure are valid for the case where more equations exist than unknowns and are the key to a rigorous test for distinguishing between these cases.

## 4.5 Linear vector spaces

A different way of looking at

$$\mathbf{A}^{I \times J} \mathbf{x}^{J \times 1} = \mathbf{b}^{I \times 1}$$

to write

$$\mathbf{A} = [\mathbf{a}_1 \ \mathbf{a}_2 \ \mathbf{a}_3]$$

so that

$$\mathbf{a}_1 x_1 + \mathbf{a}_2 x_2 + \mathbf{a}_3 x_3 = \mathbf{b}.$$

Writing the linear system in this form emphasizes that it is the linear combination of the  $\mathbf{a}_i$  that gives  $\mathbf{b}$  that defines the solution to this set of linear equations. Therefore, it is apparent that if  $\mathbf{a}_3$  is a linear combination of  $\mathbf{a}_1$  and  $\mathbf{a}_2$ , no unique value for  $x_3$  will exist. Note that this is different from the previous situations where there were more equations than unknowns – this procedure indicates when there are fewer equations than unknowns. A comprehensive discussion of this geometrical interpretation of the linear algebra problem is discussed in Strang [26].

If  $r$  is the number of linearly independent columns (which must be equal to the number of linearly independent rows), then  $r$  is the **rank** of matrix  $\mathbf{A}$  and the linearly-independent columns define the column space.  $\mathbf{Ax} = \mathbf{b}$  has a solution when  $\mathbf{b}$  is in the column space of  $\mathbf{A}$ . The **null space** of  $\mathbf{A}$  satisfies

$$\mathbf{Ax} = \mathbf{0}$$

The null space has dimension  $J - r$  and the solutions to  $\mathbf{Ax} = \mathbf{b}$  differ by a vector in the nullspace. If the null space has zero dimension,  $\mathbf{x} = \mathbf{0}$  is the only solution to the homogeneous system, thus, the solution to  $\mathbf{Ax} = \mathbf{b}$  is unique.

The length of a vector (originating at the origin) is defined by a **norm**:

$$\begin{aligned} d &= \sqrt{a_{1,1}^2 + a_{2,1}^2 + a_{3,1}^2} \\ &= \sqrt{\mathbf{a}_1^T \mathbf{a}_1} \\ &= \sqrt{\langle \mathbf{a}_1, \mathbf{a}_1 \rangle} \\ &= \|\mathbf{a}_1\| \end{aligned}$$

where the notation  $\langle \cdot, \cdot \rangle$  denotes the **inner product**. We note that the norm defined by the inner product of two different vectors

$$\begin{aligned} d &= \sqrt{a_{1,1}a_{1,2} + a_{2,1}a_{2,2} + a_{3,1}a_{3,2}} \\ &= \sqrt{\langle \mathbf{a}_1, \mathbf{a}_2 \rangle} \end{aligned}$$

results in

1.  $d = 0$  if the two vectors  $\mathbf{a}_1$  and  $\mathbf{a}_2$  are **orthogonal**;
2. The vector  $\mathbf{a}_1$  has unit length if  $\|\mathbf{a}_1\| = 1$ , therefore a vector can be **normalized** by

$$\bar{\mathbf{a}}_1 = \frac{\mathbf{a}_1}{c_1} \quad \text{with} \quad c_1 = \|\mathbf{a}_1\|. \quad (4.4)$$

The **projection** of vector  $\mathbf{a}_2$  onto  $\mathbf{a}_1$  is the length  $l_1$  of  $\mathbf{a}_2$  along  $\mathbf{a}_1$ . It is computed as the value of  $l_1$  of the following expression

$$l_1 \mathbf{a}_1 = \mathbf{a}_2 + \mathbf{w}$$

where  $\mathbf{w}$  is a vector orthogonal to  $\mathbf{a}_1$ . If  $\mathbf{a}_1$  is nontrivial, we can make use of the orthogonality by projecting this equation onto  $\mathbf{a}_1$  and solve for  $l_1$ :

$$\begin{aligned} l_1 \langle \mathbf{a}_1, \mathbf{a}_1 \rangle &= \langle \mathbf{a}_2, \mathbf{a}_1 \rangle + 0 \\ l_1 &= \frac{\langle \mathbf{a}_2, \mathbf{a}_1 \rangle}{\langle \mathbf{a}_1, \mathbf{a}_1 \rangle} \\ &= \frac{\langle \mathbf{a}_2, \mathbf{a}_1 \rangle}{\|\mathbf{a}_1\|^2}. \end{aligned}$$

Note that

$$\begin{aligned} l_1 &= 1 \quad \text{if} \quad \mathbf{a}_1 = \mathbf{a}_2 \\ l_1 &= 0 \quad \text{if} \quad \mathbf{a}_1, \mathbf{a}_2 \quad \text{are orthogonal} \end{aligned}$$

#### 4.5.1 The Gram-Schmidt orthogonalization procedure

The Gram-Schmidt orthogonalization procedure produces an orthonormal basis  $\mathbf{Q}$  for a vector space using a sequence of normalization and projection operations. Consider three, three-element column arrays written in matrix form:

$$\mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \mathbf{b}_3] = \begin{bmatrix} 1 & 1 & -1 \\ 0.5 & 0 & -0.1 \\ 0 & 0 & 0.1 \end{bmatrix}$$

**Step 1** Normalize  $\mathbf{b}_1$ :

$$\mathbf{q}_1 = \frac{\mathbf{b}_1}{\|\mathbf{b}_1\|} = \begin{bmatrix} 0.8944 \\ 0.4472 \\ 0 \end{bmatrix}.$$

**Step 2** Project  $\mathbf{b}_2$  onto  $\mathbf{q}_1$  to find  $l_1 = \langle \mathbf{b}_2, \mathbf{q}_1 \rangle = 0.8944$ ; subtract the result from  $\mathbf{b}_2$ , and normalize the final result:

$$\mathbf{q}_2 = \frac{\mathbf{b}_2 - l_1 \mathbf{q}_1}{\|\mathbf{b}_2 - l_1 \mathbf{q}_1\|} = \begin{bmatrix} 0.4472 \\ -0.8944 \\ 0 \end{bmatrix}.$$

**Step 3** Repeat the procedure for the third vector  $\mathbf{b}_3$  to find  $l_1 = \langle \mathbf{b}_3, \mathbf{q}_1 \rangle = -0.9391$  and  $l_2 = \langle \mathbf{b}_3, \mathbf{q}_2 \rangle = -0.3578$  and:

$$\mathbf{q}_3 = \frac{\mathbf{b}_3 - l_2 \mathbf{q}_2 - l_1 \mathbf{q}_1}{\|\mathbf{b}_3 - l_2 \mathbf{q}_2 - l_1 \mathbf{q}_1\|} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Having completed the orthonormalization procedure, it is easy to check the calculations:

$$\mathbf{Q}^T \mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

To summarize,

1. Because  $\mathbf{Q}$  is orthonormal, the projection operation simply is an array product operation and so  $\mathbf{Q}^T = \mathbf{Q}^{-1}$ ;
2. If  $\mathbf{Q}$  is generated from  $\mathbf{A}$ , the columns of  $\mathbf{A}$  **span** the space defined by the orthonormal basis  $\mathbf{Q}$ .

## 4.6 Least squares

In cases where our factorization procedure reveals that there are more independent equations than unknowns, we search for the best solution possible for these cases by minimizing a *residual* corresponding to the solution estimate  $\mathbf{x}$ .

We define the residual of a set of linear equations  $\mathbf{Ax} = \mathbf{b}$  as

$$\mathbf{r} = \mathbf{Ax} - \mathbf{b}$$

noting that the residual is a vector with the same dimensions as  $\mathbf{b}$  (an  $N \times 1$  column vector). As we saw in the previous section, minimizing a vector means minimizing its length, which is defined by the Euclidean norm:

$$\begin{aligned} \text{length of } \mathbf{r} &= \|\mathbf{r}\| \\ &= \sqrt{\mathbf{r} \cdot \mathbf{r}} \\ &= \sqrt{\mathbf{r}^T \mathbf{r}} \end{aligned}$$

Having defined the residual, we minimize the square of the norm (to make computations easier) by determining the values of  $\mathbf{x}$  that minimize

$$\min_{\mathbf{x}} \|\mathbf{r}\|^2 = \min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|^2.$$

Now,

$$\begin{aligned}
 \|\mathbf{Ax} - \mathbf{b}\|^2 &= [\mathbf{Ax} - \mathbf{b}]^T [\mathbf{Ax} - \mathbf{b}] \\
 &= [(\mathbf{Ax})^T - \mathbf{b}^T] [\mathbf{Ax} - \mathbf{b}] \\
 &= (\mathbf{Ax})^T (\mathbf{Ax}) - (\mathbf{Ax})^T \mathbf{b} - \mathbf{b}^T (\mathbf{Ax}) + \mathbf{b}^T \mathbf{b} \\
 &= \mathbf{x}^T \mathbf{A}^T \mathbf{Ax} - 2\mathbf{b}^T \mathbf{Ax} + \mathbf{b}^T \mathbf{b}
 \end{aligned}$$

One can understand how the terms are combined in the final step above by remembering that each product in the sum evaluates to a scalar value. The derivatives with respect to the  $x_i$  will vanish at the extrema of the norm of the residual:

$$\begin{aligned}
 \frac{\partial}{\partial x_i} \|\mathbf{Ax} - \mathbf{b}\|^2 &= \frac{\partial}{\partial x_i} [\mathbf{x}^T \mathbf{A}^T \mathbf{Ax} - 2\mathbf{b}^T \mathbf{Ax}] \quad \text{for } i = 1, \dots, N \\
 &= \mathbf{A}^T(i, :) \mathbf{Ax} + \mathbf{x}^T \mathbf{A}^T \mathbf{A}(:, i) - 2\mathbf{b}^T \mathbf{A}(:, i) \\
 &= 2\mathbf{A}^T(i, :) \mathbf{Ax} - 2\mathbf{A}^T(i, :) \mathbf{b}
 \end{aligned}$$

using MATLAB notation to denote a column of the coefficient array as  $\mathbf{A}(:, i)$ . Writing the derivatives for all  $x_i$  in column vector form and setting the result to zero gives:

$$\mathbf{A}^T \mathbf{Ax} = \mathbf{A}^T \mathbf{b} \quad (4.5)$$

We note that the product of  $\mathbf{A}^T \mathbf{A}$  gives a square, symmetric array of the same length as  $\mathbf{x}$ . At this point we also note that it is easy to see that we are truly minimizing the residual because the second derivative with respect to each  $x_i$  is strictly positive:

$$\frac{\partial^2}{\partial x_i^2} \|\mathbf{Ax} - \mathbf{b}\|^2 = 2\mathbf{A}^T(i, :) \mathbf{A}(:, i).$$

The inner product above must be positive, otherwise one of the columns of  $\mathbf{A}$  would be zero.

It may be difficult to see how the differentiation was done in the minimization process used to find (4.5); therefore, if we write the problem out explicitly for a  $2 \times 2$  system and compute the derivatives with respect to  $x_1$  and  $x_2$ ,

$$\begin{aligned}
 &\frac{\partial}{\partial x_1} \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix}^T \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} - 2 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}^T \frac{\partial}{\partial x_1} \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} \\
 &= \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}^T \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} + \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix}^T \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} - 2 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}^T \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \\
 &= 2 \begin{bmatrix} A_{11} & A_{21} \end{bmatrix} \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} - 2 \begin{bmatrix} A_{11} & A_{21} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}
 \end{aligned}$$

Computing the derivative with respect to  $x_2$  reveals a similar structure, the only difference being in the transposed row of  $\mathbf{A}$  coefficients; again, stacking the array equations and setting them to zero allows us write the results of the minimization simply as determining the solution to (4.5).

#### 4.6.1 Linear regression models

An important application of the least-squares technique is in identifying the coefficients of linear regression models of the form

$$y = \beta_0 + \beta_1 x + \dots + \epsilon$$

where in this case  $x$  represents the input variable (factor) to an experiment (such as the temperature at which a reaction takes place),  $y$  is the measured output (such as the fraction of reactant converted in the reaction), and  $\epsilon$  is measurement noise with expected value  $E(\epsilon) = 0$  and variance  $Var(\epsilon) = \sigma^2$ . The ... refer to additional input variables or higher-degree terms in the model.

If we perform  $n$  experiments corresponding to input values  $x_i$ ,  $i = 1, \dots, n$ , we obtain  $n$  measured values  $y_i$ . We can write the resulting system of equations in matrix form

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

or

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

Given that the measured values  $y_i$  differ from their true values because of the (unmeasurable) noise term  $\epsilon_i$ , our goal is to determine the best possible fit of the measured data to the model

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{b}$$

where the coefficients  $b_n$  are our estimates of the  $\beta_n$ , and to estimate the statistics of the noise, specifically,  $\sigma^2$ . Following [22], we solve for the model coefficients  $\mathbf{b}$  by solving the least-squares problem

$$\begin{aligned} \mathbf{y} &= \mathbf{X}\mathbf{b} \\ \mathbf{b} &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \end{aligned}$$

With the model coefficients in hand, we now recover an approximation to the variance in the data by first defining the *residual sum of squares*:

$$SS_E = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \mathbf{e}^T \mathbf{e}$$

where  $e = y_i - \hat{y}_i$  are the residuals - the differences between the true and predicted values. From this definition of error, we compute the (model-dependent) approximation to the variance [22]:

$$\hat{\sigma}^2 = \frac{SS_E}{n - p}$$

where  $n - p$  represents the number of degrees of freedom in our regression problem and is defined as the difference between the number of data points  $n$  and the regression parameters  $p$ , the latter of which includes  $b_0$ .

Defining the overall deviation of each data point  $y_i$  from the mean of the data  $\bar{y}$  as the *total sum of squares*:

$$SS_T = \sum_{i=1}^n (y_i - \bar{y})^2$$

we can now assess the accuracy we gain (over simply averaging the measurements) using our regression model from the  $R^2$  value:

$$R^2 = 1 - \frac{SS_E}{SS_T}$$

In other words, as  $R^2 \rightarrow 1$ , the prediction error of our model grows vanishingly small relative to the deviation from the mean, and as  $R^2 \rightarrow 0$ , the model essentially performs no better than if we were simply to model the system as a constant determined by the mean value of the measurements.

### Transforming nonlinear regression problems into linear form

The least squares procedure described above works only if the problem can be written so that it is linear with respect to its parameters. Some problems that initially appear to be nonlinear regression problems can be converted into a linear form; consider, for example, the problem of determining the activation energy  $E_a$  and pre-exponential rate constant  $k_0$  in a chemical reaction rate equation of the form

$$R = k_0 e^{-E/T} p_A \quad (4.6)$$

where experimentally measured values of the reaction rate  $R$  are given as a function of temperature  $T$  and reactant partial pressure  $p_A$ ; equation 4.6 can be transformed into a linear equation by taking the natural logarithm of each side:

$$\ln R = \ln k_0 - \frac{E}{T} + \ln p_A,$$

Likewise, the parameters of the Antoine equation (used to determine the vapor pressure of a pure liquid)

$$\ln P^{sat} = A - \frac{B}{T + C} \quad (4.7)$$

are found by rearranging (4.7) to find

$$T \ln P^{sat} = -C \ln P^{sat} + AT + AC - B.$$

Defining  $D = AC - B$ , the least squares procedure can then be carried out to identify values of  $A$ ,  $C$ , and  $D$ , after which the value of  $B$  is determined by  $B = AC - D$ .

### 4.6.2 Computational example: Linear regression

In this example, we fabricate experimental data using a linear equation model and MATLAB's random number generator:

```
x = [0:0.05:1]'; n = length(x);
beta0 = 0.5; beta1 = 1; % true coefficient values
truevar = 0.05^2 % variance of noise added to data
randn('state',0)
y = beta0 + beta1*x + sqrt(truevar)*randn(n,1); % compute data points
```

Note how we fix MATLAB's random number generator's seed value to allow for reproducible results. Having calculated the "experimental data," we identify our model

$$\hat{y} = b_0 + b_1 x$$

and plot the data and interpolated linear equation in Fig. 4.1.



```

p = 2;
% perform linear regression
X = [ ones(n,1) x ];
coeff = X'*X\X'*y; % note precedence in action
b0 = coeff(1)
b1 = coeff(2)
yhat = b0 + b1*x;
plot(x,y,'ro',x,yhat)
xlabel('x')
legend('data','prediction',2)
grid on, axis tight

```

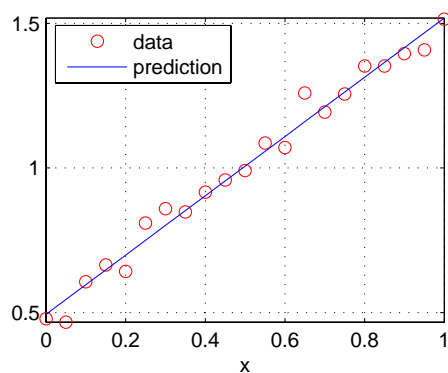


Figure 4.1: *Demonstration of least-squares fit of noisy data to a linear model.*

We observe that the identified values of the coefficients  $b_0 = 0.4936$  and  $b_1 = 1.0252$  are reasonable approximations to the true values. Likewise, knowing that the true value of the variance we set to  $\sigma^2 = 0.0025$ , we attempt to recover that value to find

```

>> SSE = (y - yhat)'*(y - yhat);
>> sighat = SSE/(n-p)

sighat =
    0.0019

```

which is reasonably close, and

```

>> SST = (y-mean(y))'*(y-mean(y));
>> R2 = 1 - SSE/SST

R2 =
    0.9828

```

which gives a strong indication that we have a good fit to our “experimental” data.

## 4.7 The singular value decomposition

Any matrix  $\mathbf{A}^{I \times J}$  can be decomposed into

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \quad (4.8)$$

where  $\mathbf{U}^{I \times I}$  contains the (orthogonal) left singular vectors,  $\mathbf{V}^{J \times J}$  contains the (orthogonal) right singular vectors, and  $\mathbf{\Sigma}^{I \times J}$  contains the  $r = \min(I, J)$  real, non-negative singular values on the diagonal. Therefore,  $\sigma_k \neq 0$  for a rank  $r$  matrix for  $k \leq r$  and  $\sigma_k = 0$  for all  $k > r$ .

The computational procedure defining the left and right singular vectors and singular values has a great deal of physical meaning. Consider matrix  $\mathbf{A}^{I \times J}$  and the problem of assessing how much a vector  $\mathbf{u}^{I \times 1}$ , defined in the space spanned by the columns of  $\mathbf{A}$ , contributes to each column in matrix  $\mathbf{A}$ . In other words, we compute the quantity  $\sigma$  by the projection operation

$$[\mathbf{A}^T \mathbf{u}]^T [\mathbf{A}^T \mathbf{u}] = \sigma^2 \quad (4.9)$$

which can be rearranged to

$$\begin{aligned} \mathbf{u}^T \mathbf{A} \mathbf{A}^T \mathbf{u} &= \sigma^2 \\ \mathbf{A} \mathbf{A}^T \mathbf{u} &= \sigma^2 \mathbf{u} \end{aligned}$$

if the  $\mathbf{u}_i$  constitute an orthonormal sequence of vectors; therefore, the left singular vectors are computed as the eigenvectors associated with the eigenvalue problem defined for  $I \times I$  matrix  $\mathbf{A} \mathbf{A}^T$  (eigenvalue problems will be defined in the next Chapter).

The right singular vectors can be found similarly using the transpose of  $\mathbf{A}$  denoted by  $\bar{\mathbf{A}} = \mathbf{A}^T$

$$\begin{aligned} [\bar{\mathbf{A}}^T \mathbf{v}]^T [\bar{\mathbf{A}}^T \mathbf{v}] &= \gamma^2 \\ \mathbf{v}^T \bar{\mathbf{A}} \bar{\mathbf{A}}^T \mathbf{v} &= \gamma^2 \\ \mathbf{A}^T \mathbf{A} \mathbf{v} &= \gamma^2 \mathbf{v} \end{aligned}$$

Solving the eigenvalue problem gives the  $J \times J$  array  $\mathbf{V}$ .

It can be shown that  $\gamma_i$  and  $\sigma_i$  are identical by rearranging (4.8) and comparing the results to the eigenvalue relationships:

$$\begin{aligned} \mathbf{A} \mathbf{v}_i &= \sigma_i \mathbf{u}_i & \mathbf{A}^T \mathbf{u}_i &= \sigma_i \mathbf{v}_i \\ \mathbf{A}^T \mathbf{A} \mathbf{v}_i &= \sigma_i \mathbf{A}^T \mathbf{u}_i & \mathbf{A} \mathbf{A}^T \mathbf{u}_i &= \sigma_i \mathbf{A} \mathbf{v}_i \\ \mathbf{A}^T \mathbf{A} \mathbf{v}_i &= \sigma_i^2 \mathbf{v}_i & \mathbf{A} \mathbf{A}^T \mathbf{u}_i &= \sigma_i^2 \mathbf{u}_i \end{aligned}$$

Finally, the singular value decomposition gives insight into the action of matrix  $\mathbf{A}$  on vector  $\mathbf{x}$

$$\mathbf{A} \mathbf{x} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T \mathbf{x}$$

in that  $\mathbf{V}^T \mathbf{x}$  projects the vector  $\mathbf{x}$  onto an orthogonal set of basis vectors spanning the column space of  $\mathbf{A}$ ; the  $\sigma_i$  then determine the degree of stretching or contraction along these directions, and then the column vectors that result from the product  $\mathbf{\Sigma} \mathbf{V}^T \mathbf{x}$  are used to determine the linear combination of basis vectors in  $\mathbf{U}$  to reconstruct the result of the matrix operation in the original space.

### 4.7.1 Computational example: orthogonalization using the SVD

Consider the array

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

It should be clear that the set of vectors spanning the column space of  $\mathbf{A}$  will contain only two vectors because the last two columns of  $\mathbf{A}$  are linear combinations of the first two. We should expect the singular value decomposition to produce

- a  $3 \times 3$  array  $\mathbf{U}$  where the first two columns are the orthonormal vectors spanning the column space of  $\mathbf{A}$ ; the third column of  $\mathbf{U}$  is orthogonal to the first two columns of the array *and is orthogonal to every column in  $\mathbf{A}$* ;
- two non-zero singular values  $\sigma_1, \sigma_2$  representing the contribution of each basis vector to all of the columns of  $\mathbf{A}$  (see eqn. 4.9);
- and a  $4 \times 4$  array  $\mathbf{V}$ , the columns of which can be used to reconstruct the basis vectors contained in  $\mathbf{U}$ , viz,

$$\mathbf{A}\mathbf{v}_i = \sigma_i\mathbf{u}_i \quad (4.10)$$

where  $\mathbf{v}_i$  is the  $i$ th column of  $\mathbf{V}$  and  $\mathbf{u}_i$  is the  $i$ th column of  $\mathbf{U}$ . Note that this implies that *those columns of  $\mathbf{V}$  corresponding to  $\sigma_i = 0$  define linear combinations of the columns of  $\mathbf{A}$  that vanish, making these  $\mathbf{v}_i$  orthogonal to the row space of  $\mathbf{A}$ .*

For this example we find

```
>> A=[1 0 2 1; 1 0 2 1; 0 1 0 1];
>> [U,S,V]=svd(A)
U =
   -0.7004   -0.0971   -0.7071
   -0.7004   -0.0971    0.7071
   -0.1374    0.9905   -0.0000
S =
    3.4923         0         0         0
         0    1.3431         0         0
         0         0    0.0000         0
V =
   -0.4011   -0.1447    0.9045   -0.0000
   -0.0393    0.7375    0.1005   -0.6667
   -0.8022   -0.2893   -0.4020   -0.3333
   -0.4405    0.5928   -0.1005    0.6667
```

It is easy to check that condition (4.10) holds:

```
>> u1 = A*V(:,1)/S(1,1)
u1 =
   -0.7004
   -0.7004
   -0.1374
```

and that the last two columns of  $\mathbf{V}$  are orthogonal to the row space of  $\mathbf{A}$

```
>> A*V(:,3:4)
ans =
    1.0e-14 *
    0.0860      0
    0.0860      0
   -0.3608   -0.0111
```

Finally, we note that

1. adding a column of zeros to the  $\mathbf{A}$  array does not change the orthogonal basis vectors  $\mathbf{U}$  or the non-zero singular values  $\sigma_i$ ; the first  $I$  columns of the  $\mathbf{V}$  array change only by the addition of a row of zeros in the row number corresponding to the column number of the zeros in  $\mathbf{A}$ , for example

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 2 & 1 & 0 \\ 1 & 0 & 2 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \text{ generates } \mathbf{V} = \begin{bmatrix} -0.4011 & -0.1447 & 0.9045 & -0.0000 & 0 \\ -0.0393 & 0.7375 & 0.1005 & -0.6667 & 0 \\ -0.8022 & -0.2893 & -0.4020 & -0.3333 & 0 \\ -0.4405 & 0.5928 & -0.1005 & 0.6667 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

2. and if the columns of  $\mathbf{A}$  are orthogonal,  $\mathbf{V}$  becomes the identity array and the nonzero singular values are the coefficients that normalize the corresponding columns of  $\mathbf{A}$  (see eqn. 4.4).

## 4.7.2 Matrix condition number

We note an important connection between the singular value decomposition and the solution to

$$\mathbf{Ax} = \mathbf{b}$$

whether the system is square with solution

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

or requires the least squares solution procedure:

$$\mathbf{x} = [\mathbf{A}^T \mathbf{A}]^{-1} \mathbf{A}^T \mathbf{b}.$$

In either case, the solution can be written as

$$\mathbf{x} = \mathbf{V} \mathbf{\Sigma}^{-1} \mathbf{U}^T \mathbf{b}$$

This relationship gives remarkable insight into the numerical accuracy of computing the solution to any linear system. Written in the form above, we see that the solution consists of the projection of  $\mathbf{b}$  onto the orthonormal basis vectors  $\mathbf{U}$  spanning the column space of  $\mathbf{A}$ ; elements in the resulting vector  $\mathbf{c} = \mathbf{U}^T \mathbf{b}$  then are multiplied by the  $\sigma_i$ . By taking the largest singular value  $\sigma_1$  and the singular value corresponding to the number of variables  $\sigma_J$ , inverting the two means the largest contribution to the solution  $\mathbf{x}$  corresponds to  $1/\sigma_J$  and the smallest to  $1/\sigma_1$ ; in other words, the solution takes the form

$$\mathbf{x} = \begin{bmatrix} V_{1,1}c_1/\sigma_1 + \cdots + V_{1,J}c_J/\sigma_J \\ \vdots \\ V_{J,1}c_1/\sigma_1 + \cdots + V_{J,J}c_J/\sigma_J \end{bmatrix}.$$

Therefore, the accuracy of the solution in terms of assessing the information lost due to finite-precision truncation errors in terms of decimal digits lost is

$$\text{lost decimal places} = \log_{10}(1/\sigma_J) - \log_{10}(1/\sigma_1) = \log_{10}\left(\frac{\sigma_1}{\sigma_J}\right)$$

Therefore, the **condition number** of a matrix can be defined as

$$C = \frac{\sigma_{\text{smallest}}}{\sigma_{\text{largest}}}$$

with small condition numbers corresponding to nearly singular systems.

### 4.7.3 Computational example: solving linear systems

Many of the linear systems concepts necessary for understanding the solution of linear system have been incorporated into the `double/linearsystemsolver.m` method of the `mdpsas` software library. Consider solving many of the previously discussed examples in the following manner:

```
>> A = [1 1 0 0; -220 0 1 0; 0 -0.5 0 1; 0 0 1 1];
>> b = [1; 0; 0; 1];
>> x = linearsystemsolver(A,b,1)
```

```
Number of unknowns = number of equations: 4
Approximate rank: 4
Gaussian elimination solution
x =
    0.0023
    0.9977
    0.5011
    0.4989
```

and now the obviously singular system:

```
>> A = [1 1 0 0; 2 2 0 0; 0 -0.5 0 1; 0 0 1 1];
>> b = [1; 0; 0; 1];
>> rcond(A)
ans =
    0
>> x = linearsystemsolver(A,b,1)
```

```
Number of unknowns: 4
Number of equations: 4
Approximate rank: 3
Rank deficient problem ==> no unique solution
x =
    NaN
```

and now more equations than unknowns, but with one redundant equation:

```
>> A = [1 1 -1; 0.5 0 -0.1; 0 0 0.1; 0.5 1 -0.9];
>> b = [0; 0; 10; 0];
>> x = linearsystemsolver(A,b,1)
```

```
Number of unknowns = number of equations: 3
Approximate rank: 3
Gaussian elimination solution
x =
    20.0000
    80.0000
   100.0000
```

and finally, more independent equations than unknowns:

```
>> A = [1 1 -1; 0.5 0 -0.1; 0 0 0.1; 0.5 1 1];
>> b = [0; 0; 10; 0];
>> x = linearsystemsolver(A,b,1)

Number of unknowns: 3
Number of equations: 4
Approximate rank: 3
More equations than unknowns ==> least squares solution
x =
    1.2088
   -0.9066
    0.8242
```

## 4.8 Review problems

1. Consider the following arrays written in MATLAB format:

```
A = [1 2 3; 3 2 5; 1 3 5]
B = [1; 1; 3]
C = [ 3 2 -1]
D = [1 2; 4 5; 2 -5]
E = [-2 -3 1; -1 2 3]
```

Construct a table indicating which products exist (e.g., AB, CD, DA, etc.), and for those cases where a product exists, compute the result.

2. Create a MATLAB script for computing solutions to the system  $Ax = b$  for the follow systems; which have unique solutions; why cannot some of the solutions be found?

```
A = [ 1 2 3; 3 4 5; 5 5 6] and b = [1; 2; 4]
A = [ 1 2 3; 3 4 5; 4 6 8] and b = [1; 2; 4]
A = [ 1 2 ; 3 4] and b = [2; 4]
A = [ 1 2 3 4; 3 4 5 6; 4 5 6 7; 5 5 6 9] and b = [1; 2; 4; 6]
A = [ 1 2 3 4; 3 4 5 6; -1 5 6 7; 5 5 6 9] and b = [1; 2; 4; 6]
```

3. Repeat the previous problem with

$$A = [1, 1, -1; 0.5, 0, -0.1; 0, 0, 0.1; 0.5, 1, -0.9]$$

$$b = [0; 0; 10; 0]$$

$$A = [1, 1, 1, 0, 0, 0; -1, 2, 0, 0, 0, 0; 0, 0, 2, 0, 0, 0; \dots$$

$$0, -1, -1, 0, 0, 0; 0, 0, 0, -1, -1, -1; 0, 0, 0, 1, -2, 0; \dots$$

$$0, 0, 0, 0, 0, -2; 0, 0, 0, 0, 1, 1]$$

$$b = [1; 2; 2; -5/3; -1; -2; -2; 5/3]$$

$$A = [1, 1, 1, 1e-4; -1, 2, 0, 0; 0, 0, 2, 0; 0, -1, -1, 0]$$

$$b = [1; 2; 2; -5/3]$$

4. Write the following equations in matrix form and solve using the `linearsolver.m` function.

$$\begin{aligned} 2x + 3y &= -2z - 1 \\ 3x &= 0 \\ x + y + z - 2 &= 0 \end{aligned}$$

$$\begin{aligned} 0 &= -2z - 1 \\ 3x + 5y - z &= 0 \\ x + y + z - 3 &= 0 \end{aligned}$$

$$\begin{aligned} 0 &= -2z - 1 \\ 3x + 5y - z &= 0 \\ 6x + 10y + 2z - 3 &= 0 \end{aligned}$$

5. Using Gaussian elimination without pivoting, compute the solutions to sets of linear equations where the elements of the  $A$  and  $b$  are defined below. Be sure to show all your work during the forward elimination portion of the solution

$$A = [1 \ 2 \ ; \ 3 \ 4] \text{ and } b = [2; 4]$$

$$A = [1 \ 2 \ 3; 3 \ 4 \ 5; 5 \ 5 \ 6] \text{ and } b = [1; 2; 4]$$

$$A = [1 \ 2 \ 3 \ 4; 3 \ 4 \ 5 \ 6; 4 \ 5 \ 6 \ 7; 5 \ 5 \ 6 \ 9] \text{ and } b = [1; 2; 4; 6]$$

$$A = [1 \ 2 \ 3 \ 4; 3 \ 4 \ 5 \ 6; -1 \ 5 \ 6 \ 7; 5 \ 5 \ 6 \ 9] \text{ and } b = [1; 2; 4; 6]$$

6. Compare your hand-computed solutions to those produced using `linearsystemsolver.m`; in particular, compare the number of independent equations identified by this function to your hand calculations.
7. An insulating wall is made up of a sequence of insulation layers of different thicknesses and thermal conductivities. If the temperature of the interface between each later is denoted as  $T_j, j = 0, 1, \dots, 4$  located at position  $z_j$ , the heat flux  $q$  through each layer can be approximated by

$$q = k_j \frac{T_j - T_{j-1}}{z_j - z_{j-1}}$$

Given  $T_0 = 0^\circ \text{C}$ ,  $T_4 = 100^\circ \text{C}$ ,  $k_{1,2,3,4} = [3, 1.5, 5, 2] \text{ W/(mK)}$  and  $z_{0,1,2,3,4} = [0, 0.1, 0.2, 0.4, 0.45] \text{ m}$ , set up the four linear equations needed to solve for  $T_1$ ,  $T_2$ ,  $T_3$ , and  $q$ ; solve this system of equations by hand using Gaussian Elimination with partial pivoting; compare the hand-calculated solution to a solution computed using MATLAB.

8. The resistance of a metal follows the following relationship:

$$R = a + bT + cT^2$$

Determine parameter values,  $a$ ,  $b$ , and  $c$ , using the following the two data sets. Set up a least-squares problem for each case and determine which data set contains noisy data, assessing the approximate value of  $\sigma^2$  in each case.

$T(K)$	300	350	400	450	500	550	600
$R^A(\text{Ohms})$	15.0	17.9	21.1	24.5	28.1	31.9	36.0
$R^B(\text{Ohms})$	13.2	16.0	19.5	22.3	25.0	28.7	31.0

9. A rate at which a chemical reaction takes place at a constant temperature is hypothesized to have the form

$$\text{Rate} = kc^2$$

where  $c$  is the chemical species concentration (moles/ $\text{m}^3$ ) and  $k$  is a reaction rate coefficient ( $\text{m}^3/(\text{moles sec})$ ). Given the following data, determine the rate coefficient  $k$  using least squares; plot the data points and the fitted curve.

$c$	$\frac{\text{moles}}{\text{m}^3}$	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5
Rate	$\frac{\text{moles}}{\text{m}^3 \text{ sec}}$	0.2047	0.5166	0.9404	1.5553	1.7459	2.6536	2.8818	4.3548	5.4650

10. As a function of temperature, a reaction rate coefficient is found to take the form

$$k = k_0 e^{-E_a/T}$$

where  $k_0$  is a rate constant,  $E_a$  is the activation energy divided by the gas constant, and  $T$  is temperature. Given the following data, determine the rate constant  $k_0$  and activation energy  $E_a$  using least squares; plot the data points and the fitted curve.

$T$	$K$	250	300	350	400	500	600	800
$k$	$\frac{\text{m}^3}{\text{moles sec}}$	13.3695	21.0674	29.8008	30.2843	36.4605	57.2018	56.3843



## Chapter 5

# Case Study: Nonadiabatic methane/octane flash drum

As an example that makes use of many linear algebra concepts, consider the nonadiabatic flash separation system shown in Fig. 5.1. It is assumed the system is operated in such a way that at equilibrium, two phases (liquid and vapor) exist, and that the drum temperature and pressure can be maintained at some specified set of values. Under these conditions, component M (methane) has a boiling point of  $-162^{\circ}\text{C}$  and component O (*n*-octane)  $126^{\circ}\text{C}$ ; we should expect that virtually all of the methane leaves through the vapor stream and that there is a distribution of octane in both product streams.

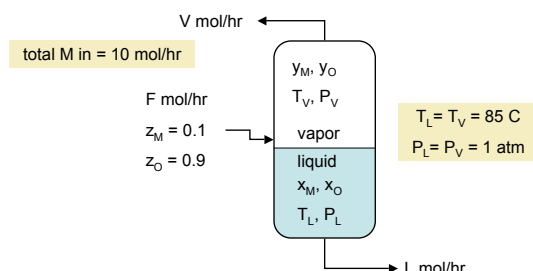


Figure 5.1: The single-stage separator.

### 5.1 Model derivation

By the definition of mole fractions

$$\begin{aligned}x_M + x_O &= 1 \quad (\text{liquid mole fractions}) \\y_M + y_O &= 1 \quad (\text{vapor mole fractions})\end{aligned}$$

Phase equilibrium is achieved when the *chemical potential*  $\mu_{\alpha}^i$  of component  $i$  is equal in each of the phases  $\alpha$ . The methane/octane mixture can be accurately approximated by properties of ideal mixtures,

so

$$\begin{aligned}\mu_I^M(x_M, x_O, T_I, P_I) - \mu_v^M(y_M, y_O, T_v, P_v) &= 0 \implies y_M/x_M = K_M(T, P) = 220 \\ \mu_I^O(x_M, x_O, T_I, P_I) - \mu_v^O(y_M, y_O, T_v, P_v) &= 0 \implies y_O/x_O = K_O(T, P) = 0.3\end{aligned}$$

The K values are taken from [25]. The material balances and definition of total methane feed rate give the four equations

$$\begin{array}{llll}\text{Mass in} & = & \text{Mass out} & \\ F & = & V + L & \text{overall} \\ Fz_M & = & Vy_M + Lx_M & \text{methane} \\ Fz_O & = & Vy_O + Lx_O & \text{octane} \\ Fz_M & = & 10 & \text{by definition}\end{array}$$

At this point we define the vector of state variables

$$\mathbf{x} = \begin{bmatrix} x_M \\ x_O \\ y_M \\ y_O \\ V \\ L \\ F \end{bmatrix}$$

and so it appears there are 7 variables and 8 equations, some of which are nonlinear due to product terms such as  $Vy_O$ .

If we rearrange the equations,

$$\begin{array}{llll}x_M + x_O & = & 1 & \\ y_M - 220x_M & = & 0 & \\ y_O - 0.3x_O & = & 0 & \\ y_M + y_O & = & 1 & \uparrow \text{ Intensive} \\ \hline V + L - F & = & 0 & \downarrow \text{ Extensive} \\ y_M V + x_M L - z_M F & = & 0 & \\ z_M F & = & 10 & \\ y_O V + x_O L - z_O F & = & 0 & \end{array} \quad (5.1)$$

the full set of modeling equations can be split into two sets (corresponding to intensive and extensive variables) which, if solved consecutively, can be solved independently:

$$\begin{aligned}\begin{bmatrix} 1 & 1 & 0 & 0 \\ -220 & 0 & 1 & 0 \\ 0 & -0.3 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_M \\ x_O \\ y_M \\ y_O \end{bmatrix} &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} & \text{or} & \mathbf{Ay} = \mathbf{b} \\ \begin{bmatrix} 1 & 1 & -1 \\ y_M & x_M & -z_M \\ 0 & 0 & z_M \\ y_O & x_O & -z_O \end{bmatrix} \begin{bmatrix} V \\ L \\ F \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \\ 10 \\ 0 \end{bmatrix} & \text{or} & \mathbf{Bz} = \mathbf{c}.\end{aligned}$$

## 5.2 Solution computed using partial pivoting

Returning to the four modeling equations corresponding to the intensive variable of the flash drum example (5.1), consider a solution obtained with pivoting, starting with interchanging the first two rows:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{A} \mathbf{y} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{b}$$

$$\begin{bmatrix} -220 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & -0.3 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

so if we multiply row 1 by 1/220 (a small multiplier) and add the product to row 2 (do not forget the RHS!):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1/220 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -220 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & -0.3 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1/220 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} -220 & 0 & 1 & 0 \\ 0 & 1 & 1/220 & 0 \\ 0 & -0.3 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Our multiplier is now 3/10, which gives

$$\begin{bmatrix} -220 & 0 & 1 & 0 \\ 0 & 1 & 1/220 & 0 \\ 0 & 0 & 3/2200 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 3/10 \\ 1 \end{bmatrix}$$

so interchanging rows 3 and 4,

$$\begin{bmatrix} -220 & 0 & 1 & 0 \\ 0 & 1 & 1/220 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 3/2200 & 1 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 3/10 \end{bmatrix}$$

and using  $-3/2200$  as the multiplier gives

$$\begin{bmatrix} -220 & 0 & 1 & 0 \\ 0 & 1 & 1/220 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 - 3/2200 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 3/10 - 3/2200 \end{bmatrix}.$$

The solution is found by *back substitution*. We now notice a very important point: If we do all of our calculations ignoring the small terms such as  $1/220$  and  $3/2200$ ,

$$\begin{aligned} y_O &\approx 0.3 \\ y_M &\approx 1 - 0.3 = 0.7 \\ x_O &\approx 1 \\ x_M &\approx 0.3/220 \approx 0.0 \end{aligned}$$

The same results would have been found if we neglected these terms during the elimination steps. We compare these results to those generated by a commercial process simulator (ChemCAD) to find

$$\begin{aligned} y_O &= 0.28 \\ y_M &= 0.72 \\ x_O &= 0.9977 \\ x_M &= 0.0023 \end{aligned}$$

illustrating the accuracy of our computational procedure.

### 5.2.1 The solution without pivoting

We now compare the results obtained without pivoting:

$$\begin{aligned} \begin{bmatrix} 1 & 1 & 0 & 0 \\ -220 & 0 & 1 & 0 \\ 0 & -0.3 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 220 & 1 & 0 \\ 0 & -0.3 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} &= \begin{bmatrix} 1 \\ 220 \\ 0 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 220 & 1 & 0 \\ 0 & 0 & 3/2200 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \mathbf{y} &= \begin{bmatrix} 1 \\ 220 \\ 3/10 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 220 & 1 & 0 \\ 0 & 0 & 3/2200 & 1 \\ 0 & 0 & 0 & -2200/3 + 1 \end{bmatrix} \mathbf{y} &= \begin{bmatrix} 1 \\ 220 \\ 3/10 \\ -219 \end{bmatrix} \end{aligned}$$

which means we find by back substitution and sloppy arithmetic an incorrect result:

$$\begin{aligned} y_O &\approx 0.3 \\ y_M &\approx 2200(0.3 - 0.3)/3 = 0 \\ x_O &\approx 1 \\ x_M &\approx 1 - 1 = 0 \end{aligned}$$

Why are the results so different and so clearly incorrect in the calculations without pivoting? The small error generated in the first back substitution step (0.3 instead of the more accurate 0.2990) is greatly magnified ( $2200/3 \times$ ) in the second step.

### 5.3 Number of equations $\neq$ number of unknowns

Recall that the basic matrix operations used in the Gaussian elimination procedure are valid for the case where more equations exist than unknowns and are the key to a rigorous test for distinguishing between these cases. Consider, for example, if we use our correct approximate solution  $\mathbf{y} = [x_M \ x_O \ y_M \ y_O]^T \approx [0 \ 1 \ 0.7 \ 0.3]^T$  to compute the coefficients of  $\mathbf{B}$ ,

$$\mathbf{B}\mathbf{z} = \begin{bmatrix} 1 & 1 & -1 \\ 0.7 & 0 & -0.1 \\ 0 & 0 & 0.1 \\ 0.3 & 1 & -0.9 \end{bmatrix} \begin{bmatrix} V \\ L \\ F \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 10 \\ 0 \end{bmatrix} = \mathbf{c}$$

Proceeding with the elimination procedure as before, we find

$$\begin{bmatrix} 1 & 1 & -1 \\ 0 & -0.7 & 0.6 \\ 0 & 0 & 0.1 \\ 0 & 0.7 & -0.6 \end{bmatrix} \mathbf{z} = \begin{bmatrix} 0 \\ 0 \\ 10 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & -1 \\ 0 & -0.7 & 0.6 \\ 0 & 0 & 0.1 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{z} = \begin{bmatrix} 0 \\ 0 \\ 10 \\ 0 \end{bmatrix} \quad (5.2)$$

which is easily solved to find

$$F = 100 \text{ moles/hr} \quad L = 85.7 \text{ moles/hr} \quad V = 100 - 85.7 = 14.3 \text{ moles/hr.}$$

As before, we compare these results to those produced by a commercial process simulator:

$$L = 86.5 \text{ moles/hr} \quad V = 13.4 \text{ moles/hr}$$

which gives us some sense of confidence in our computational procedure.

#### Least-squares approach

We can also solve the 4-equation and 3-unknown problem directly with the least-square approach:

$$\mathbf{B}^T \mathbf{B} = \begin{bmatrix} 1.58 & 1.30 & -1.34 \\ 1.30 & 2.00 & -1.90 \\ -1.34 & -1.90 & 1.83 \end{bmatrix} \quad \mathbf{B}^T \mathbf{c} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

It is left as an exercise for the reader to show that this system is equivalent to (5.2).

## 5.4 Review problems

1. Write the following set of three equations in matrix form  $\mathbf{Aw} = \mathbf{b}$  given  $x_M = 0.0032$ ,  $x_O = 0.9968$ ,  $y_M = 0.7010$ ,  $y_O = 0.2990$ ,  $z_M = 0.1$ ,  $z_O = 0.9$ , and  $\mathbf{w} = [FVL]^T$

$$Fz_M = y_M V + x_M L$$

$$Fz_M = 10$$

$$Fz_O = y_O V + x_O L$$

Solve for the unknowns  $F$ ,  $V$ , and  $L$  using Gaussian Elimination (GE) without partial pivoting; substitute your solution into the matrix equation defined in the first part of this problem to assess the accuracy of your solution; find the matrix  $\mathbf{C}$  such that  $\mathbf{CA}$  gives the upper triangular matrix found as a result of the forward elimination step of the GE procedure; show that  $\mathbf{Cb}$  produces the same vector of nonhomogeneous terms found in the hand-calculated GE procedure.

## Chapter 6

# Case Study: Underdetermined systems

In this section, we consider sets of linear algebraic equations where the number of variables exceeds the number of equations. Without additional information, unique solutions for these cases cannot be found. However, the solutions we find yield important information regarding the systems under consideration.

### 6.1 Diesel fuel blending

In this case, we consider the problem of blending three feedstocks to produce diesel fuel that meets certain specifications, such as that the final product must contain 0.05 wt% sulfur<sup>1</sup>. We blend three feedstocks with qualities listed in Table 6.1 to achieve the product specification.

Feedstock	blend wt%	sulfur wt%
A	$m_A$	0.04
B	$m_B$	0.03
C	$m_C$	0.06

Table 6.1: *Diesel blend feedstock qualities*

During the solution of this problem, we will keep in mind that feedstock C, having the highest sulfur content, will be treated as the lowest-value blend component, and so in the end we will want to maximize  $m_C$  in the blend. Assuming that the product diesel fuel properties can be described by a simple mixing rule (each feedstock contributes to the blend property in a manner proportional to the weight fraction of that feed), we can write the mass balance for the sulfur specification as

$$0.04m_A + 0.03m_B + 0.06m_C = 0.05$$

and the definition of the weight fractions as

$$m_A + m_B + m_C = 1.$$

<sup>1</sup>These are the maximum permissible levels as set by the US EPA in page 5135 of the Federal Register/Vol. 66, No. 12/Thursday, January 18, 2001/Rules and Regulations.

Written in matrix form

$$\begin{bmatrix} 4 & 3 & 6 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} m_A \\ m_B \\ m_C \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

we notice that the sulfur mass balance equation was scaled to simplify the arithmetic of the factorization procedure; this operation has no effect on the solution. One step of the forward elimination procedure gives

$$\begin{bmatrix} 4 & 3 & 6 \\ 0 & 1/4 & -1/2 \end{bmatrix} \begin{bmatrix} m_A \\ m_B \\ m_C \end{bmatrix} = \begin{bmatrix} 5 \\ -1/4 \end{bmatrix}$$

and so the bottom equation can be written as

$$m_B - 2m_C = -1 \quad \text{or} \quad m_B = 2m_C - 1$$

implying that  $m_C \in [1/2, 1]$  for a physically meaningful solution. Using the equation above, value of  $m_A$  can be computed

$$m_A = 2 - 3m_C$$

which implies  $m_C \in [1/3, 2/3]$  to produce a physically meaningful solution. Combining the two ranges of validity, we see that  $m_C \in [1/2, 2/3]$  for both conditions to be valid. Returning to the issue of maximizing  $m_C$ , the optimal solution for this problem is  $m_C = 2/3$ .

We can check our limits of validity by plotting the weight fractions  $m_A$ ,  $m_B$  as a function of  $m_C$ . We first create the function `dieselblend.m`:

```
function [wA, wB] = dieselblend(wC)
% calculate weight fractions wA, wB given wC
wB = 2*wC-1;
wA = 2-3*wC;
```

(note how the function will operate properly when the input parameter  $w_C$  is a matrix) and then the script

```
mC = [0 1];
[mA,mB]=dieselblend(mC);
plot(mC,mA,mC,mB,'--')
grid on
xlabel('mC'); ylabel('wt fraction'); legend('mA','mB')
```

Results can be seen in Fig. 6.1, where it can be clearly seen that  $m_C \in [1/2, 2/3]$  in order that  $m_A$  and  $m_B$  be positive.

## 6.2 Chemical reaction stoichiometry

Chemical reactions can involve a very large number of chemical species, including the species of primary interest as well as intermediate and undesirable chemical compounds. Especially when writing chemical reactions for equilibrium calculations, the form of the reaction may not be at all related to the fundamental reaction processes that are actually taking place. In such cases, we wish to find the minimum number of reactions that must be written that *guarantees that all possible reactions between a specified list of*



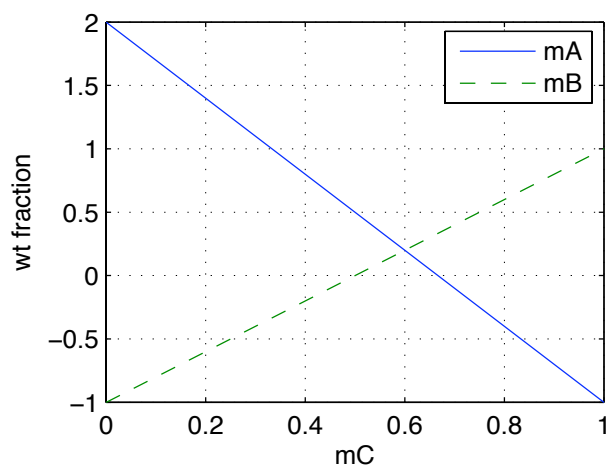


Figure 6.1: Diesel blend mixtures resulting in 0.05 wt% sulfur.

species are accounted for. This calculation depends directly on finding the null space of a set of linear algebraic equations.

We consider the problem of determining all possible chemical reactions that can take place in the vapor phase of a chemical vapor deposition reactor system used to deposit thin films of copper oxide materials for solar energy applications<sup>2</sup>. The vapor phase components of interest are CuI, Cu, Cu<sub>2</sub>, and I<sub>2</sub>; ultimately, we would like to determine the equilibrium composition of a mixture containing these compounds, but for now, consider only the problem of finding the chemical reactions that link all the components. We start by forming an array **A**, with columns defined by the four chemical species, and rows by the elements making up the compounds:

$$\begin{array}{c|cccc}
 & \text{CuI} & \text{Cu} & \text{Cu}_2 & \text{I}_2 \\
 \hline
 \text{Cu} & 1 & 1 & 2 & 0 \\
 \text{I} & 1 & 0 & 0 & 2
 \end{array} \implies \mathbf{A} = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix}$$

It is clear that the rows of **A** are linearly independent and so the  $\text{rank}(\mathbf{A}) = 2$ , and so given the previous discussion of the null space, the dimension of the null space must be  $4 - 2 = 2$  (the value of 4 is the number of columns in **A**), and so we will be able to connect all four chemical reaction species using two reactions. The MATLAB calculations are as follows

```

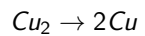
>> A = [1 1 2 0; 1 0 0 2];
>> null(A,'r')
ans =
     0    -2
    -2     2
     1     0
     0     1

```

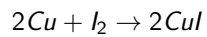
We note that the 'r' argument to the null function produces solutions which are not normal orthogonal, but contain integer-valued elements making for more sensible stoichiometric coefficients.

<sup>2</sup>SPIE paper reference

The standard notation for reaction stoichiometry is to use positive coefficients for reactants, negative coefficients for reaction products. The first null space vector (left column) thus gives the reaction



and the second gives the reaction



both of which make physical sense and what would have been likely choices based on our intuition.

## Chapter 7

# Case Study: Enzyme kinetics

Because biochemical reactions take place at essentially room temperature (as compared to the highly elevated temperatures of most chemical manufacturing processes), catalysts are needed to allow these reactions to proceed at biologically relevant rates. Enzymes are bio-catalysts that bind with a reactant chemical species (called the substrate) and then either rearrange or break the substrate species to form a new product species.

An important conceptual model of the enzyme action on the substrate is given by the *lock and key* model (see Fig. 7.1); the enzyme has a binding site that is configured so that only molecules of a specific configuration (and composition) can bind with that site. Once the substrate is bound to that receptor, the chemical reaction takes place, and the products of the reaction are released. While conceptually attractive, this model has been supplanted by others (e.g., the *induced fit model*) that give a more refined understanding of the overall process.

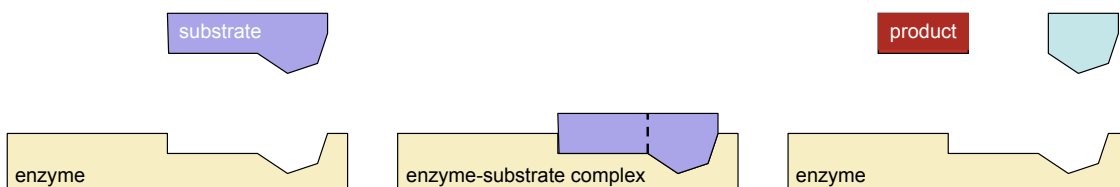
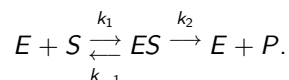


Figure 7.1: The role of an enzyme in catalyzing the decomposition of a substrate chemical species into a product through the lock and key mechanism.

Given this physical description of the reaction process, we can begin to quantify the rate at which the reactions take place by first defining the chemical species  $E$  = enzyme,  $P$  = product, and  $S$  = substrate (reactant). With this notation, the reactions to form the enzyme-substrate complex  $ES$  and the subsequent product  $P$  can be written as



## 7.1 Michaelis-Menten kinetics

The total rate at which the substrate  $S$  is consumed and product species  $P$  is produced will depend on the concentration of the relevant chemical species in the solution. To distinguish the species (e.g.,  $S$ ) from its concentration value, the  $[ ]$  notation frequently is used, and so the concentration of  $S$  will be denoted  $[S]$ . It is important to keep clear the differences between this notation and the use of the brackets to denote MATLAB matrices.

We now consider a case where the enzyme and substrate are mixed initially in a flask and some means of measuring the instantaneous substrate concentration and product production rate are available (as an approximation, one may accurately measure the initial substrate concentration and then estimate the initial reaction rate from data taken shortly after the reaction proceeds). The overall system is homogeneous - the enzyme is not fixed to a solid substrate, but is well-mixed with the reactants. In this experiment, the initial enzyme concentration  $[E_0]$  also is known.

A critical assumption in analyzing the kinetics of the process is that the reaction between the substrate and enzyme is extremely fast relative to the reverse reaction or the product-forming reaction; under this assumption the concentration of the  $ES$  complex is at quasi-steady state with respect to substrate concentration  $[S]$ :

$$k_1[E][S] - k_{-1}[ES] - k_2[ES] = 0$$

or

$$[ES] = \frac{k_1[E][S]}{k_{-1} + k_2} = \frac{[E][S]}{K_m}$$

with

$$K_m = \frac{k_{-1} + k_2}{k_1}.$$

Determining a total molar balance on the enzyme:

$$[E_0] = [E] + [ES] \quad \text{so} \quad [E] = [E_0] - [ES]$$

and

$$[ES] = \frac{([E_0] - [ES])[S]}{K_m} \implies [ES] = \frac{[E_0][S]}{K_m + [S]}.$$

So the production rate of species  $P$  (mol/time or mass/time) is

$$R_p = k_2[ES] = \frac{k_2[E_0][S]}{K_m + [S]} = \frac{V_m[S]}{K_m + [S]}.$$

Having found

$$R_p = \frac{V_m[S]}{K_m + [S]} \tag{7.1}$$

we can invert the equation to find

$$\frac{1}{R_p} = \frac{K_m}{V_m} \frac{1}{[S]} + \frac{1}{V_m}$$

and so the traditional method to determining the rate parameters  $V_m$  and  $K_m$  is to create a plot of  $1/R_p$  versus  $1/[S]$  in the form of a Lineweaver-Burk plot, where the intercept is used to determine  $V_m$ , and the slope is used to subsequently determine  $K_m$ .

[S]	(a)	(b)	(c)	(d)
12.0	13.3	17.5	15.4	15.0
1.5	6.4	6.45	6.23	6.0
6.0	13.3	12.2	13.1	12.0
15.0	16.2	14.9	15.2	15.0
0.5	2.95	2.92	3.01	3.0
8.0	14.4	13.2	14.4	15.0
2.0	7.91	7.67	7.52	7.0
20.0	15.9	15.4	13.7	15.0
10.0	15.0	13.8	15.1	15.0
5.0	11.2	11.5	12.2	12.0
1.0	4.94	4.97	4.67	5.0
3.0	8.45	9.39	9.54	10.0

Table 7.1: Enzyme kinetic data taken from [7].

Given a set of data corresponding to measured values of  $R_p$  versus  $[S]$ , we can use the least squares technique to fit a straight line  $\hat{y} = b_0 + b_1x$  through the plotted data  $y = 1/R_p$  versus  $x = 1/[S]$ , determine  $V_m$  from  $b_0$ , and then  $K_m$  from  $b_1$  and  $V_m$ . However, because  $V_m$  is used to compute  $K_m$ , any error in determining the former is propagated in computing the latter. One approach that may result in a more balanced distribution of error among the identified coefficients is to rearrange (7.1) in the following way:

$$R_p K_m - [S] V_m = -R_p [S] \quad (7.2)$$

### 7.1.1 Reaction data

We demonstrate our approach using data taken from Table 1.1 of [7] and reproduced in Table 7.1. The first column of the data represents the substrate concentrations (experiment inputs) and the remaining four columns represent four sets (a-d) of experimental data. Using (7.2), we identify a set of  $K_m$  and  $V_m$  values for each of the four data sets (a-d) and plot the results in Fig. 7.2. We see that in each case, we achieve a reasonable degree of accuracy in terms of the model fit. In particular, case (a) appears to be the best in that the residuals (the difference between the model prediction and measured values) are normally distributed and are not a function of  $R_p$ . However, we see that one data point in (b) appears questionable, there is a strong dependence of the residual on  $R_p$  in case (c), and the lack of precision in data set (d) strongly affects the residual for that case.

## 7.2 Review Problems

1. Determine the Michaelis-Menten kinetics constants  $V_m$  and  $K_m$  for the following set of data:

$$[S] = [0.138, 0.220, 0.291, 0.560, 0.766, 1.46]$$

and

$$R_p = [0.148, 0.171, 0.234, 0.324, 0.390, 0.493]$$

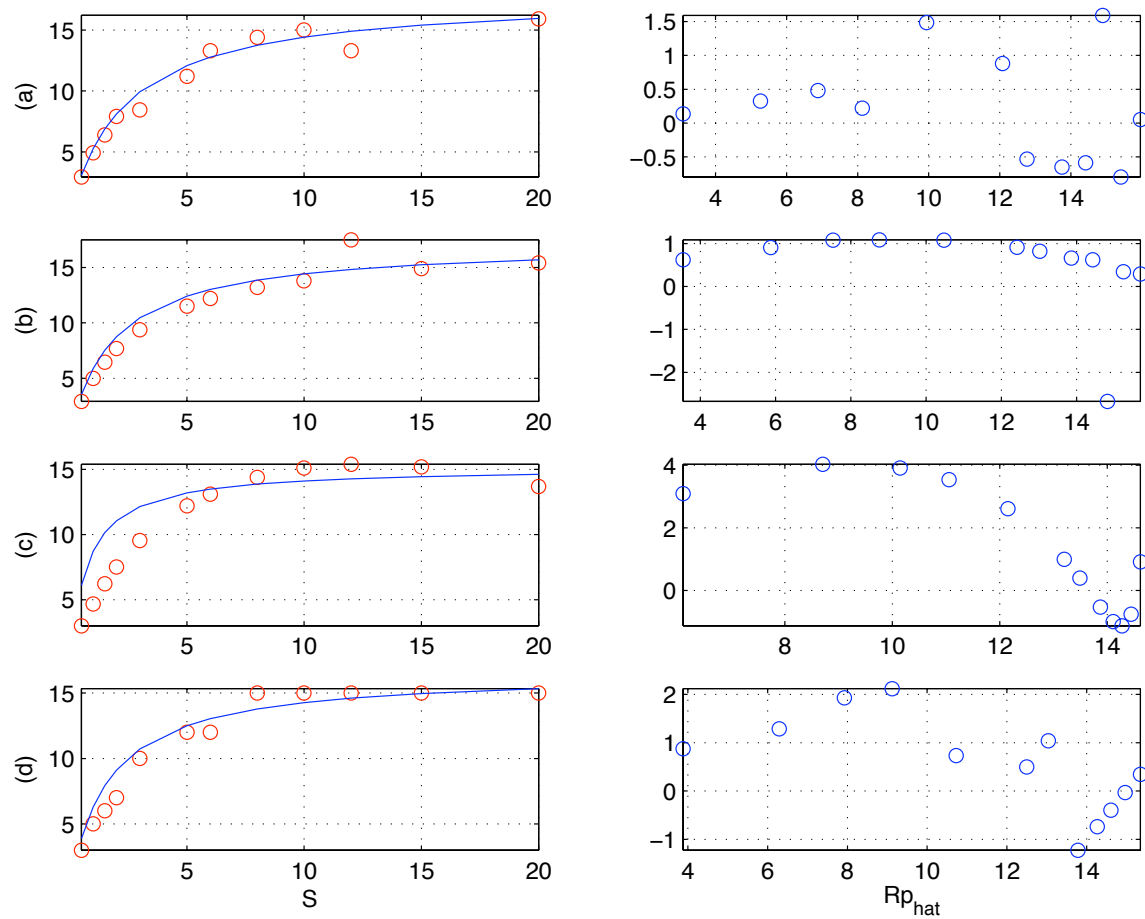
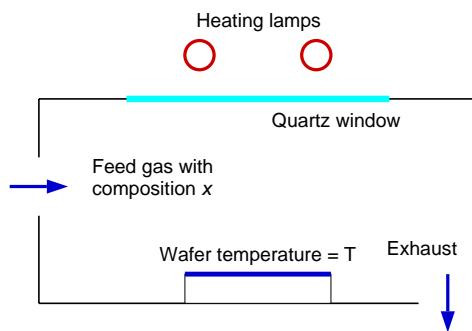


Figure 7.2: Plots of demonstration data compared to rate curves fitted by linear regression (left) and residual plots ( $\hat{R}_p - R_p$  versus  $\hat{R}_p$ ).

## Chapter 8

# Case Study: Least squares fit of CVD reactor data

It has been observed that the temperature of a lamp-heated wafer in a chemical vapor deposition reactor can be a strong function of gas temperature when the lamp power and all other factors are kept constant [5]. In the cited tungsten CVD reactor system study, the following dependency was measured:



$H_2$ mole fraction ( $x$ )	$T$ (K)
1.0	553.33
0.8	577.33
0.6	602.33
0.4	626.67
0.0	672.67

Figure 8.1: Wafer temperature as a function of gas composition;  $x$  represents  $H_2$  model fraction in a hydrogen-nitrogen gas mixture.

### 8.1 The least squares fit

We wish to fit these data to the following linear equation:

$$T = ax + b$$

using a least squares procedure. First, the equations are written out explicitly as

$$\begin{aligned} T_1 &= ax_1 + b \\ T_2 &= ax_2 + b \\ &\vdots \\ T_N &= ax_N + b \end{aligned}$$

so

$$\mathbf{T} = \mathbf{A}\mathbf{y}$$

with

$$\mathbf{A} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_N & 1 \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_N \end{bmatrix} \quad \text{and,} \quad \mathbf{y} = \begin{bmatrix} a \\ b \end{bmatrix}$$

## 8.2 Using linearsystemsolver

Given the experimentally determined wafer temperature values  $\mathbf{T}$  at the given gas composition values  $\mathbf{x}$ , we set up the coefficient array  $\mathbf{A}$  and find the coefficient values using the `linearsystemsolver` function; the experimental values and plot of the linear function fitted to these data are shown in the left side of Fig. 8.2.

```
>> x = [1; 0.8; 0.6; 0.4; 0];
>> T = [553.3; 577.3; 602.3; 626.7; 672.7];
>> A = [x, ones(size(x))];
>> coeff = linearsystemsolver(A,T)

coeff =
-119.7432
 673.5162

>> plot(x,coeff(1)*x+coeff(2),x,T,'o')
```

## 8.3 Using the myline class

We carry out these operations by using the data to define an object of `myline` class:



```

>> x = [1; 0.8; 0.6; 0.4; 0];
>> T = [553.3; 577.3; 602.3; 626.7; 672.7];
>> A = mylinelim(x,T)
mylinelim class object, with y = ax+b
a = -119.7432
b = 673.5162
limits of validity, xmin: 0 xmax: 1
>> plot(A,0,1)
>> hold on
>> plot(x,T,'o')
>> grid on, xlabel('H_2 mole fraction'), ylabel('T (K)')

```

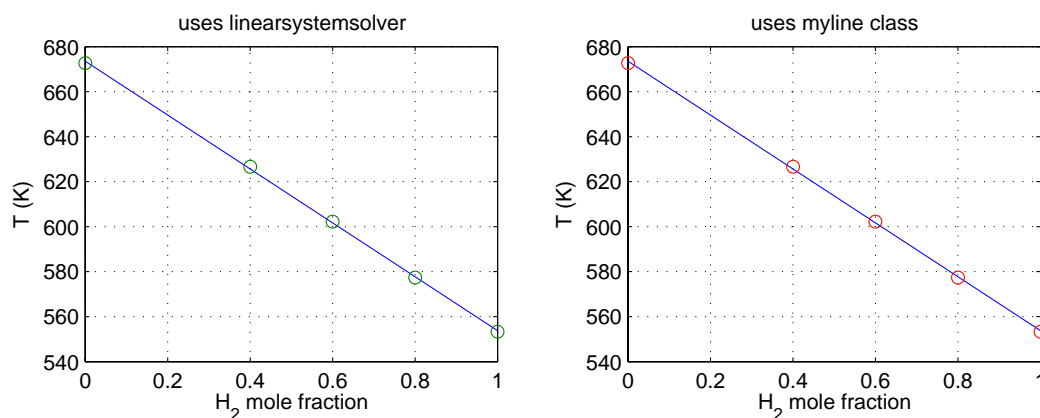


Figure 8.2: *Experimental data plotted with the linear interpolation.*

The results are plotted in the right side of Fig. 8.2, illustrating the good fit achieved using the linear interpolation. The large temperature variation with gas composition is attributed to the much higher thermal conductivity of H<sub>2</sub> relative to N<sub>2</sub>. Note that when the yval method is asked to compute a value outside the range of validity, a warning message is produced.

```

>> Ttest = yval(A,1.1)
yval: some input values are outside the limits
Ttest =
    541.7986

```

## Chapter 9

# Linear ordinary differential equations

In this chapter, we study systems that vary as a function of time, spatial position, or are otherwise *distributed* in the sense that one or more states of the system depend on a coordinate value. We focus on two common forms of this type of model: those systems that depend on the derivative of the model's state(s) as a function of time or of spatial position. Examples of the first include models with accumulation terms, and the second includes models where the transport of heat or chemical species is a function of temperature or mole fraction gradient, respectively.

### 9.1 Scalar ordinary differential equations

Consider, for example, a liquid storage tank in which the rate of liquid outflow is proportional to the height  $h$  of liquid in the tank. The tank has cross-sectional area  $A$  and has a constant volumetric input flow rate of  $q$ . A model describing the liquid height in the tank can be written as

$$A \frac{dh}{dt} = q - \alpha h \quad (9.1)$$

where  $t$  represents time and  $\alpha$  the proportionality constant defining the liquid outflow rate. We can proceed immediately to finding a *general* solution to (9.1), but normally we are more interested in finding the *particular* solution corresponding to an *initial condition*:

$$h(t = 0) = h_0. \quad (9.2)$$

Let us first consider the simpler case where there is no inflow of liquid to the tank ( $q = 0$ ); the problem reduces to

$$\frac{dh}{dt} = -\frac{\alpha}{A}h = -\gamma h \quad (9.3)$$

subject to the initial condition (9.2).

For this system with no inflow to the tank, we intuitively expect that for some initial condition the level  $h$  will drop in time, asymptotically reaching  $h \rightarrow 0$  as  $t \rightarrow \infty$ . This makes sense in many other contexts; for example, if we had a microbial population that reproduced at a rate proportional to the population  $p$

with rate constant  $b$  and died with rate constant  $d$ , the overall population balance would take the form

$$\frac{dp}{dt} = bp - dp = \lambda p.$$

Therefore, if  $\lambda < 0$ , the population ultimately would die out, if  $\lambda > 0$ , the population would grow explosively, and if  $\lambda = 0$  the population would remain constant. Known as an **eigenvalue**, the sign of  $\lambda$  determines the stability of the system (whether perturbations grow infinitely large or dissipate) and the magnitude the time constant of the system. As we will see, an eigenvalue has real physical meaning and can be a strong function of the system's parameters.

With this qualitative understanding of the expected solution in hand, we turn to solving (9.3) subject to initial condition (9.2) with the three-step procedure:

1. Assume that the form of a solution consists of the product of a time-dependent function and a constant;
2. Substitute this guess into the differential equation and determine a general solution by adjusting some of the constants of the assumed solution so that the differential equation is satisfied;
3. Compute a particular solution that satisfies the initial conditions using any remaining undefined constants.

While this may first appear to turn a simple problem into a more elaborate procedure, we will find this procedure to be applicable to all differential equation models to follow, including the numerical procedures for finding approximate solutions to nonlinear partial differential equation models.

Therefore, as step 1, we try

$$h = ce^{\lambda t} \tag{9.4}$$

as a solution, containing the two unknown constants  $\lambda$  and  $c$ ; Step 2 requires substituting this into (9.3):

$$\begin{aligned} \frac{d}{dt}(ce^{\lambda t}) &= -\gamma ce^{\lambda t} \\ \lambda ce^{\lambda t} &= -\gamma ce^{\lambda t} \\ \lambda &= -\gamma. \end{aligned}$$

Thus, the general solution satisfies the ordinary differential equation (9.3), and can be written as

$$h(t) = ce^{-\gamma t}.$$

The particular solution satisfies both the differential equation and the initial condition:

$$h_0 = ce^{-\gamma \times 0} \implies c = h_0$$

so

$$h(t) = h_0 e^{-\gamma t}.$$

### 9.1.1 A non-homogeneous problem

Before moving on to sets of linear, autonomous differential equations, let us consider the scalar differential equation and initial condition:

$$\frac{dx}{dt} = \alpha x + \beta \quad \text{subject to } x(0) = x_0 \quad (9.5)$$

which is the form of the storage tank problem with non-zero inlet feed flow rate.

This linear, scalar, time-invariant ordinary differential equation is solved easily by first transforming the nonhomogeneous problem to homogeneous form by using the variable transformation

$$\begin{aligned} y &= \alpha x + \beta \\ \frac{dy}{dt} &= \alpha \frac{dx}{dt} \\ &= \alpha [\alpha x + \beta] \\ &= \alpha y \end{aligned}$$

so the particular solution is

$$\begin{aligned} y &= y_0 e^{\alpha t} \\ \alpha x(t) + \beta &= [\alpha x_0 + \beta] e^{\alpha t} \\ x(t) &= -\frac{\beta}{\alpha} + \left[ x_0 + \frac{\beta}{\alpha} \right] e^{\alpha t} \end{aligned}$$

This solution makes physical sense, because at  $t = 0$   $x = x_0$  and as  $t \rightarrow \infty$   $x \rightarrow -\beta/\alpha$  provided  $\alpha < 0$ .

## 9.2 Sets of linear ODEs

If we now consider the problem of determining the solution to a set of coupled ordinary differential equations

$$\frac{dz}{dt} = \mathbf{A}z \quad \text{subject to } z(0) = z_0 \quad (9.6)$$

we can pose a guess for the solution as

$$z = e^{\mathbf{A}t} z_0 \quad (9.7)$$

which is similar in form to (9.4) except for the more complicated exponential function in time;  $e^{\mathbf{A}t}$  is known as the matrix exponential. Our solution begins with defining a new set of state variables  $\mathbf{q}$ ; these states can be seen as the linear combinations of the **eigenvectors** contained in  $\mathbf{U}$  [26] that produce the vector  $\mathbf{z}$ :

$$\mathbf{z} = \mathbf{U}\mathbf{q}.$$

Substituting this into (9.6):

$$\begin{aligned} \mathbf{U} \frac{d\mathbf{q}}{dt} &= \mathbf{A}\mathbf{U}\mathbf{q} \quad \text{subject to } \mathbf{q}_0 = \mathbf{U}^{-1}z_0 \\ \frac{d\mathbf{q}}{dt} &= \mathbf{U}^{-1}\mathbf{A}\mathbf{U}\mathbf{q} \end{aligned}$$

Under most circumstances we encounter, the matrix  $\mathbf{A}$  can be factored into the product

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} \quad (9.8)$$

where  $\mathbf{\Lambda}$  is a diagonal array, and so

$$\begin{aligned} \frac{d\mathbf{q}}{dt} &= \mathbf{\Lambda}\mathbf{q} \\ &= \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \mathbf{q}. \end{aligned}$$

The last equation is so important, we write it out explicitly for this  $2 \times 2$  case and will return to it in a later discussion:

$$\begin{aligned} dq_1/dt &= \lambda_1 q_1; \\ dq_2/dt &= \lambda_2 q_2. \end{aligned}$$

We have solved this problem before (see equation 9.4). Thus, we can write a solution for the decoupled linear equations in matrix form:

$$\begin{aligned} \mathbf{q} &= \begin{bmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{bmatrix} \mathbf{q}_0 \\ \mathbf{U}^{-1}\mathbf{z} &= e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0 \\ \mathbf{z} &= \mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0. \end{aligned}$$

Therefore, we see the matrix exponential is defined as

$$e^{\mathbf{A}t} = \mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}.$$

Substituting this solution into the set of differential equations (9.6) validates the particular solution,

$$\begin{aligned} \frac{d}{dt} (\mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0) &= \mathbf{A}\mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0 \\ \mathbf{U}\mathbf{\Lambda}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0 &= \mathbf{A}\mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0 \\ \mathbf{U}\mathbf{\Lambda} &= \mathbf{A}\mathbf{U} \end{aligned}$$

so indeed (9.8) holds true and

$$\mathbf{A}\mathbf{u}_1 = \lambda_1 \mathbf{u}_1, \quad \mathbf{A}\mathbf{u}_2 = \lambda_2 \mathbf{u}_2$$

where  $\mathbf{u}_i$  are the column vectors of  $\mathbf{U}$ .

### 9.2.1 Computing eigenvalues/vectors

As part of our procedure for computing a general solution to the set of ordinary differential equations, we must solve the eigenvalue problem

$$\mathbf{A}\mathbf{u} = \lambda\mathbf{u} \quad \text{or} \quad (\mathbf{A} - \lambda\mathbf{I})\mathbf{u} = \mathbf{0}.$$

Writing this out for a  $2 \times 2$  system:

$$\begin{bmatrix} A_{1,1} - \lambda & A_{1,2} \\ A_{2,1} & A_{2,2} - \lambda \end{bmatrix} \mathbf{u} = \mathbf{0} \quad (9.9)$$

Given the coefficient array  $\mathbf{A}$ , we can use our linear equation solving tools to determine  $\lambda_1$  and  $\lambda_2$  that correspond to *non-trivial*  $\mathbf{u}_1$  and  $\mathbf{u}_2$ , respectively. Because of the constraint that we find only non-trivial  $\mathbf{u}_i$ , the coefficient array in (9.9) must be singular; using one step of the forward elimination procedure gives

$$\begin{bmatrix} A_{1,1} - \lambda & A_{1,2} \\ 0 & A_{2,2} - \lambda - \frac{A_{1,2}A_{2,1}}{A_{1,1} - \lambda} \end{bmatrix} \mathbf{u} = \mathbf{0}$$

and so setting

$$A_{2,2} - \lambda - \frac{A_{1,2}A_{2,1}}{A_{1,1} - \lambda} = 0$$

gives a quadratic *characteristic equation* for this  $2 \times 2$  system:

$$\lambda^2 - (A_{1,1} + A_{2,2})\lambda + A_{1,1}A_{2,2} - A_{1,2}A_{2,1} = 0. \quad (9.10)$$

Using the definitions of the trace and determinant of the  $\mathbf{A}$  matrix

$$tr(\mathbf{A}) = A_{1,1} + A_{2,2} \quad \text{and} \quad det(\mathbf{A}) = A_{1,1}A_{2,2} - A_{1,2}A_{2,1}$$

we see that the eigenvalues switch from being real and distinct to complex when the discriminant

$$\Delta = tr(\mathbf{A})^2 - 4det(\mathbf{A})$$

vanishes. The eigenvalues themselves for this  $2 \times 2$  system are found to be

$$\lambda_{1,2} = \frac{tr(\mathbf{A}) \pm \sqrt{tr(\mathbf{A})^2 - 4det(\mathbf{A})}}{2}.$$

Substituting the eigenvalues  $\lambda_i$  into (9.9), we can determine the corresponding eigenvectors  $\mathbf{u}_i$ :

$$\begin{aligned} \begin{bmatrix} A_{1,1} - \lambda_1 & A_{1,2} \\ A_{2,1} & A_{2,2} - \lambda_1 \end{bmatrix} \mathbf{u}_1 &= \mathbf{0} \\ \mathbf{u}_1 &= c_1 \begin{bmatrix} U_{1,1} \\ U_{2,1} \end{bmatrix} \quad \text{corresponding to } \lambda_1 \\ \begin{bmatrix} A_{1,1} - \lambda_2 & A_{1,2} \\ A_{2,1} & A_{2,2} - \lambda_2 \end{bmatrix} \mathbf{u}_2 &= \mathbf{0} \\ \mathbf{u}_2 &= c_2 \begin{bmatrix} U_{1,2} \\ U_{2,2} \end{bmatrix} \quad \text{corresponding to } \lambda_2 \end{aligned}$$

The constants  $c_1, c_2$  can be anything including zero: we can write the results in array form:

$$\mathbf{V} = \begin{bmatrix} U_{1,1} & U_{1,2} \\ U_{2,1} & U_{2,2} \end{bmatrix} \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix} = \mathbf{U}\mathbf{C}.$$

and substituting these eigenvectors into the solution using  $\mathbf{q} = \mathbf{V}^{-1}\mathbf{z}$ ,

$$\begin{aligned} \mathbf{q} &= e^{\mathbf{\Lambda}t} \mathbf{q}_0 \\ \mathbf{z} &= \mathbf{U}\mathbf{C}e^{\mathbf{\Lambda}t}\mathbf{V}^{-1}\mathbf{z}_0 \\ &= \mathbf{U}\mathbf{C}e^{\mathbf{\Lambda}t}\mathbf{C}^{-1}\mathbf{U}^{-1}\mathbf{z}_0 \\ &= \mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0 \quad \text{because } \mathbf{C} \text{ and } e^{\mathbf{\Lambda}t} \text{ are diagonal arrays.} \end{aligned}$$

Therefore, the solution does not depend on the values  $c_1$  and  $c_2$ .

### 9.2.2 Eigenvectors and the phase space

Using our view that the vector of initial conditions  $\mathbf{z}_0$  can be written in terms of a linear combination of the eigenvectors  $\mathbf{U}$ :

$$\mathbf{U}\mathbf{q} = \mathbf{z}_0$$

we see that the solution approach we have described also gives a clear view into the modal structure of the particular solution:

1. At  $t = 0$ , we decompose the initial condition column vector into some linear combination of the (column) eigenvectors (i.e., we *project* the initial condition vector onto the eigenvectors):

$$\mathbf{z}_0 = q_1\mathbf{u}_1 + q_2\mathbf{u}_2;$$

2. The solution then evolves as the lengths of the individual vectors  $q_j\mathbf{u}_j$  shrink or grow, independently of one-another and at a rate depending on the corresponding eigenvalues  $\lambda_1$  and  $\lambda_2$ :

$$\mathbf{z} = q_1 e^{\lambda_1 t} \mathbf{u}_1 + q_2 e^{\lambda_2 t} \mathbf{u}_2. \quad (9.11)$$

We see that if the initial condition falls directly on one of the eigenvectors, it stays on that eigenvector for all time.

For the  $2 \times 2$  system, we write the solution out explicitly as

$$\begin{bmatrix} x \\ y \end{bmatrix} = q_1 \begin{bmatrix} U_{1,1} \\ U_{2,1} \end{bmatrix} e^{\lambda_1 t} + q_2 \begin{bmatrix} U_{1,2} \\ U_{2,2} \end{bmatrix} e^{\lambda_2 t}$$

Consider plotting the solution  $\mathbf{z}(t)$  as a curve in the  $(x, y)$  plane (the *phase plane*); plots of these type result in curves that begin at the initial condition and are parameterized by time.

We can observe the manner by which trajectories in the phase plane are governed by the magnitude of the eigenvalues and location of the eigenvectors by computing these quantities and then plotting the eigenvectors together with plots of the linear ODE solution computed using randomly chosen initial conditions. This is done in the MATLAB script below for

$$\mathbf{A} = \begin{bmatrix} -2 & 1 \\ 1 & -1.5 \end{bmatrix}$$

and results are displayed in Fig. 9.1. For this example, we find that both eigenvalues are real, distinct, and less than zero; this corresponds to a **stable node**. The figure illustrates how the trajectories, as they approach the origin, are contracted more strongly along the eigenvector direction corresponding to the larger in magnitude eigenvalue.

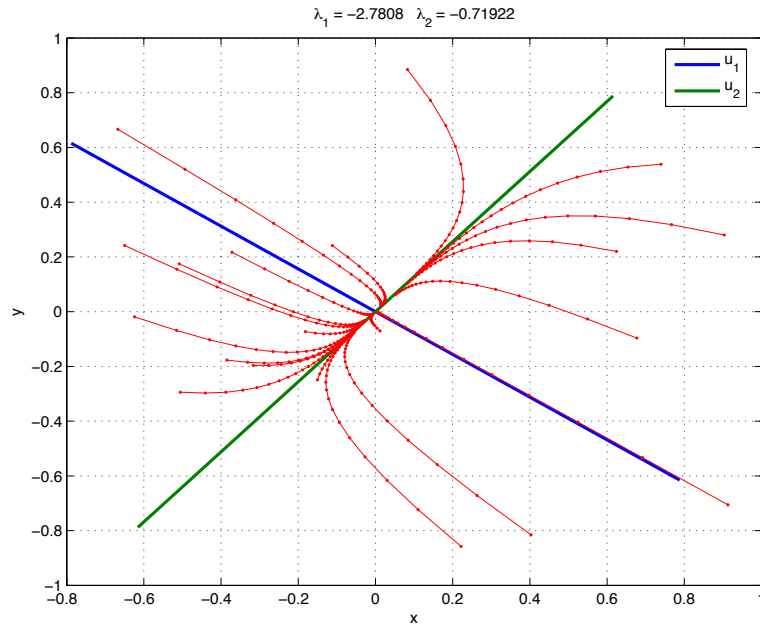


Figure 9.1: A demonstration of using `lodesolver.m`, showing how the trajectories in phase space are organized by the eigenvectors of the coefficient matrix.

```
A = [-2 1; 1 -1.5]
[U,S] = eig(A)

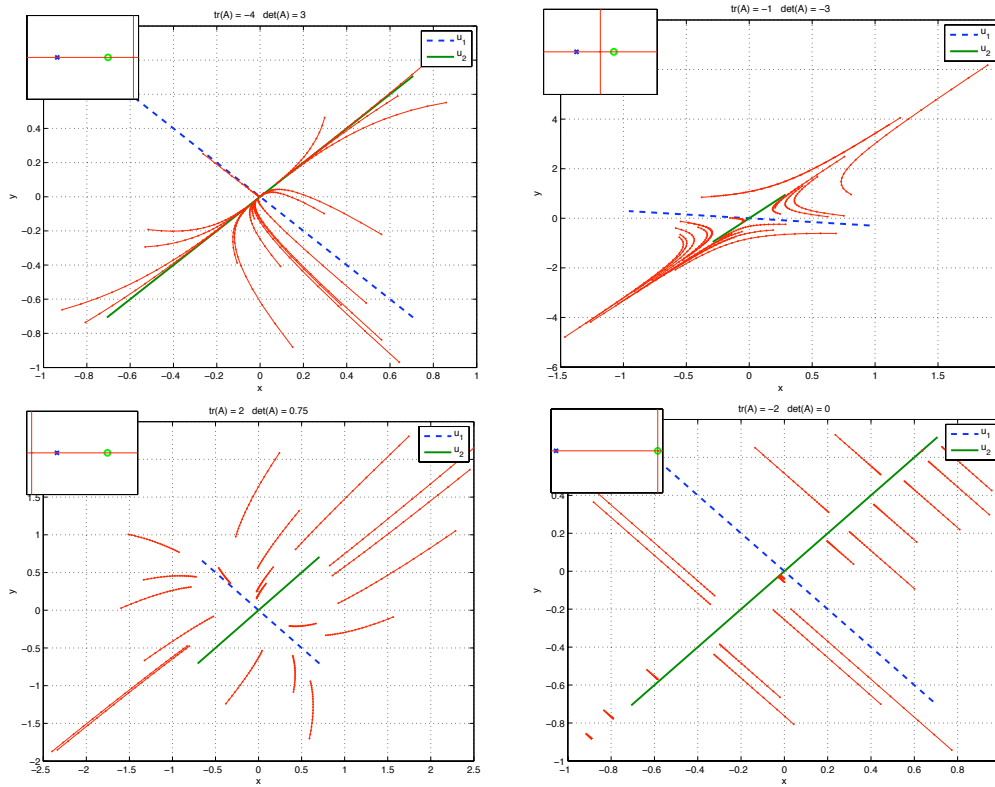
plot([-U(1,1) U(1,1)],[-U(2,1) U(2,1)], ...
     [-U(1,2) U(1,2)],[-U(2,2) U(2,2)], 'Linewidth',2)
legend('u_1','u_2')
xlabel('x'), ylabel('y')
grid on
title(['\lambda_1 = ',num2str(S(1,1)), '   \lambda_2 = ',num2str(S(2,2))])

t = [0:0.1:2];
hold on
for i = 1:20
    z0 = 2*rand(2,1)-[1;1];
    z = lodesolver(A,t,z0);
    plot(z(1,:),z(2:,:), 'r.-')
end
hold off
```

### 9.2.3 Real and distinct eigenvalues ( $\Delta > 0$ )

When the discriminant  $\Delta$  is positive, we have the following possibilities:



Figure 9.2: Phase portraits for cases where  $\Delta > 0$ .

1.  $\text{tr}(\mathbf{A}) < 0$  and  $\det(\mathbf{A}) > 0$  results in two negative eigenvalues and corresponds to a stable node;
2.  $\text{tr}(\mathbf{A}) < 0$  and  $\det(\mathbf{A}) < 0$  results in eigenvalues of opposite signs and corresponds to a saddle point;  $\text{tr}(\mathbf{A}) > 0$  and  $\det(\mathbf{A}) < 0$  also results in eigenvalues of opposite signs and corresponds to a saddle point
3.  $\text{tr}(\mathbf{A}) > 0$  and  $\det(\mathbf{A}) > 0$  results in two positive eigenvalues and corresponds to an unstable node
4.  $\det(\mathbf{A}) = 0$  results in one eigenvalue with value 0 and the other positive or negative depending on the sign of  $\text{tr}(\mathbf{A})$ ; when the latter is negative the system is neutrally stable, and when positive, the system is unstable.

Examples of each case can be found in Fig. 9.2 and a summary of the expected behavior is shown in Fig. 9.3.

### 9.2.4 Repeated eigenvalues ( $\Delta = 0$ )

The eigenvalues are real and identical when  $\Delta = 0$  for the  $2 \times 2$  system. Because only one eigenvector  $\mathbf{u}_1$  can be computed in this situation, but initial conditions can be defined anywhere in the phase plane,

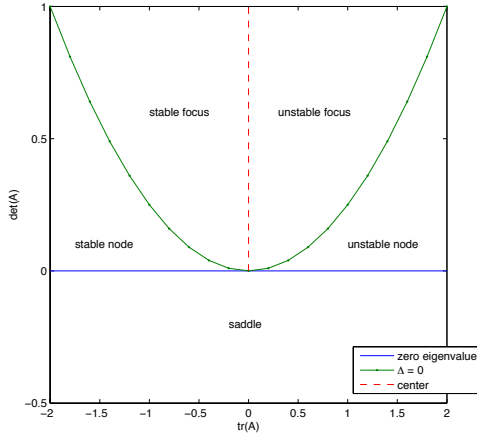


Figure 9.3: Expected dynamic behavior as a function of  $\text{tr}(\mathbf{A})$  and  $\text{det}(\mathbf{A})$ .

another linearly independent general solution must be found for the set of ODEs. We begin by rewriting (9.11) in a form that is valid for solutions to a set of  $N$  linear ODEs with distinct eigenvalues  $\lambda_n$

$$\mathbf{z} = \sum_{n=1}^N q_n e^{\lambda_n t} \mathbf{u}_n$$

where the initial condition  $\mathbf{z}_0$  is used to determine the  $q_n$  using  $\mathbf{q} = \mathbf{U}^{-1} \mathbf{z}_0$ . For the case  $\lambda_N = \lambda_{N-1}$  let's try

$$\mathbf{z} = \sum_{n=1}^{N-1} q_n e^{\lambda_n t} \mathbf{u}_n + q_N (te^{\lambda_N t} \mathbf{u}_{N-1} + e^{\lambda_N t} \mathbf{w}) \quad (9.12)$$

as a solution where, as before,

$$(\mathbf{A} - \lambda_n \mathbf{I}) \mathbf{u}_n = \mathbf{0} \quad n = 1, 2, \dots, N-1 \quad (9.13)$$

but now

$$(\mathbf{A} - \lambda_N \mathbf{I}) \mathbf{w} = \mathbf{u}_{N-1}. \quad (9.14)$$

To prove this is the true solution, we substitute (9.12) into (9.6) to find

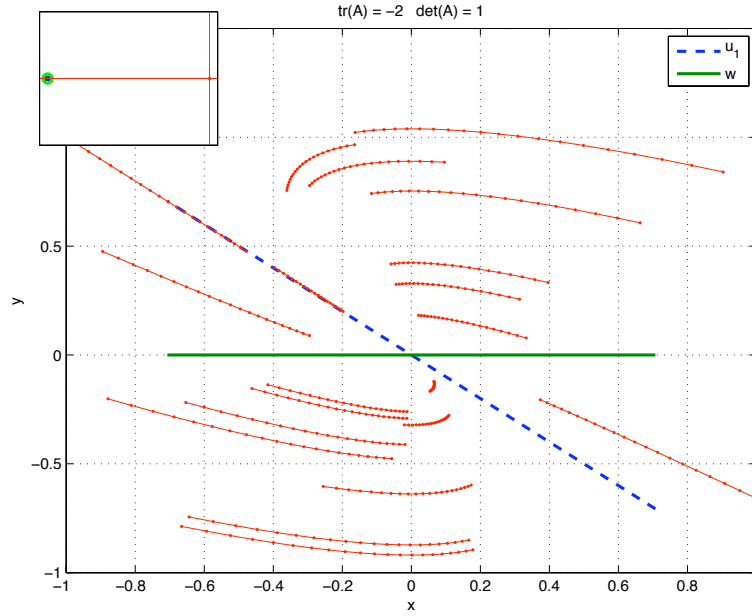
$$\begin{aligned} & \sum_{n=1}^{N-1} q_n \lambda_n e^{\lambda_n t} \mathbf{u}_n + q_N e^{\lambda_N t} \mathbf{u}_{N-1} + q_N t \lambda_N e^{\lambda_N t} \mathbf{u}_{N-1} + q_N \lambda_N e^{\lambda_N t} \mathbf{w} \\ &= \sum_{n=1}^{N-1} q_n e^{\lambda_n t} \mathbf{A} \mathbf{u}_n + q_N t e^{\lambda_N t} \mathbf{A} \mathbf{u}_{N-1} + q_N e^{\lambda_N t} \mathbf{A} \mathbf{w}. \end{aligned}$$

From the equation above, we conclude that (9.12) is indeed a solution to (9.6) for the case of two repeated eigenvalues because (9.13) implies both

$$\sum_{n=1}^{N-1} q_n e^{\lambda_n t} \mathbf{A} \mathbf{u}_n - q_n \lambda_n e^{\lambda_n t} \mathbf{u}_n = 0$$

and

$$q_N t e^{\lambda_N t} \mathbf{A} \mathbf{u}_{N-1} - q_N t \lambda_N e^{\lambda_N t} \mathbf{u}_{N-1} = 0$$

Figure 9.4: A phase portrait for the case  $\Delta = 0$ .

because  $\lambda_N = \lambda_{N-1}$  in the second equation. This leaves

$$q_N e^{\lambda_N t} \mathbf{A} \mathbf{w} - q_N \lambda_N e^{\lambda_N t} \mathbf{w} = q_N e^{\lambda_N t} \mathbf{u}_{N-1}$$

which is true if (9.14) is satisfied.

A representative case can be seen in Fig. 9.4 for

$$\mathbf{A} = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix}.$$

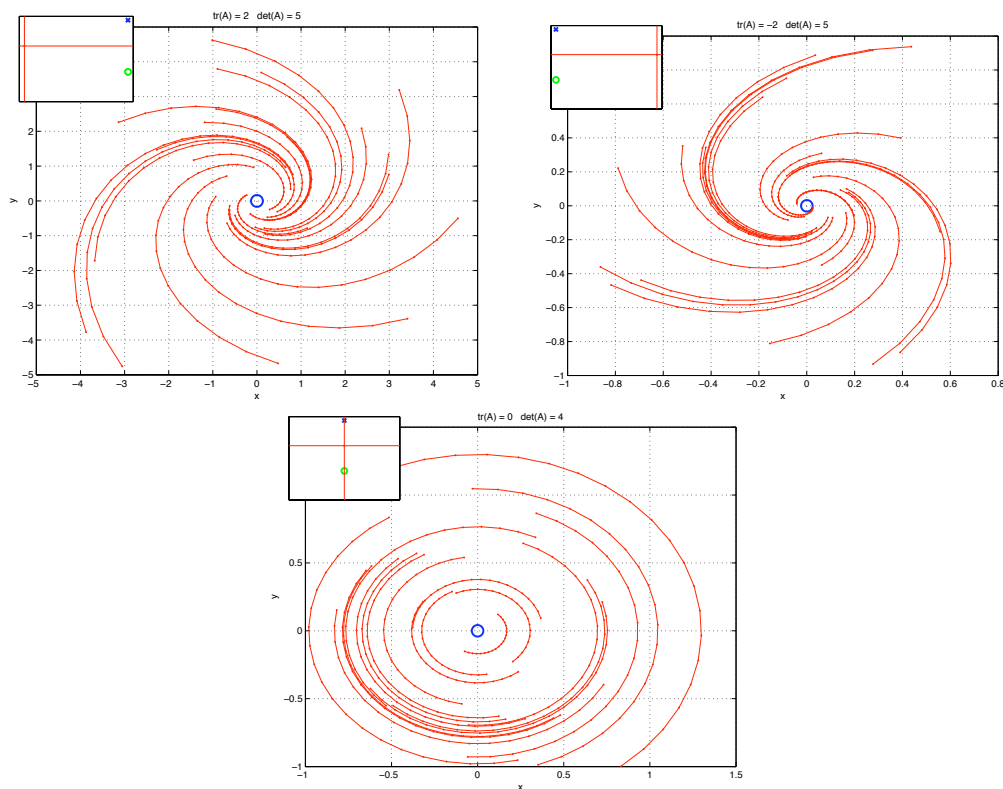
For this case we find

$$\lambda_1 = \lambda_2 = -1 \quad \text{with} \quad \mathbf{u}_1 = \mathbf{u}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Now using (9.14) we find

$$\begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \mathbf{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \text{so clearly} \quad \mathbf{w} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

is a valid solution for  $\mathbf{w}$ . The eigenvector  $\mathbf{u}_1$  and  $\mathbf{w}$ , together with representative time-dependent trajectories for this stable system are plotted in Fig. 9.4. To check this solution, we observe that  $dx/dt = 0$  along the line  $y = -2x$ ; this indicates that trajectories in time will switch their left-to-right direction along this line. Furthermore, we observe that  $dy/dt = 0$  when  $x = 0$  and so trajectories switch in the vertical direction along this axis. We see evidence for both behaviors in Fig. 9.4.

Figure 9.5: Phase portraits for cases where  $\Delta < 0$ .

### 9.2.5 Complex eigenvalues ( $\Delta < 0$ )

For cases of complex eigenvalues  $\lambda_j = \alpha + i\beta$ ,  $\lambda_{j+1} = \alpha - i\beta$  and the associated eigenvectors  $\mathbf{u}_j = \mathbf{u}^R + i\mathbf{u}^I$ ,  $\mathbf{u}_{j+1} = \mathbf{u}^R - i\mathbf{u}^I$ , the solution can be written in the form

$$\mathbf{z} = \cdots + q_j e^{\alpha t} [\mathbf{u}^R \cos(\beta t) - \mathbf{u}^I \sin(\beta t)] + q_{j+1} e^{\alpha t} [\mathbf{u}^R \cos(\beta t) + \mathbf{u}^I \sin(\beta t)] + \cdots$$

and so the solution has an oscillatory component with frequency  $\beta$  and an amplitude growth rate governed by  $\alpha$ ; the eigenspace associated with this oscillatory mode is defined by  $\text{span}\{\mathbf{u}^R, \mathbf{u}^I\}$ . See [15] for more information and Fig. 9.5 for representative phase portraits.

A simple way to determine the direction of rotation of the trajectories around the origin is to consider the system when  $x = 0$ ; the differential equation for  $x$  at this point becomes

$$\frac{dx}{dt} = A_{1,2}y$$

and so the system rotates in the clockwise direction when  $A_{1,2} > 0$  and counterclockwise when  $A_{1,2} < 0$ .

### 9.2.6 Symmetric systems

If the coefficient array  $\mathbf{A}$  for our  $2 \times 2$  system is *symmetric* and we compute the inner product of the two eigenvectors

$$\mathbf{u}_1^T \mathbf{u}_2 = 1 - 1 = 0 \implies \mathbf{u}_1 \text{ and } \mathbf{u}_2 \text{ are orthogonal.}$$

This is, in fact, one of the general properties of symmetric systems. Proof follows [16]; if we consider the two eigenvalue problems where  $\lambda_1$  and  $\lambda_2$  are distinct eigenvalues of  $\mathbf{A} = \mathbf{A}^T$ :

$$\mathbf{A}\mathbf{u}_1 = \lambda_1\mathbf{u}_1 \quad \mathbf{A}\mathbf{u}_2 = \lambda_2\mathbf{u}_2 \quad (9.15)$$

so

$$\begin{aligned} (\mathbf{A}\mathbf{u}_1)^T &= \lambda_1\mathbf{u}_1^T \\ \mathbf{u}_1^T \mathbf{A} &= \lambda_1\mathbf{u}_1^T \\ \mathbf{u}_1^T \mathbf{A}\mathbf{u}_2 &= \lambda_1\mathbf{u}_1^T \mathbf{u}_2 \end{aligned} \quad (9.16)$$

where the last equation was obtained from postmultiplying the second by  $\mathbf{u}_2$ . Premultiplying the second equation of (9.15) by  $\mathbf{u}_1^T$  we find

$$\mathbf{u}_1^T \mathbf{A}\mathbf{u}_2 = \lambda_2\mathbf{u}_1^T \mathbf{u}_2$$

and subtracting this from (9.16) gives

$$0 = (\lambda_1 - \lambda_2)\mathbf{u}_1^T \mathbf{u}_2$$

which implies that  $\mathbf{u}_1$  and  $\mathbf{u}_2$  are *orthogonal* for  $\lambda_1 \neq \lambda_2$ :

$$\mathbf{u}_1 \cdot \mathbf{u}_2 = 0.$$

Another important property of the eigenvectors generated from symmetric coefficient arrays  $\mathbf{A}$  is demonstrated by first writing the array of column eigenvectors as

$$\mathbf{U} = [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \dots \quad \mathbf{u}_N]$$

and so

$$\begin{aligned} \mathbf{U}^T \mathbf{U} &= \begin{bmatrix} \mathbf{u}_1 \cdot \mathbf{u}_1 & \mathbf{u}_1 \cdot \mathbf{u}_2 & \mathbf{u}_1 \cdot \mathbf{u}_3 & & \\ \mathbf{u}_2 \cdot \mathbf{u}_1 & \mathbf{u}_2 \cdot \mathbf{u}_2 & \mathbf{u}_2 \cdot \mathbf{u}_3 & & \\ & & \ddots & & \\ & & & \mathbf{u}_N \cdot \mathbf{u}_N \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{u}_1 \cdot \mathbf{u}_1 & 0 & 0 & & \\ 0 & \mathbf{u}_2 \cdot \mathbf{u}_2 & 0 & & \\ & & \ddots & & \\ & & & \mathbf{u}_N \cdot \mathbf{u}_N \end{bmatrix}. \end{aligned}$$

If  $\mathbf{u}_j \cdot \mathbf{u}_j = 1$  for  $j = 1, \dots, N$ ,

$$\mathbf{U}^T \mathbf{U} = \mathbf{I}.$$

If the last condition ( $\mathbf{u}_j \cdot \mathbf{u}_j = 1$ ) holds, the eigenvectors form an *orthonormal* set of vectors and

$$\mathbf{U}^T = \mathbf{U}^{-1}.$$

This simplifies evaluating the vector of coefficients  $\mathbf{q}$  because

$$\mathbf{q} = \mathbf{U}^T \mathbf{z}_0$$

and so the initial conditions are projected directly onto the eigenvectors – in other words, we can calculate the  $q_j$  as being the lengths along the  $\mathbf{u}_j$  directly.

### 9.2.7 Unsymmetric systems

In cases where the matrix  $\mathbf{A}$  is unsymmetric ( $\mathbf{A} \neq \mathbf{A}^T$ ), if we compute the eigenvalues of  $\mathbf{A}$  and  $\mathbf{A}^T$ , we find that the eigenvalues are identical. If the eigenvalues  $\lambda_1$  and  $\lambda_2$  are distinct,

$$\mathbf{A}\mathbf{u}_1 = \lambda_1\mathbf{u}_1 \quad \mathbf{A}^T\mathbf{v}_2 = \lambda_2\mathbf{v}_2 \quad (9.17)$$

so

$$\begin{aligned} (\mathbf{A}\mathbf{u}_1)^T &= \lambda_1\mathbf{u}_1^T \\ \mathbf{u}_1^T\mathbf{A}^T &= \lambda_1\mathbf{u}_1^T \\ \mathbf{u}_1^T\mathbf{A}^T\mathbf{v}_2 &= \lambda_1\mathbf{u}_1^T\mathbf{v}_2 \end{aligned} \quad (9.18)$$

where the last equation was obtained from postmultiplying the second by  $\mathbf{v}_2$ . Premultiplying the second equation of (9.17) by  $\mathbf{u}_1^T$  we find

$$\mathbf{u}_1^T\mathbf{A}^T\mathbf{v}_2 = \lambda_2\mathbf{u}_1^T\mathbf{v}_2$$

and subtracting this from (9.18) gives

$$0 = (\lambda_1 - \lambda_2)\mathbf{u}_1^T\mathbf{v}_2$$

which implies that  $\mathbf{u}_1$  and  $\mathbf{v}_2$  are orthogonal for  $\lambda_1 \neq \lambda_2$ :

$$\mathbf{u}_1 \cdot \mathbf{v}_2 = 0.$$

This means the factorization which diagonalizes  $\mathbf{A}$  is

$$\mathbf{U}^{-1}\mathbf{A}\mathbf{U} = \mathbf{V}^T\mathbf{A}\mathbf{U} = \mathbf{\Lambda}.$$

## 9.3 Summary of solutions to sets of linear ODEs

Given the general set of  $N$  independent linear equations

$$\frac{d\mathbf{z}}{dt} = \mathbf{A}\mathbf{z}$$

subject to initial conditions  $\mathbf{z}(t=0) = \mathbf{z}_0$ , we can write the solution as

$$\begin{aligned} \mathbf{z}(t) &= \mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1}\mathbf{z}_0 && \text{for any } \mathbf{A} \\ \mathbf{z}(t) &= \mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^T\mathbf{z}_0 && \text{for symmetric } \mathbf{A} \end{aligned}$$

where  $\mathbf{\Lambda}$  is a diagonal array of eigenvalues and  $\mathbf{U}$  is the array of associated eigenvectors.

Solutions to the *nonhomogeneous* system

$$\frac{dz}{dt} = \mathbf{A}z - \mathbf{b} \quad (9.19)$$

where  $\mathbf{b}$  is a vector of constants the same size as  $\mathbf{z}$ , subject to initial conditions  $\mathbf{z}(t = 0) = \mathbf{z}_0$ , can be computed using a change of variables. Defining  $\mathbf{s} = \mathbf{A}z - \mathbf{b}$  and substituting it into (9.19), we find a homogeneous set of ordinary differential equations subject to the transformed initial conditions  $\mathbf{s}_0$ . Solving this problem and transforming back to  $\mathbf{z}$ , the solution, after several simplifying steps, can be written as

$$\begin{aligned} \mathbf{z}(t) &= \mathbf{U}e^{\mathbf{A}t}\mathbf{U}^{-1}\mathbf{z}_0 - \mathbf{A}^{-1}\left[\mathbf{U}e^{\mathbf{A}t}\mathbf{U}^{-1} - \mathbf{I}\right]\mathbf{b} \quad \text{for any } \mathbf{A} \\ &= \mathbf{U}e^{\mathbf{A}t}\mathbf{U}^T\mathbf{z}_0 - \mathbf{A}^{-1}\left[\mathbf{U}e^{\mathbf{A}t}\mathbf{U}^T - \mathbf{I}\right]\mathbf{b} \quad \text{for symmetric } \mathbf{A} \end{aligned}$$

where  $\mathbf{I}$  is the identity matrix. Note that the steady state solution can be computed as  $\mathbf{z}_{stst} = \mathbf{A}^{-1}\mathbf{b}$ .

## 9.4 Boundary-value problems

We now discuss an important class of linear ordinary differential equations that arise in many chemical engineering modeling problems. Consider, for example, the problem of computing the steady state, but spatially dependent temperature profile  $T(\xi)$ , in a uniform, one-dimensional material defined over  $0 \leq \xi \leq 1$ . Furthermore, we define the temperature at the endpoints as  $T(\xi = 0) = 1$  and  $T(\xi = 1) = 0$ . Later in this test, we will shown that this situation can be described by the nonhomogeneous, second-order *boundary-value problem* (BVP):

$$\frac{d^2 T}{d\xi^2} = 0 \quad (9.20)$$

subject to the *boundary conditions*

$$\begin{aligned} T &= 1 \quad \text{at} \quad \xi = 0 \\ T &= 0 \quad \text{at} \quad \xi = 1 \end{aligned}$$

An important distinction between this type of problem and the ones studied previously in this chapter is that the conditions used to define the particular solution are not all defined at one point along the coordinate  $\xi$ , but are defined at the boundaries of the region of interest, hence the name BVP.

Clearly we can solve (9.20) by directly integrating it twice with respect to  $\xi$  to obtain the general solution

$$T(\xi) = a\xi + b \quad (9.21)$$

where  $a$  and  $b$  are the constants of integration. We then can use the two boundary conditions to determine values for  $a$  and  $b$  to find the particular solution:

$$T(\xi) = 1 - \xi.$$

The resulting linear profile makes sense in that an assumption implicit to (9.20) is that Fourier's Law holds, necessitating a constant temperature gradient for the steady state when there are no internal heat generation terms.

### 9.4.1 Where are the eigenvalues?

After our substantial previous efforts at computing eigenvalues and eigenvectors, a natural question to ask is where do these concepts appear in the solution of this BVP. We start our explanation by splitting (9.20) into the following two first-order ODEs:

$$\begin{aligned}\frac{dT}{d\xi} &= y \\ \frac{dy}{d\xi} &= 0\end{aligned}$$

and so

$$\frac{d}{d\xi} \begin{bmatrix} T \\ y \end{bmatrix} = \frac{dz}{d\xi} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} z = \mathbf{A}z.$$

We can immediately observe that  $\det(\mathbf{A}) = 0$  and  $\text{tr}(\mathbf{A}) = 0$  indicating that

$$\lambda_1 = \lambda_2 = 0 \quad \text{and} \quad \mathbf{u}_1 = \mathbf{u}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

We now use (9.14) to find  $\mathbf{w}$ :

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{w} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{so} \quad \mathbf{w} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and the general solution is

$$\begin{bmatrix} T \\ y \end{bmatrix} = q_1 e^0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + q_2 \left( \xi e^0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + e^0 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right).$$

Further simplification shows that the temperature profile is

$$T(\xi) = q_1 + q_2 \xi$$

which is exactly the same as the solution (9.21) computed by direct integration of the original BVP.

### 9.4.2 A nonhomogeneous BVP

If we now consider the same problem of heat transfer in one-dimension, but add a constant heat source term along the length of the spatial dimension, we obtain the BVP

$$\frac{d^2 T}{d\xi^2} + \gamma = 0$$

subject to the same boundary conditions as before

$$\begin{aligned}T &= 1 \quad \text{at} \quad \xi = 0; \\ T &= 0 \quad \text{at} \quad \xi = 1.\end{aligned}$$

Again, directly integrating twice with respect to  $\xi$  and then applying the boundary conditions, we find

$$T(\xi) = 1 - \frac{1}{2}\gamma\xi^2 + \left(\frac{1}{2}\gamma - 1\right)\xi.$$



The resulting quadratic profile bows upward in the center region when the heat source is positive ( $\gamma > 0$ ), and bows down in the opposite case.

We now split the BVP into the two first-order ODEs:

$$\frac{d}{d\xi} \begin{bmatrix} T \\ y \end{bmatrix} = \frac{dz}{d\xi} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{z} - \begin{bmatrix} 0 \\ \gamma \end{bmatrix} = \mathbf{A}\mathbf{z} - \mathbf{b}.$$

Using our standard variable transformation  $\mathbf{s} = \mathbf{A}\mathbf{z} - \mathbf{b}$  we find the general solution

$$\mathbf{s}(\xi) = \mathbf{A}\mathbf{z}(\xi) - \mathbf{b} = q_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + q_2 \left( \xi \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right)$$

and so

$$\begin{aligned} y &= q_1 + q_2 \xi \\ \gamma &= q_2. \end{aligned}$$

Therefore, substituting the second equation above into the first and integrating *once* with respect to  $\xi$  gives

$$T(\xi) = \frac{1}{2}\gamma\xi^2 + q_1\xi + q_3$$

validating that after we determine the constants  $q_1$  and  $q_3$  using the boundary conditions, that both solution approaches produce precisely the same solution.

### 9.4.3 A diffusion-convection problem

Finally, let us consider the BVP

$$\frac{d^2 T}{d\xi^2} - \alpha \frac{dT}{d\xi} + \beta T = 0$$

subject to the same boundary conditions as before

$$\begin{aligned} T &= 1 \quad \text{at} \quad \xi = 0; \\ T &= 0 \quad \text{at} \quad \xi = 1. \end{aligned}$$

In this problem, the  $-\alpha(dT/d\xi)$  term represents the convective transport of thermal energy in the positive  $\xi$  direction when  $\alpha$ , a constant proportional to the velocity of the material, is positive. The last term  $\beta T$  represents a temperature dependent heat source.

Splitting the BVP into the two first-order ODEs:

$$\frac{d}{d\xi} \begin{bmatrix} T \\ y \end{bmatrix} = \frac{dz}{d\xi} = \begin{bmatrix} 0 & 1 \\ -\beta & \alpha \end{bmatrix} \mathbf{z} = \mathbf{A}\mathbf{z}.$$

We observe that  $\det(\mathbf{A}) = \beta$  and  $\text{tr}(\mathbf{A}) = \alpha$  indicating that

$$\lambda_{1,2} = \frac{-\alpha \pm \sqrt{\alpha^2 - 4\beta}}{2} \quad \text{with} \quad \mathbf{U} = \begin{bmatrix} 1 & 1 \\ \lambda_1 & \lambda_2 \end{bmatrix}.$$

We see that distinct, real eigenvalues will be found when the convection term coefficient is sufficiently large relative to the heat generation term coefficient. We can write the general solution as

$$T(\xi) = q_1 e^{\lambda_1 \xi} + q_2 e^{\lambda_2 \xi}$$

and the particular solution is found using the boundary conditions and solving the following linear system:

$$\begin{bmatrix} 1 & 1 \\ e^{\lambda_1} & e^{\lambda_2} \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

## 9.5 Review problems

1. Consider the CSTR modeling equations describing the constant-volume reactor in which the reaction  $A \rightleftharpoons B \rightleftharpoons C$  takes place:

$$\begin{aligned} \frac{dx_A}{dt} &= (0.1 - x_A) - 4x_A + x_B \\ \frac{dx_B}{dt} &= (0 - x_B) + 4x_A - x_B - x_B \end{aligned}$$

- (a) Write the equations in matrix form;
  - (b) Compute the steady state solution(s), if any;
  - (c) Compute the eigenvalues and associated eigenvectors of the coefficient array using hand-calculation methods that can be extended to higher-dimensional systems;
  - (d) Normalize the eigenvectors;
  - (e) Write the solution for  $x_A(0) = 1$ ,  $x_B(0) = 0$  and plot the results.
2. Consider the linear system of ODEs:

$$\frac{dz}{dt} = \mathbf{A}z - \mathbf{b}$$

with

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} -2 & 2 \\ 3 & -4 \end{bmatrix} \\ \mathbf{b} &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \end{aligned}$$

subject to initial conditions  $z = 0$ .

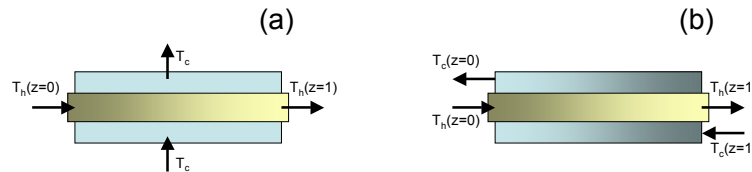
- (a) Compute the steady-state solution by hand using Gaussian elimination; compare your result to that produced using `linearsystemsolver.m`
  - (b) Using `lodesolver.m`, compute the solution in time, starting from  $t = 0$  and the given initial conditions; determine the length of time necessary for the solution to reach within 5 percent of the steady state value.
3. Consider the system  $dx/dt = \mathbf{A}x$ . For the following  $\mathbf{A}$  matrices, plot the trajectories as a function of time over the time interval  $0 \leq t \leq 2$  for the initial conditions  $x = [1; 1]$  for the first three systems, and for  $x = [1; 1; 1]$ .

$$\begin{aligned}
 A &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\
 A &= \begin{bmatrix} -1 & 0 \\ 0 & 3 \end{bmatrix} \\
 A &= \begin{bmatrix} -1 & -1 \\ 3 & -3 \end{bmatrix} \\
 A &= \begin{bmatrix} -1 & 2 & 3 \\ 0 & -2 & 0 \\ 0 & 1 & -1 \end{bmatrix}
 \end{aligned}$$

4. Compute the eigenvalues of the arrays in the previous problem using the MATLAB function eig.m and interpret the results of the first problem with respect to the conclusions you draw from a stability analysis of the system based on the eigenvalues.
5. Consider the system  $d\mathbf{x}/dt = \mathbf{A}\mathbf{x}$ . For the following  $\mathbf{A}$  matrices, compute the eigenvalues and eigenvectors by hand; plot the eigenvectors for each case and describe the expected dynamic behavior in each case (e.g., is the system stable?)

$$\begin{aligned}
 A &= \begin{bmatrix} -3 & 0 \\ 1 & -0.5 \end{bmatrix} \\
 A &= \begin{bmatrix} -3 & 0 \\ 0 & 1 \end{bmatrix} \\
 A &= \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \\
 A &= \begin{bmatrix} 0 & 2 \\ -2 & 0 \end{bmatrix} \\
 A &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\
 A &= \begin{bmatrix} 1 & 2 \\ -2 & -4 \end{bmatrix} \\
 A &= \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix}
 \end{aligned}$$

6. Consider the two heat exchangers shown below:



- (a) In this configuration, a hydrocarbon liquid flowing through a  $r = 1\text{cm}$  (internal) radius heat exchanger tube with length  $1\text{m}$ . The fluid enters at a temperature of  $T_h(z = 0) = 600\text{K}$  and is cooled by a boiling liquid on the outside of the tube; the latter liquid's boiling point is  $T_c = 373\text{K}$ . We can model the hydrocarbon liquid temperature as a function of the tube coordinate  $z$  with the single differential equation:

$$m_h C_{ph} \frac{dT_h}{dz} = 2\pi r h (T_c - T_h)$$

where the heat transfer coefficient is given as  $h = 7500\text{W}/(\text{m}^2 \text{K})$ ,  $C_{ph} = 2300\text{J}/(\text{kg K})$ , and the mass flow rate of the hydrocarbon stream is  $m_h = 0.1\text{kg/s}$ .

Using hand calculations, compute the hydrocarbon stream outlet temperature  $T_h(z = 1)$ ; compare your solution to that found by linearodesolver.m by plotting the results.

- (b) Now consider the case where the same hydrocarbon stream passes through the same heat exchanger tube, but now is cooled by a liquid flowing in the opposite direction (this is known as a counter-current flow). If the coolant fluid enters at  $z = 1\text{m}$  and has  $C_{pc} = 4200\text{J}/(\text{kg K})$ , exits the heat exchanger at  $z = 0$  at  $T_c(z = 0) = 550\text{K}$ , and flows at rate of  $m_c = 0.01\text{kg/s}$ , we can model the new heat exchanger system by adding the new equation

$$-m_c C_{pc} \frac{dT_c}{dz} = 2\pi r h (T_h - T_c)$$

For the case  $h = 300\text{W}/(\text{m}^2 \text{K})$  and  $m_h = 0.005\text{kg/s}$ , compute the *inlet* coolant temperature  $T_c(z = 1)$  and the outlet hydrocarbon stream temperature  $T_h(z = 1)$  using `lodesolver.m` (plot the results as a function of  $z$ ). Note that you must use the new value of  $h$  in both equations.

Compute the eigenvalues and discuss their physical significance.

## Chapter 10

### Case Study: Quenching dynamics

Consider a partially insulated tank with  $V_w$  m<sup>3</sup> of water into which a steel sphere of volume  $V_s$  is dropped. The tank walls are in contact with the laboratory's environment kept at a cool  $T_l = 10^\circ\text{C}$ . If the heat transfer coefficient between the sphere and water is  $h$  J/(m<sup>2</sup> s K), surface area of the sphere  $A$  m<sup>2</sup>, heat capacity of steel and water is  $Cp_s$  and  $Cp_w$ , respectively, and density of each is  $\rho_s$  and  $\rho_w$ , heat transfer coefficient between the tank and laboratory air  $h_c$  and heat transfer area  $A_c$ , the sphere and water temperatures  $T_s$  and  $T_w$  are defined as the deviation from the lab air temperature, and defining

$$\alpha_s = \frac{hA}{\rho_s V_s Cp_s} \quad \alpha_w = \frac{hA}{\rho_w V_w Cp_w} \quad \alpha_c = \frac{h_c A_c}{\rho_w V_w Cp_w}$$

we can write

$$\frac{dT_s}{dt} = \alpha_s (T_w - T_s) \quad (10.1)$$

$$\frac{dT_w}{dt} = \alpha_w (T_s - T_w) - \alpha_c T_w \quad (10.2)$$

subject to the initial conditions:

$$T_s(t=0) = 70\text{K} \quad T_w(t=0) = 10\text{K}.$$

The two modeling equations are written in matrix form,

$$\frac{d}{dt} \mathbf{z} = \frac{d}{dt} \begin{bmatrix} T_s \\ T_w \end{bmatrix} = \begin{bmatrix} -\alpha_s & \alpha_s \\ \alpha_w & -\alpha_w - \alpha_c \end{bmatrix} \mathbf{z}$$

For this example, we will consider a system consisting of a 10cm sphere dropped into 2 liters of water; this corresponds to the parameter values listed in Fig. 10.1. The resulting numerical values for the model coefficients are found to be

$$\alpha_s = 11.7 \text{ hr}^{-1} \quad \alpha_w = 2.7 \text{ hr}^{-1} \quad \alpha_c = 0.37 \text{ hr}^{-1}$$

It is straightforward to prove the only steady-state solution to (10.1)-(10.2), written as,

$$\begin{bmatrix} -\alpha_s & \alpha_s \\ \alpha_w & -\alpha_w - \alpha_c \end{bmatrix} \begin{bmatrix} T_s \\ T_w \end{bmatrix} = \mathbf{0}$$

is  $T_s = T_w = 0$  for  $\alpha_c > 0$ .

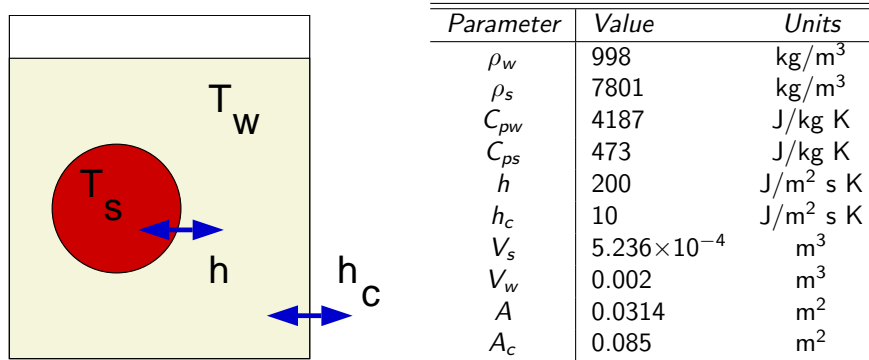


Figure 10.1: Water tank and steel sphere of the quenching dynamics problem.

## 10.1 Transient solution

If we now consider the problem of determining the solution to the full set of ODEs (10.1)-(10.2) subject to the initial conditions (10), we can pose the solution as

$$\mathbf{z} = e^{\mathbf{A}t} \mathbf{z}_0$$

With

$$\mathbf{A} = \begin{bmatrix} -11.7 & 11.7 \\ 2.7 & -3.1 \end{bmatrix}$$

and recalling the quadratic form of the characteristic equation for  $2 \times 2$  systems (9.10), we can immediately compute the eigenvalues as

$$\begin{aligned} \lambda_1, \lambda_2 &= \frac{-14.8 \pm \sqrt{14.8^2 - 4(11.7(3.1) - 11.7(2.7))}}{2} \\ &= -0.32, -14.5 \end{aligned}$$

Writing out the eigenvalue problem for the quench modeling equations,

$$\begin{bmatrix} -11.7 - \lambda & 11.7 \\ 2.7 & -3.1 - \lambda \end{bmatrix} \mathbf{u} = \mathbf{0}$$

so

$$\begin{bmatrix} -11.38 & 11.7 \\ 2.7 & -2.78 \end{bmatrix} \mathbf{u}_1 = \mathbf{0}$$

so we see that a solution accurate to one decimal place for  $\mathbf{u}_1$  is

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ 0.97 \end{bmatrix}.$$

We note that this is one of an infinite number of solutions because  $\mathbf{u}_1$  multiplied by any nonzero constant also will be a solution. Notice how the values of both elements in this eigenvector are nearly equal; because this eigenvector corresponds to the (much) smaller magnitude eigenvalue, we interpret this as an indicator that the sphere and water temperature will be nearly equal (which is greater?) as the system slowly relaxes to the laboratory room temperature.

For the second eigenvalue we find

$$\begin{bmatrix} 2.8 & 11.7 \\ 2.7 & 11.4 \end{bmatrix} \mathbf{u}_2 = \mathbf{0}$$

so a solution accurate to one decimal place is

$$\mathbf{u}_2 = \begin{bmatrix} 4.19 \\ -1 \end{bmatrix}.$$

## 10.2 Case $\alpha_c = 0$

For  $\alpha_c = 0$  it appears there is no unique solution because of the linear dependence of each row of **A**. This means a particular, steady-state solution for this model depends entirely on the initial conditions of the problem, a characteristic of closed systems. Equilibrium solutions to open systems such as the flash drum, on the other hand, normally will consist only of a finite number of solutions, and may depend on initial conditions in the case of multistability for nonlinear systems.

One piece of information we have not explicitly made use of is found by adding the two differential equations:

$$\begin{aligned} \frac{d}{dt} \left[ \frac{T_s}{\alpha_s} + \frac{T_w}{\alpha_w} \right] &= 0 \\ \alpha_w T_s + \alpha_s T_w &= \delta \quad (\text{a constant}) \\ \alpha_w T_s + \alpha_s T_w &= \alpha_w T_s(t=0) + \alpha_s T_w(t=0) \\ T_w &= \frac{\alpha_w T_s(t=0) + \alpha_s T_w(t=0)}{\alpha_s} - \frac{\alpha_w T_s}{\alpha_s} \end{aligned} \quad (10.3)$$

Substituting (10.3) into (10.1) gives

$$\begin{aligned} \frac{dT_s}{dt} &= \alpha_w T_s(t=0) + \alpha_s T_w(t=0) - \alpha_w T_s - \alpha_s T_s \\ \frac{dT_s}{dt} &= \gamma T_s + \delta \end{aligned} \quad (10.4)$$

where  $\gamma = -\alpha_w - \alpha_s$  and so is always negative and has a magnitude that increases with decreasing steel sphere total heat capacity ( $\rho_s V_s C p_s$ ).

### 10.2.1 Scalar ODE solution

The analysis leading to (10.4) indicates that (10.1)-(10.2) only has a single dynamic degree of freedom when  $\alpha_c = 0$ . This linear, scalar, time-invariant ODE is solved easily by the variable transform approach described earlier. With

$$y = \gamma T_s + \delta$$

the particular solution is

$$\begin{aligned} y &= y_0 e^{\gamma t} \\ \gamma T_s(t) + \delta &= [\gamma T_s(t=0) + \delta] e^{\gamma t} \\ T_s(t) &= -\frac{\delta}{\gamma} + \left[ T_s(t=0) + \frac{\delta}{\gamma} \right] e^{\gamma t} \end{aligned}$$

This solution makes physical sense, because at  $t = 0$   $T_s = T_s(t = 0)$  and as  $t \rightarrow \infty$   $\alpha_w T_s + \alpha_s T_s = \alpha_w T_s(t = 0) + \alpha_s T_w(t = 0)$  which implies  $T_s = T_w$  and that energy is conserved.

### 10.2.2 Solution in the phase plane

When  $\alpha_c = 0$ ,

$$\begin{aligned} 0 &= -\alpha_w - \lambda + \frac{\alpha_s \alpha_w}{\alpha_s + \lambda} \\ &= -(\alpha_s + \lambda)(\alpha_w + \lambda) + \alpha_s \alpha_w \\ &= \lambda^2 + (\alpha_s + \alpha_w)\lambda \alpha_w \\ \text{so } \lambda_{1,2} &= 0, -\alpha_s - \alpha_w. \end{aligned}$$

Substituting the eigenvalues  $\lambda_i$  into the eigenvalue problem, we can determine the corresponding eigenvectors  $\mathbf{u}_i$ :

$$\begin{aligned} \begin{bmatrix} -\alpha_s & \alpha_s \\ \alpha_w & -\alpha_w \end{bmatrix} \mathbf{u}_1 &= \mathbf{0} \\ \mathbf{u}_1 &= c_1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ corresponding to } \lambda_1 = 0 \\ \begin{bmatrix} \alpha_w & \alpha_s \\ \alpha_w & \alpha_s \end{bmatrix} \mathbf{u}_2 &= \mathbf{0} \\ \mathbf{u}_2 &= c_2 \begin{bmatrix} 1 \\ -\alpha_w/\alpha_s \end{bmatrix} \text{ corresponding to } \lambda_2 = -\alpha_s - \alpha_w \end{aligned}$$

The constants  $c_1, c_2$  can be anything including zero: we can write the results in array form:

$$\mathbf{V} = \begin{bmatrix} 1 & 1 \\ 1 & -\alpha_w/\alpha_s \end{bmatrix} \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix} = \mathbf{U}\mathbf{C}.$$

### 10.2.3 Physical interpretation of eigenvectors

We continue our practice of avoiding matrix inversion, and so actually solve the system

$$\begin{aligned} \mathbf{U}\mathbf{d} &= \mathbf{z}_0 \\ \begin{bmatrix} 1 & 1 \\ 1 & -\alpha_w/\alpha_s \end{bmatrix} \mathbf{d} &= \begin{bmatrix} T_s(0) \\ T_w(0) \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \\ 0 & -\alpha_w/\alpha_s - 1 \end{bmatrix} \mathbf{d} &= \begin{bmatrix} T_s(0) \\ T_w(0) - T_s(0) \end{bmatrix} \\ \mathbf{d} &= \begin{bmatrix} \frac{\alpha_s(T_w(0) - T_s(0))}{\alpha_s + \alpha_w} + T_s(0) \\ \frac{-\alpha_s(T_w(0) - T_s(0))}{\alpha_s + \alpha_w} \end{bmatrix} \end{aligned}$$



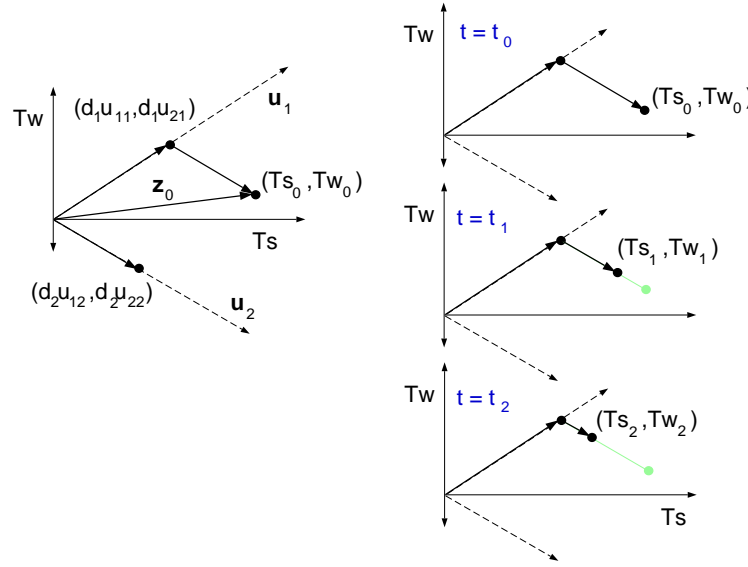


Figure 10.2: The geometry of the phase plane showing the decomposition of initial conditions (left) and the time-evolution of the quenching problem, showing the convergence to points on the eigenvector corresponding to the zero eigenvalue.

This gives a clear view into the modal structure of the particular solution, as shown in Fig. 10.2. For our system, we write the solution out explicitly as

$$\begin{bmatrix} T_s \\ T_w \end{bmatrix} = \left[ \frac{\alpha_s(T_w(0) - T_s(0))}{\alpha_s + \alpha_w} + T_s(0) \right] \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \left[ \frac{-\alpha_s(T_w(0) - T_s(0))}{\alpha_s + \alpha_w} \right] e^{-(\alpha_s + \alpha_w)t} \begin{bmatrix} 1 \\ -\alpha_w/\alpha_s \end{bmatrix}.$$

Notice how the zero eigenvalue means that solutions evolve parallel to the other (“nonzero”) eigenvector and ultimately come to fall on the eigenvector associated with the zero eigenvalues. Inspection of the coefficients of the ODE solution shows that

$$T_s, T_w \rightarrow \left[ \frac{\alpha_s(T_w(0) - T_s(0))}{\alpha_s + \alpha_w} + T_s(0) \right] \quad \text{as } t \rightarrow \infty.$$

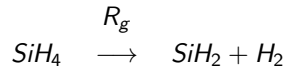
These results corroborate with our previous analysis of the steady-state, suggesting the vector  $d_1 \mathbf{u}_1$  can be thought of as the total energy of the system.

## Chapter 11

# Case Study: Isothermal CVD reactor model

Microelectronic devices are manufactured in a sequence of processing steps that include numerous thin-film deposition and etching operations. As one example, consider the reaction process where thin films of polycrystalline silicon are deposited using a Chemical Vapor Deposition (CVD) system [21].

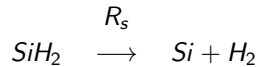
This process consists of a feed gas of silane ( $\text{SiH}_4$ ) fed to a vacuum chamber containing the wafer(s) to be processed. The reaction process consists of the gas phase decomposition of silane to silylene



where the gas phase reaction rate  $R_1$  is given by

$$R_g = k_g C_{\text{SiH}_4}$$

has units  $\text{mol}/(\text{m}^3 \text{ s})$ . The actual Si film deposition is governed by the surface reaction:



where

$$R_s = k_s C_{\text{SiH}_2}$$

and has units  $\text{mol}/(\text{m}^2 \text{ s})$ . We note that the gas phase reaction produces two moles of gas for each consumed while the surface reaction produces one mole of gas for each consumed.

If we approximate the gas phase behavior of this system as an isothermal, well-mixed reactor with pure silane feed ( $y_{\text{SiH}_4}^f = 1$ ) and for now assume the conversion rate of silane is low in this reactor (we will check the validity of this assumption later), a material balance on each gas phase chemical species gives the following nonlinear equations

$$\begin{aligned} CV \frac{dy_{\text{SiH}_4}}{dt} &= QC - (QC + k_g CV y_{\text{SiH}_4}) y_{\text{SiH}_4} - k_g CV y_{\text{SiH}_4} \\ CV \frac{dy_{\text{SiH}_2}}{dt} &= -(QC + k_g CV y_{\text{SiH}_4}) y_{\text{SiH}_2} - k_s CW_a y_{\text{SiH}_2} + k_g CV y_{\text{SiH}_4} \\ CV \frac{dy_{\text{H}_2}}{dt} &= -(QC + k_g CV y_{\text{SiH}_4}) y_{\text{H}_2} + k_s CW_a y_{\text{SiH}_2} + k_g CV y_{\text{SiH}_4} \end{aligned}$$

Under the assumption of low silane conversion in our system ( $y_{SiH_4} \rightarrow 1$  and  $y_{SiH_2}, y_{H_2} \rightarrow 0$ ), we can replace the nonlinear terms in the modeling equations with the following linear approximations (the topic of linearization will be discussed later in this text):

$$\begin{aligned} CV \frac{dy_{SiH_4}}{dt} &= QC - QCy_{SiH_4} - k_g CV - 2k_g CV(y_{SiH_4} - 1) - k_g CVy_{SiH_4} \\ &= -QCy_{SiH_4} - 3k_g CVy_{SiH_4} + QC + k_g CV \end{aligned} \quad (11.1)$$

$$CV \frac{dy_{SiH_2}}{dt} = -QCy_{SiH_2} - k_g CVy_{SiH_2} - k_s CW_a y_{SiH_2} + k_g CVy_{SiH_4} \quad (11.2)$$

$$CV \frac{dy_{H_2}}{dt} = -QCy_{H_2} - k_g CVy_{H_2} + k_s CW_a y_{SiH_2} + k_g CVy_{SiH_4} \quad (11.3)$$

For operating conditions typical for low-pressure CVD of poly-Si (1 torr total pressure and 900K wafer temperature) the reaction rate coefficients [21] and gas total concentration  $C$  computed using the ideal gas law are

$$\begin{aligned} k_g &= 0.016 \text{ s}^{-1} \\ k_s &= 190 \text{ m/s} \\ C &= 0.0178 \text{ mol/m}^3 \end{aligned}$$

If we consider a reactor loaded with 20, 200mm wafers, and that the reactor is a tube 1m in length and 0.15m in radius, and that the feed gas flow to the reactor is pure silane at 100 sccm, the remaining model parameters can be computed:

$$\begin{aligned} Q &= 0.0042 \text{ m}^3/\text{s} \\ V &= 0.0707 \text{ m}^3 \\ W_a &= 2 \times 20 \times 0.0314 \text{ m}^2 = 1.2566 \text{ m}^2 \end{aligned}$$

The final form of the reactor model can be written in matrix form as

$$\frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y} - \mathbf{b} \quad \text{with} \quad \mathbf{y} = \begin{bmatrix} y_{SiH_4} \\ y_{SiH_2} \\ y_{H_2} \end{bmatrix}$$

with

$$\mathbf{A} = \begin{bmatrix} -0.1071 & 0 & 0 \\ 0.016 & -3378 & 0 \\ 0.016 & 3378 & -0.0751 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -0.0751 \\ 0 \\ 0 \end{bmatrix}.$$

All coefficients have units  $\text{s}^{-1}$ .

## 11.1 CVD deposition rate

The steady state solution is computed by setting the time-derivatives to zero and solving the resulting linear system in the form of (4.1).

Using the constants listed, we can set up the coefficient array and vector of nonhomogeneous terms by

```
A = zeros(3,3); b = zeros(3,1);
```

```
A(1,1) = -Q/V - 3*kg;
```

```
b(1,1) = -Q/V - kg
```

```
A(2,1) = kg;
```

```
A(2,2) = -Q/V - kg - ks*Wa/V;
```

```
A(3,1) = kg;
```

```
A(3,2) = ks*Wa/V;
```

```
A(3,3) = -Q/V - kg;
```

and compute the steady-state solution using

```
>> format short e
```

```
>> y = A\b
```

```
y =
```

```
7.0115e-01
```

```
3.3211e-06
```

```
2.9885e-01
```

We can determine the deposition rate using this solution by computing the rate of the deposition reaction by solid silicon's molar density  $\rho = 82920 \text{ mol}/\text{m}^3$ :

```
>> DepRate = ks*C*y(2)/82920 * 1e6 * 3600 % deposition rate in microns/hr
```

```
DepRate =
```

```
0.4881
```

which indicates that this wafer would have to be processed for over one hour to achieve a  $0.5\mu\text{m}$  film, which is a typical length scale for microelectronic devices.

## 11.2 CVD reactor dynamics

Using the coefficient array **A** and nonhomogeneous array **b** used for computing the steady-state solution of the CVD problem, we can compute the solution to the dynamic problem using the spectral factorization technique of the `lodesolver.m` method:

```
t = [0:1:60];
```

```
y0 = [0; 0; 1];
```

```
y = lodesolver(A,t,y0,b);
```

As can be seen in Fig. 11.1, even though  $\text{SiH}_2$  governs the overall deposition rate, very little exists in the reactor at any time; the practical implication is that it would not be feasible to measure deposition rate indirectly by measuring the gas phase concentration of this component. We see that the system, started with a reactor filled with pure  $\text{H}_2$  at time  $t = 0$ , comes to equilibrium after about one minute of operation.

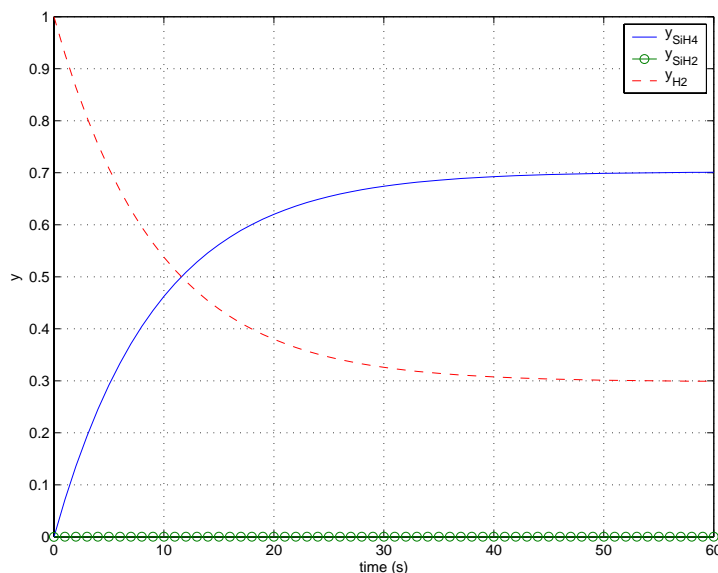


Figure 11.1: *Dynamic behavior of the CVD reactor during the first minute of start-up operation.*

### 11.3 Review problems

1. Rigorously argue that  $y_{SiH_4} + y_{SiH_2} + y_{H_2} = 1$  must always be satisfied in the poly-Si steady-state modeling equations. Then, compute a representative solution and condition number for the linear coefficient array and discuss any discrepancies between the computed and true sums.
2. Write a MATLAB program to compute the steady-state deposition rate (in  $\mu m/min$ ) for the polySi steady-state model. Use this program to compute deposition rates for  $100 \leq Q \leq 5000 sccm$ , in increments of  $100 sccm$  plotting the deposition rate as a function of  $Q$  over this interval. Compare this curve to the asymptotic deposition rate found as  $Q \rightarrow \infty$ .

## Chapter 12

# Nonlinear systems

This chapter focuses on developing numerical approaches to solving models in the form of sets of nonlinear algebraic equations. These models result from descriptions of physical systems and are created in two different ways:

1. **physically based** models derived from first-principles concepts, such as material or energy balances;
2. **empirical** models in the form of a Taylor's series expansion in the model parameters, with model coefficients determined by a parameter estimation technique using experimental data.

The models we consider can be written in the form

$$g(x) = 0$$

where the goal is to compute values of variable(s)  $x$  such that the equation(s) above is satisfied; we note that for nonlinear systems, there may be more than one solution  $x$  that satisfies  $g(x) = 0$ . Modeling problems such as these also are generated when steady state solutions are to be computed for differential equation models of the form

$$\frac{dx}{dt} = g(x).$$

### 12.1 The generic characteristics of a nonlinear AE model

Every nonlinear model has variables and parameters; the distinction is artificial and can be the source of trouble in developing modular Newton approaches; however, for now we will retain the concept because it is familiar to most engineers. As a generic nonlinear system, consider the following two independent nonlinear equations containing variables  $x$  and  $y$  and parameters  $p$  and  $q$ :

$$\begin{aligned}g(x, y, p) &= 0 \\f(x, y, q) &= 0\end{aligned}$$

The second key feature of nonlinear algebraic equation models are the modeling equations themselves. For the systems we study, we consider there are to be at least as many equations and variables for the

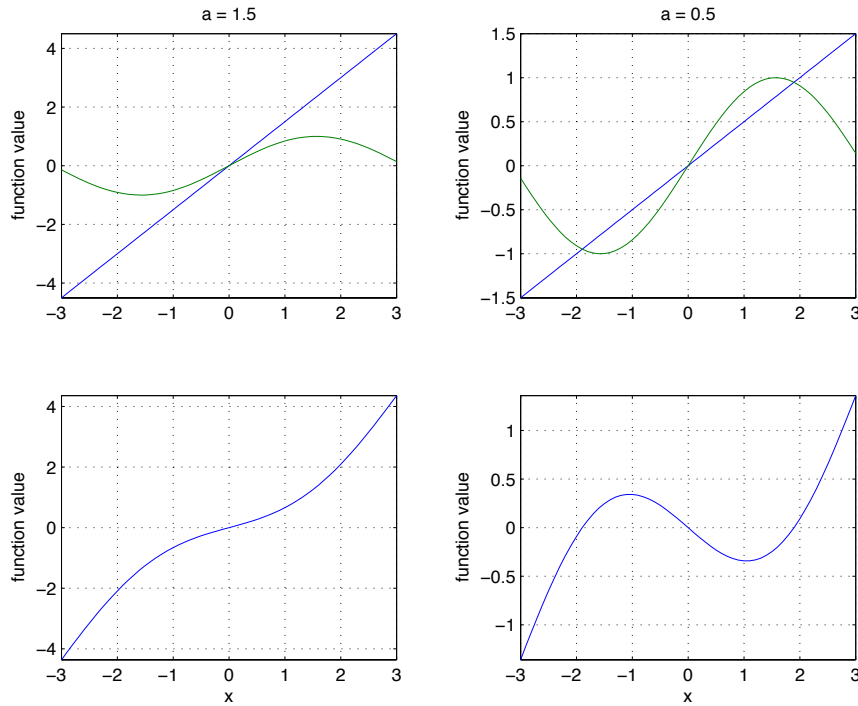


Figure 12.1: Multiple methods for viewing the solution to  $g(x) = ax - \sin(x) = 0$ .

system. We note that because parameters can be defined in terms of a equation, we see the previously noted artificiality of the distinction between parameters and variables. The modeling equations define a residual - the degree of inaccuracy of each modeling equation as determined by the current variable values.

## 12.2 Nonlinear model examples

Let us consider the nonlinear system

$$g(x) = ax - \sin(x) = 0. \quad (12.1)$$

We can view finding solutions to this problem in two different ways; the first is to re-write (12.1) as

$$g(x) = g_1(x) - g_2(x) = ax - \sin(x)$$

and plot each function  $g_1(x)$ ,  $g_2(x)$  individually as a function of  $x$  to find the intersections of  $g_1$  and  $g_2$ . These intersections determine the solutions to (12.1). Plots for two different values of parameter  $a$  are illustrated in Fig. 12.1.

Alternatively, we can plot (12.1) directly to determine the intersections with  $g = 0$ ; one may argue there is a slight disadvantage for this problem in that we lose some information that gives insight into the “global” behavior of this system. For example, compared to the previous approach of plotting the

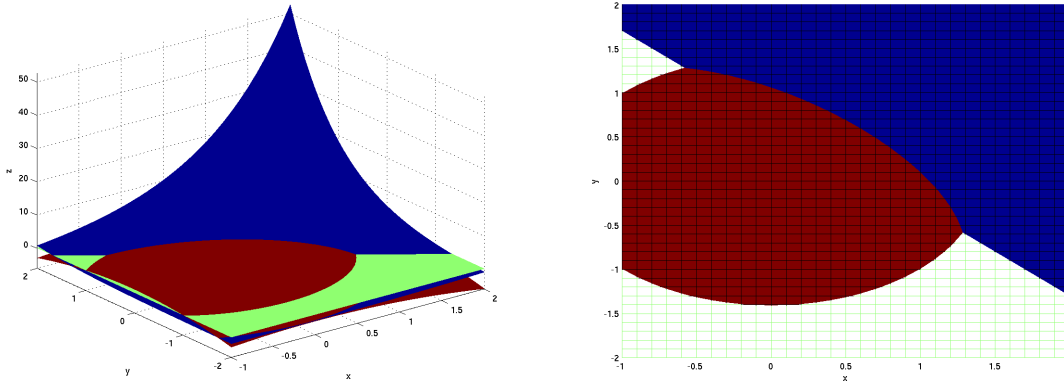


Figure 12.2: Two views of the solutions to a set of nonlinear algebraic equations. Locations of two solutions are indicated by intersection points of all three surfaces, where the red surface denotes  $z = g(x, y)$ , blue denotes  $z = f(x, y)$  and green denotes  $z = 0$ .

separate terms making up  $g(x)$ , we can no longer visualize the interaction of these terms that gives rise to a different number of solutions for different values of parameter  $a$  (see the lower plots of Fig. 12.1).

### 12.2.1 Multiple equation models

We also will consider systems of nonlinear equations, such as

$$\begin{aligned} f(x, y) &= \exp(x + y) - 2 \\ g(x, y) &= -x^2 - y^2 + 2 \end{aligned}$$

where the sets of equations cannot be decoupled to reduce the system to a single variable and single equation. Again, we can take a visual approach to understanding the solutions to this system of equations by plotting  $z =$  the right-side of each as a function of the independent variables  $x$  and  $y$ . As seen in Fig. 12.2, solutions to the set of equations require the simultaneous intersection of the  $f(x, y)$  and  $g(x, y)$  together with the plane defined by  $z = 0$ ; from the plots, we observe two potential solutions, one at  $(x, y) \approx (-0.59, 1.28)$  and the other at  $(x, y) \approx (1.28, -0.59)$ . The symmetry of the solutions is not surprising considering the interchangeability of  $x$  and  $y$  in the modeling equations.

If we modify  $g$  slightly to be

$$g(x, y) = -x^2 - y^2 + b$$

we can imagine that as we reduce the value of  $b$  from 2, the red, dome-shaped surface defined by this equation will sink into the green  $z = 0$  plane, and will eventually lose contact with  $f$  resulting in no real solutions to this problem. Visually, we find this value to be approximately  $b = 0.25$ ; for  $b$  less than this value, there appear to be no solutions to the problem.



## 12.3 Newton's method

Consider the problem of finding solution(s) to

$$g(x; p) = 0$$

We now solve for solutions  $x_s$  using the iterative Newton's method for a specified set of parameters  $p$ . If  $x^0$  is our guess for a solution, a more refined solution  $x^1$  can be computed using the Taylor's series of  $g$  evaluated at the current solution estimate  $x^0$ :

$$g(x^1) \approx g(x^0) + \left. \frac{dg}{dx} \right|_{x^0} (x^1 - x^0) + \frac{1}{2!} \left. \frac{d^2g}{dx^2} \right|_{x^0} (x^1 - x^0)^2 + \text{h.o.t.}$$

where h.o.t. refers to order three and higher derivative terms. To compute the more refined solution estimate  $x^1$ , a linear curve  $g_L(x)$  tangent to  $g(x)$  at the current solution estimate  $g(x^0)$  is defined from the Taylor's series expansion by

$$g_L(x) = g(x^0) + \left. \frac{dg}{dx} \right|_{x^0} (x - x^0).$$

We see that  $g_L(x)$  is a linear approximation to the true  $g(x)$ , but one that can be solved explicitly for the value of  $x$  where the function crosses zero; therefore, setting the right side of the equation above equal to zero and solving for the corresponding  $x = x^1$  gives

$$x^1 = x^0 - \frac{g(x^0)}{dg/dx|_{x^0}}$$

with

$$\frac{dg}{dx} = a - \cos(x) \tag{12.2}$$

for example (12.1). It is interesting to observe that because (12.2) is positive for all parameter values  $a > 1$ , the problem  $g(x) = 0$  must have a unique solution under these conditions.

Produced by this computation is a (hopefully) better estimate  $x^1$  of a true solution to  $g(x) = 0$ ; the word hopefully is used because the actual convergence behavior of the Newton procedure can be difficult to predict. We can continue to refine the solution estimate by replacing the previous estimate  $x^0$  with the newer estimate  $x^1$  and then repeating the procedure until a sufficiently accurate solution is reached.

### 12.3.1 Quadratic convergence - numerical analysis

The convergence of Newton's method for solving  $g(x) = 0$  (for cases where the first  $g'$  and second  $g''$  derivatives do not vanish and for sufficiently-good initial guesses) is said to be quadratic: if  $x = r$  is the numerical value of the root we are attempting to find, in the neighborhood of  $x = r$  with  $\epsilon = x - r$ ,

$$\begin{aligned} g(x) &\approx g(r) + (x - r)g'(r) + \frac{1}{2}(x - r)^2g''(r) \\ &= 0 + \epsilon g'(r) + \frac{1}{2}\epsilon^2 g''(r) \end{aligned}$$

Differentiating each term of the right side of the equation above with respect to  $x$  (i.e., with respect to  $\epsilon$ ) gives

$$g'(x) = g'(r) + \epsilon g''(r).$$

Thus we can analyze our Newton scheme to find

$$\begin{aligned}
 x^1 &= x^0 - \frac{g(x^0)}{g'(x^0)} \\
 x^1 - r &= x^0 - r - \frac{g(x^0)}{g'(x^0)} \\
 \epsilon^1 &= \epsilon^0 - \frac{\epsilon^0 g'(r) + \frac{1}{2}(\epsilon^0)^2 g''(r)}{g'(r) + \epsilon^0 g''(r)} \\
 &= \frac{\epsilon^0 g'(r) + (\epsilon^0)^2 g''(r) - \epsilon^0 g'(r) - \frac{1}{2}(\epsilon^0)^2 g''(r)}{g'(r) + \epsilon^0 g''(r)} \\
 &= \frac{\frac{1}{2}(\epsilon^0)^2 g''(r)}{g'(r) + \epsilon^0 g''(r)}
 \end{aligned}$$

so for sufficiently small  $\epsilon^0$ ,

$$\epsilon^1 \approx (\epsilon^0)^2 \frac{g''(r)}{2g'(r)} = C(\epsilon^0)^2.$$

Thus, we see that after six iterations, the error becomes

$$\epsilon^6 = \epsilon^0 (C\epsilon^0)^{63}$$

which normally will be sufficiently accurate if  $|C\epsilon^0| < 1$  because, for example, if  $|C\epsilon^0| = 0.5$

$$\epsilon^6 = 0.5 (0.5)^{63} = 5.4 \times 10^{-20}$$

which is less than round-off error. It is important to point out, however, that choosing initial guesses based on  $|C\epsilon^0| < 1$  do not necessarily have to converge to the fixed points used to calculate the values of  $C$ .

The analysis above illustrates the quadratic convergence behavior of the Newton method when the estimate of the solution approaches the true value. Because the next value of the error  $\epsilon^1$  is proportional to the square of the previous error  $\epsilon^0$ , the accuracy of the solution estimates improves with each iteration in such a way that the number of accurate digits doubles with each iteration, i.e., if  $\epsilon^0 = 0.001$ , then  $\epsilon^1 = 0.00001$  in the neighborhood of the converged solution. We note that explicit computation of  $g'$  and  $g''$  may be used to gain further insight into the true accuracy of the solution.

Casting the Newton procedure in terms of updates

$$\begin{aligned}
 u^1 &= -\frac{g(x^0)}{g'(x^0)} \\
 &= -\frac{(x^0 - r)g' + (1/2)(x^0 - r)^2 g''}{g' + (x^0 - r)g''} \\
 u^2 &= -\frac{g(x^1)}{g'(x^1)} \\
 &= -\frac{g(x^0 + u^1)}{g'(x^0 + u^1)} \\
 &= -\frac{(x^0 + u^1 - r)g' + (1/2)(x^0 + u^1 - r)^2 g''}{g' + (x^0 + u^1 - r)g''} \\
 &= -\frac{(x^0 - r)g' + (1/2)(x^0 - r)^2 g'' + u^1 g' + (1/2)[(u^1)^2 + 2u^1(x^0 - r)]g''}{g' + (x^0 + u^1 - r)g''} \\
 &= -\frac{-u^1[g' + (x^0 - r)g''] + u^1 g' + (1/2)[(u^1)^2 + 2u^1(x^0 - r)]g''}{g' + (x^0 + u^1 - r)g''} \\
 &= -\frac{(1/2)(u^1)^2 g''}{g' + (x^0 + u^1 - r)g''} \\
 &\approx -\frac{g''}{2g'}(u^1)^2 \quad \text{because } x^0 + u^1 - r = x^1 - r = \text{small}.
 \end{aligned}$$

Because this analysis was done in the neighborhood of the true solution point  $r$ , we see that convergence of  $u$  to zero implies convergence of  $x$  to  $r$ , and that the (negative!) exponent of the update will double during each iteration. This is demonstrated in the following computational example.

### 12.3.2 Computational example: Quadratic convergence of Newton's method

We consider the MATLAB script written to solve the single-equation model problem (12.1)

```

% Newton procedure for g = ax-sin(x)
a = 0.5;
x = 1.5; % initial solution estimate
for iters = 1:6
    g = a*x - sin(x);
    dg = a - cos(x);
    update = -g/dg;
    x = x + update;
end

```

Given an initial guess of  $x^0 = 1.5$ , the updates converge with the following behavior to the solution  $x = 1.8955$  for  $a = 0.5$

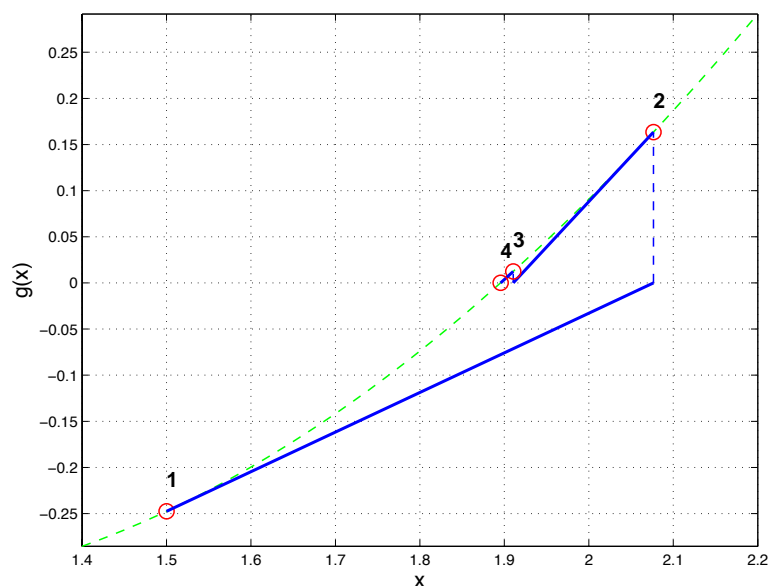


Figure 12.3: Convergence of the Newton scheme.

Iteration $i$	Update	$x^i$
0	-	1.5
1	5.7656e-01	2.0766
2	-1.6605e-01	1.9105
3	-1.4885e-02	1.8956
4	-1.2773e-04	1.8955
5	-9.4388e-09	1.8955
6	-2.7111e-16	1.8955

and so the exponent of the update indeed approximately doubles during each iteration as the algorithm approaches the true solution (note that the update does now go below  $1 \times 10^{-16}$  in magnitude because this value is approximately machine roundoff). A graphical view of the Newton iterations is shown in Fig. 12.3, where the blue-line segments are plots of the linear function  $f(x)$  while the green dashed curve is  $g(x)$ .

Before concluding this section, several important general points on the Newton method must be made:

- The numerical technique will not converge when the derivative vanishes at the root;
- The solution computed can depend strongly on the initial guess: this is particularly true for problems that have multiple solutions.

## 12.4 Multiple equations: The Newton-Raphson method

Returning to our representative set of nonlinear algebraic equations

$$\mathbf{h}(\mathbf{z}) = \begin{bmatrix} f(x, y) \\ g(x, y) \end{bmatrix} = \begin{bmatrix} \exp(x + y) - 2 \\ -x^2 - y^2 + b \end{bmatrix} \quad (12.3)$$

we wish to find solutions to the problem written in vector form

$$\mathbf{0} = \mathbf{h}(\mathbf{z}) \quad \text{with} \quad \mathbf{z} = \begin{bmatrix} x \\ y \end{bmatrix}$$

and

$$\begin{aligned} h_1(x, y) &= \exp(x + y) - 2 \\ h_2(x, y) &= -x^2 - y^2 + b \end{aligned}$$

We can write the Taylor's series expansion of  $\mathbf{h}$  at  $(x^0, y^0)$  as

$$\begin{aligned} h_1(x, y) &\approx h_1(x^0, y^0) + \frac{\partial h_1}{\partial x} \Big|_{x^0, y^0} (x - x^0) + \frac{\partial h_1}{\partial y} \Big|_{x^0, y^0} (y - y^0) + \frac{\partial^2 h_1}{\partial x \partial y} \Big|_{x^0, y^0} (x - x^0)(y - y^0) + \dots \\ h_2(x, y) &\approx h_2(x^0, y^0) + \frac{\partial h_2}{\partial x} \Big|_{x^0, y^0} (x - x^0) + \frac{\partial h_2}{\partial y} \Big|_{x^0, y^0} (y - y^0) + \frac{\partial^2 h_2}{\partial x \partial y} \Big|_{x^0, y^0} (x - x^0)(y - y^0) + \dots \end{aligned}$$

or, neglecting the second and higher-order terms

$$\mathbf{h}(\mathbf{z}) \approx \mathbf{h}(\mathbf{z}^0) + \mathbf{J}(\mathbf{z} - \mathbf{z}^0). \quad (12.4)$$

For the representative model, the Jacobian matrix  $\mathbf{J}$  is found to be

$$\mathbf{J} = \begin{bmatrix} J_{1,1} & J_{1,2} \\ J_{2,1} & J_{2,2} \end{bmatrix} = \begin{bmatrix} \exp(x + y) & \exp(x + y) \\ -2x & -2y \end{bmatrix}.$$

This gives us the Newton-Raphson procedure,

$$\mathbf{z}^1 = \mathbf{z}^0 - \mathbf{J}^{-1} \mathbf{h}(\mathbf{z}^0). \quad (12.5)$$

Of course, we do not actually invert the Jacobian  $\mathbf{J}$ ; instead, we use the Gaussian elimination (or similar) procedure described in the linear systems chapter to solve

$$\mathbf{J}^0 \mathbf{u}^1 = -\mathbf{h}(\mathbf{z}^0) \quad \text{with} \quad \mathbf{u}^1 = \mathbf{z}^1 - \mathbf{z}^0$$

for the update vector  $\mathbf{u}$ , which is used to compute the refined solution estimate  $\mathbf{z}^1$  from

$$\mathbf{z}^1 = \mathbf{z}^0 + \mathbf{u}^1.$$

### 12.4.1 Computational example: The Newton-Raphson method

We now consider solving the two-equation model problem for  $a = 2$  using a Newton-Raphson technique to observe the convergence behavior using the following MATLAB script:

```

% Newton-Raphson procedure applied to the 2-equation model problem

x = 2; y = 1; % initial solution guess
b = 2;

for iters = 1:8
    h1 = exp(x+y) - 2;
    h2 = - x^2 - y^2 + b;
    J11 = exp(x+y);
    J12 = exp(x+y);
    J21 = -2*x;
    J22 = -2*y;

    h = [h1; h2];
    J = [J11 J12; J21 J22];
    update = -J\h;
    normup = sqrt(update'*update);
    x = x + update(1);
    y = y + update(2);
end

```

For this case we compute the length (2-norm) of the update, and again observe the quadratic convergence in the neighborhood of the solution  $x = 1.2846$  and  $y = -0.5914$ ; we note that the solution procedure makes several large initial updates that do not follow the quadratic convergence pattern:

Iteration i	Update	$x^i$	$y^i$
0	-	2	1
1	6.7082e-01	1.4004	0.6991
2	1.2630e+00	1.8323	-0.4877
3	4.4572e-01	1.3879	-0.5220
4	1.1445e-01	1.2925	-0.5853
5	9.9704e-03	1.2847	-0.5914
6	7.0877e-05	1.2846	-0.5914
7	3.5112e-09	1.2846	-0.5914
8	9.5674e-17	1.2846	-0.5914

### 12.4.2 Jacobian arrays by finite differences

We can compute approximate values of the derivatives needing for Newton's method with finite differences: if  $\delta$  is a small number (but significantly larger than the machine roundoff error),  $\mathbf{g}$  is a vector of nonlinear equations, and  $\mathbf{z}$  is the vector of variables, for centered finite differences

$$\left. \frac{\partial g_i}{\partial z_j} \right|_{\mathbf{z}^0} \approx \frac{g_i(\mathbf{z}_j^0 + \delta) - g_i(\mathbf{z}_j^0 - \delta)}{2\delta}$$

and for forward differences

$$\left. \frac{\partial g_i}{\partial z_j} \right|_{\mathbf{z}^0} \approx \frac{g_i(\mathbf{z}_j^0 + \delta) - g_i(\mathbf{z}_j^0)}{\delta}$$

The Jacobian array can be constructed using either of these approaches

$$\mathbf{J}|_{\mathbf{z}^0} = \begin{bmatrix} \frac{\partial \mathbf{g}}{\partial z_1} & \frac{\partial \mathbf{g}}{\partial z_2} & \dots \end{bmatrix}$$

### Selection of perturbation size

The perturbation  $\delta$  used in the forward finite difference calculation actually varies according to the magnitude of the variable  $z_j$  to which the variation of the residuals are being computed; following [18]:

$$\delta_j = \sqrt{\epsilon} \max[|z_j|, 1]$$

where  $\epsilon$  corresponds to machine epsilon (round-off error), found using the MATLAB `eps.m` function.

### 12.4.3 Our basic Newton-Raphson function

```
function [z,conflag,J,r] = nr(z0,p,rfun,Jfun)
% NR A general Newton-Raphson method. Called as
%   [z,conflag] = nr(z0,p,rfun,Jfun)
%   where z0 is the initial guess in column format, p is a vector of
%   parameters, rfun the residual function and Jfun the Jacobian function.
%   Note that Jfun is computed by finite differences if not specified in
%   the input parameter list. Also note that rfun and Jfun take standard
%   MATLAB format: rfun(t,z,p) and Jfun(t,z,p) where an empty array [] is
%   passed for t.

t = NaN; % dummy value
z = z0; small = sqrt(eps); conflag = 0;
if nargin == 1, p = []; end

for iters = 1:20
    r = rfun(t,z,p);
    if nargin <= 3
        J = numJ(z,p,rfun);
    else
        J = Jfun(t,z,p);
    end
    update = -J\r;
    z = z+update;
    normup = sqrt(update'*update);
    if normup <= small % converged solution
        conflag = 1; disp(['iterations to convergence: ',num2str(iters)])
        break
    end
end
end
```

```

function J = numJ(x,p,rfun)
% compute the Jacobian array by forward finite differences

t = NaN; % dummy parameter
m = length(x);
delt = sqrt(eps);
for i = 1:m
    xp = x;
    xp(i,1) = xp(i,1)+delt;
    rp = rfun(t,xp,p);
    r = rfun(t,x,p);
    J(:,i) = (rp-r)/delt;
end

```

Note that the function to be called takes the format used by MATLAB's built-in ODE solvers:

```
residual = funname(t,x,p)
```

where  $t$  is a dummy parameter. The Newton-Raphson function then is typically called in the following manner

```

>> f = @funname;
>> [x,conflag] = nr(f,x0,p);
>> if conflag, disp('Converged solution, x = '), end
>> disp(x)

```

#### 12.4.4 Jacobian-free formulations

The Newton-Raphson update can be computed using

$$\mathbf{u}^{M \times 1} = -[\mathbf{J}^{M \times M}]^{-1} \mathbf{g}^{M \times 1}$$

and is used to refine an estimate of the solution  $\mathbf{z}^{M \times 1}$ :

$$\mathbf{z}^1 = \mathbf{z}^0 + \mathbf{u}$$

If  $\mathbf{V}$  is an orthonormal basis for  $\mathbf{u}$ , we can write the update as

$$\mathbf{u}^{M \times 1} = \mathbf{V}^{M \times N} \mathbf{y}^{N \times 1} \quad N \leq M$$

and so computing the update in the Newton-Raphson procedure becomes

$$\mathbf{J}^{M \times M} \mathbf{V}^{M \times N} \mathbf{y}^{N \times 1} = -\mathbf{g}^{M \times 1}$$

We can solve for  $\mathbf{y}$  using a least-squares (a Galerkin projection) procedure:

$$\begin{aligned} \mathbf{V}^T [\mathbf{J} \mathbf{V}] \mathbf{y} &= -\mathbf{V}^T \mathbf{g} \\ \mathbf{y} &= -(\mathbf{V}^T [\mathbf{J} \mathbf{V}])^{-1} \mathbf{V}^T \mathbf{g} \end{aligned}$$

where the array  $[\mathbf{J} \mathbf{V}]$  defined by the product of the Jacobian array  $\mathbf{J}$  and each (column) basis vector  $\mathbf{V}_i$  is computed using the finite-difference approximation

$$\mathbf{J} \mathbf{V}_i = \frac{\mathbf{g}(\mathbf{z}^0 + \delta \mathbf{V}_i) - \mathbf{g}(\mathbf{z}^0)}{\delta}$$

and so avoids explicit computation and storage of the Jacobian elements.



### Recovering eigenvalues of $\mathbf{J}$

One potential disadvantage to using the Jacobian-free formulations is that because  $\mathbf{J}$  is not available, we cannot assess the linear stability characteristics of the system by simply computing the eigenvalues of  $\mathbf{J}$ . However, we can examine the relationship between  $\mathbf{J}\mathbf{v}$  and the eigenvalue problem

$$\mathbf{J}\mathbf{v} = \lambda\mathbf{v} \quad (12.6)$$

by considering  $\mathbf{y}^{N \times 1}$  to be the linear combination of the columns of  $\mathbf{V}$  that generates an eigenvector  $\mathbf{v}$  satisfying (12.6). Therefore,

$$\begin{aligned} \mathbf{J}\mathbf{v} &= \lambda\mathbf{v} \\ \mathbf{J}\mathbf{V}\mathbf{y} &= \lambda\mathbf{V}\mathbf{y} \\ \mathbf{V}^T \mathbf{J}\mathbf{V}\mathbf{y} &= \mathbf{V}^T \lambda\mathbf{V}\mathbf{y} \\ \mathbf{W}\mathbf{y} &= \lambda\mathbf{I}\mathbf{y}. \end{aligned}$$

This means we can compute  $N$  of the  $M$  eigenvalues of  $\mathbf{J}$  as the eigenvalues  $\lambda_n$  of the  $N \times N$  matrix  $\mathbf{W}$  and recover the eigenvectors  $\mathbf{v}_n$  of  $\mathbf{J}$  corresponding to these eigenvalues from  $\mathbf{v}_n = \mathbf{V}\mathbf{y}_n$ . The question now becomes how one chooses the matrix  $\mathbf{V}$  so that its vectors span the same space as the eigenvectors most relevant to the stability analysis of the system (generally, the slowest modes).

### 12.4.5 Krylov subspace methods

The residual for the linear equation is

$$\mathbf{r} = \mathbf{g} + \mathbf{J}\mathbf{u}$$

so for  $\mathbf{u} = \mathbf{0}$

$$\mathbf{r}^0 = \mathbf{g}.$$

The Krylov subspace is defined as the space spanned by the (column) vectors

$$\mathbf{K} = [\mathbf{g}, \mathbf{J}\mathbf{g}, \mathbf{J}^2\mathbf{g}, (\mathbf{J})^3\mathbf{g}, \dots, \mathbf{K}_i, \dots]$$

where

$$\mathbf{K}_i = \frac{\mathbf{g}(\mathbf{z} + \delta\mathbf{K}_{i-1}) - \mathbf{g}(\mathbf{z})}{\delta} \quad i > 1$$

and

$$\mathbf{K} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^T$$

where we retain only those vectors  $\mathbf{V}_i$  that correspond to significant singular values  $\sigma_i$ .

This leaves the question: why is  $\mathbf{K}$  a good basis for representing the update? If we return to the linear system

$$\mathbf{J}\mathbf{u} + \mathbf{g} = \mathbf{0}$$

and write the residual as

$$\mathbf{r} = \mathbf{J}\mathbf{u} + \mathbf{g}$$

choosing  $\mathbf{u} = \mathbf{0}$  gives the residual

$$\mathbf{r}^0 = \mathbf{g}$$

and so if our goal is to choose a set of basis functions to represent  $\mathbf{u}$  that are the same as the test functions used in a Galerkin procedure to solve the linear system, the first basis function should be

$$\mathbf{K}_1 = \mathbf{g}$$

because we are guaranteed that  $\mathbf{g}$  will be a component in the residual. So now, our assumed solution is

$$\mathbf{u} = \mathbf{0} + a_1 \mathbf{K}_1 = \mathbf{0} + a_1 \mathbf{g}$$

so the resulting residual is

$$\mathbf{r}_1 = a_1 \mathbf{Jg} + \mathbf{g}$$

and so it makes sense to choose  $\mathbf{K}_2 = \mathbf{Jg}$ , because the residual will be some linear combination of  $\mathbf{g}$  and  $\mathbf{Jg}$ . Continuing this line of thinking gives

$$\mathbf{u} = \mathbf{0} + a_1 \mathbf{g} + a_2 \mathbf{Jg}$$

so

$$\mathbf{r}_3 = a_2 \mathbf{J}^2 \mathbf{g} + a_1 \mathbf{Jg} + \mathbf{g}$$

and so we see why

$$\mathbf{K} = [\mathbf{g}, \mathbf{Jg}, \mathbf{J}^2 \mathbf{g}, \mathbf{J}^3 \mathbf{g}, \dots]$$

is a good basis for the solution procedure.

## 12.5 The naemodel class

The `naemodel.m` class constructor is used to define the parameters and variables of the model and to set up initial approximations to the solution - the initial guess for the Newton procedure. In this class, the `rhs.m` method returns a double or cell array containing the current value of the residual. The `jacobian.m` method returns the Jacobian array for the problem, computed using finite differences, and the `newton.m` method performs the Newton iterations for the problem, determining a search direction using the Jacobian array and then performing a linesearch to determine an update that reduces the residual norm.

<b>naemodel</b>
var : assocarray param : assocarray rhsname : char plotflag, nmax, lsmax, jup, nkvec : double
display(A) jacobian(A) naemodel(var,param,rhsname,plotflag,nmax,lsmax,jup newton(A,Abar,biffparam,ds) param2var(A,key,newval) predictor(A,biffparam,ds,dvds) sensitivity(A,Aparamkey) residual(A) rhs(A) unpack(A) var2param(A,key,newval)

The simplest way to use the `naemodel` class is to create an modeling object (a module) by calling the `naemodel.m` constructor directly, using the name of a MATLAB function that returns the “right-hand side” of the modeling equations as one of the input parameters to the constructor. The form of this function should be that of the standard “`odefun`” type function used when calling MATLAB numerical integrators.

One can also build more specialized modules by deriving a new class from `naemodel`, i.e., for the `naetemplate` class provided with the `mdpsas` software library,

1. we create the directory `naetemplate`;
2. define the constructor method `naetemplate.m` in such a way that the new class derives from the `naemodel` class;
3. create a new `rhs.m` and possibly `residual.m` method to overload the default methods;
4. create an instance of the new class to initialize the *state* of the modeling object; at the outset this operation corresponds to setting the variable values to the initial guesses for the Newton procedure and setting the parameter values;
5. and apply the inherited `newton.m` method to solve the problem.

To see these steps in action, we examine the `naetemplate.m` constructor method

```
function P = naetemplate
% naetemplate.m A nonlinear equation model object constructor, derived
%               from the naemodel class.
var   = assocarray({'x' 2 'y' 1});
param = assocarray({'b' 2});

A = naemodel(var,param);
P = class(struct([]),'naetemplate',A);
```

In the `rhs.m` method, the “right-hand sides” of the modeling equations are defined; in computing the steady-state solutions, these equations define the residual values that are to be driven to zero. Inside the `rhs.m` method, the variables and parameter values are unpacked from the modeling object and are placed in the workspace of the `rhs.m` function. The residual functions then are evaluated and the residual values, stored as a MATLAB double or cell array, are returned:

```
function r = rhs(A)

% rhs.m Returns the right-hand side values of the modeling
%       equations corresponding to naemodel object A
%
% INPUT/OUTPUT PARAMETERS
%       A : an object derived from the naemodel class

unpack(A)
r{1} = exp(x+y) - 2;
r{2} = - x^2 - y^2 + b;
```

And now we interactively solve the problem in a MATLAB session by instantiating `naetemplate` object `A` with the variable state corresponding to the initial solution estimate:

```
>> A = naetemplate
A = naetemplate

naetemplate object "A"
Variables
-----
x : 2
y : 1
Parameters
-----
b : 2
```

and now the Newton-Raphson method of the `naemodel` class is used to compute one of the actual solutions:

```
>> A = newton(A)

Update/residual norm = 1.5348 / 1.6087
Update/residual norm = 0.99912 / 0.43382
Update/residual norm = 0.17198 / 0.039344
Update/residual norm = 0.012998 / 0.0015958
Update/residual norm = 0.00053224 / 7.2802e-05
Update/residual norm = 2.4271e-05 / 3.306e-06
Update/residual norm = 1.1022e-06 / 1.5016e-07
Update/residual norm = 5.0062e-08 / 6.8202e-09
Update/residual norm = 2.2738e-09 / 3.0977e-10
Update/residual norm = 1.0328e-10 / 1.407e-11
```

This example shows that the finite difference-based Newton-Raphson method of the `naemodel` class works effectively at computing the same solution found using the Newton procedure with the explicitly defined Jacobian array, although at a slightly reduced convergence rate.

At this point, the modeling object `A` has stored in its `var` field the `assocarray` object containing the variable values corresponding to the converged solution. We can view the solution values using the `naemodel/display.m` method:

```
>> A

naetemplate object "A"
Variables
-----
x : 1.2846
y : -0.59145
Parameters
-----
b : 2
```

We can unpack the parameters and variables from the converged model object, placing copies in the current workspace:

```
>> unpack(A)
>> who
Your variables are:
A  b  x  y
```

## 12.6 Sensitivity analysis

Given the set of nonlinear equations and a single parameter  $p$

$$\mathbf{h}(\mathbf{z}, p) = 0 \quad \text{at} \quad \mathbf{z} = \bar{\mathbf{z}}$$

and a solution  $\bar{\mathbf{z}}(\bar{p})$  for  $p = \bar{p}$ , we define the sensitivity of the solution  $\bar{\mathbf{z}}$  to  $p$  as

$$\mathbf{S}(\bar{p}) = \left. \frac{d\bar{\mathbf{z}}}{dp} \right|_{\bar{p}} = [\mathbf{J}(\bar{\mathbf{z}}, \bar{p})]^{-1} \left. \frac{\partial \mathbf{h}}{\partial p} \right|_{\bar{p}}$$

Physically, the sensitivity translates into the rate of change of each state variable of the solution with respect to that parameter, in other words

$$\bar{\mathbf{z}}(\bar{p} + 1) \approx \bar{\mathbf{z}}(\bar{p}) + \mathbf{S}(\bar{p})$$

We demonstrate the use of `sensitivity.m` by computing  $\mathbf{S}$  for the converged model object  $A$ :

```
>> S = sensitivity(A, 'b')

naetemplate object "S"
Variables
-----
x : 0.26652
y : -0.26652
Parameters
-----
b : 2
```

and so the approximate solution for  $b = 3$  is computed using the  $A$  and  $S$  moduels:

```
>> xnew = get(A, 'var', 'x') + get(S, 'var', 'x')
xnew =
    1.5511

>> Ynew = get(A, 'var', 'y') + get(S, 'var', 'y')
Ynew =
   -0.8580
```

and so now we compare this solution estimate to the actual solution corresponding to  $b = 3$ :

```

>> A = set(A,'param',3,'b');
>> A = newton(A)
Update/residual norm = 0.3205 / 0.046952
...
Update/residual norm = 2.94e-13 / 4.2188e-14

naetemplate object "A"
  Variables
  -----
x : 1.5213
y : -0.82811
Parameters
  -----
b : 3

```

We conclude that the approximate and true solutions match reasonably well, especially given the linear nature of the approximation. We will find the sensitivity analysis a valuable tool for process optimization and analysis, and as a basis for the continuation procedures described next.

## 12.7 Continuation

Let us say that we have already determined a solution  $\bar{x}$ ,  $\bar{y}$  to (12.3) for some value of the parameter  $\bar{p}$ . If we now must compute a solution for another, relatively close value of  $p$ , we might consider computing a solution to

$$\mathbf{H}(\mathbf{z}) = \begin{bmatrix} \mathbf{h}(\mathbf{z}) \\ \|\mathbf{z} - \bar{\mathbf{z}}\|^2 \end{bmatrix} - \begin{bmatrix} \mathbf{0} \\ \delta_s^2 \end{bmatrix} = \mathbf{0} \quad \text{with} \quad \mathbf{z} = \begin{bmatrix} x \\ y \\ b \end{bmatrix}. \quad (12.7)$$

*Note how  $b$  has shifted its status from that of a parameter to a variable.* We can compute solutions to (12.7) with our Newton-Raphson method (12.5):

$$\mathbf{z}^1 = \mathbf{z}^0 - [\mathbf{J}_c]^{-1} \mathbf{H}(\mathbf{z}^0).$$

The Jacobian elements are found to be

$$\mathbf{J}_c = \begin{bmatrix} J_{1,1}^0 & J_{1,2}^0 & P_1^0 \\ J_{2,1}^0 & J_{2,2}^0 & P_2^0 \\ 2(x^0 - \bar{x}) & 2(y^0 - \bar{y}) & 2(b^0 - \bar{b}) \end{bmatrix}. \quad (12.8)$$

with

$$P_i = \frac{\partial h_i}{\partial b}.$$

There are at least two solutions to this set of equations, so we need to help out our algorithm with a good initial guess – that will be provided by a *predictor* step

$$\mathbf{z}^0 = \bar{\mathbf{z}} + \delta_s \left. \frac{d\mathbf{z}}{ds} \right|_{\bar{\mathbf{z}}}. \quad (12.9)$$

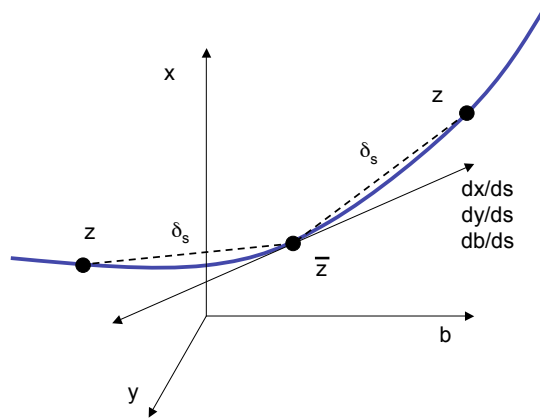


Figure 12.4: Numerical continuation of a solution to a nonlinear problem, illustrating the multiple solutions that exist corresponding to each direction of the arc.

The derivatives of  $\mathbf{z}$  are found by differentiating (12.7) with respect to  $s$  to give:

$$\mathbf{K}(\mathbf{dz}/ds) = \mathbf{J}_p \left[ \frac{d\mathbf{z}}{ds} \right] - \begin{bmatrix} 0 \\ 0 \\ 2\delta_s \end{bmatrix} = 0$$

with

$$\mathbf{J}_p = \begin{bmatrix} \bar{J}_{1,1} & \bar{J}_{1,2} & \bar{P}_1 \\ \bar{J}_{2,1} & \bar{J}_{2,2} & \bar{P}_2 \\ 2(x^0 - \bar{x}) & 2(y^0 - \bar{y}) & 2(b^0 - \bar{b}) \end{bmatrix}$$

and, by using (12.9):

$$= \begin{bmatrix} \vdots \\ 2\delta_s [dx/ds] & 2\delta_s [dy/ds] & 2\delta_s [db/ds] \end{bmatrix}.$$

Because this gives us a nonlinear set of equations for the derivatives, we linearize the system as part of the Newton procedure:

$$\mathbf{K}^0 + \mathbf{J}_p \left[ \frac{d\mathbf{z}}{ds} \right]^1 - \frac{d\mathbf{z}}{ds} \Big|_0 = 0$$

### 12.7.1 Computational example: Solutions by continuation

Continuing with the two-equation nonlinear model, we start the algorithm at  $b = 3$ . A small, negative change in  $b$  will have the opposite effect of the sensitivity we have already computed, therefore we anticipate the solution will change in the following approximate manner initially for a negative  $\delta_s$ :

$$\frac{dx}{ds} = -0.25 \quad \frac{dy}{ds} = 0.25 \quad \frac{db}{ds} = -0.6.$$

The initial state of the `naetemplate` object is used to compute the correct derivatives for the predictor step (12.9). The results of the predictor step then are used in the corrector step:

$$\mathbf{f}(\mathbf{z}^0) + \mathbf{J}_c^0(\mathbf{z}^1 - \mathbf{z}^0) = \mathbf{0}.$$

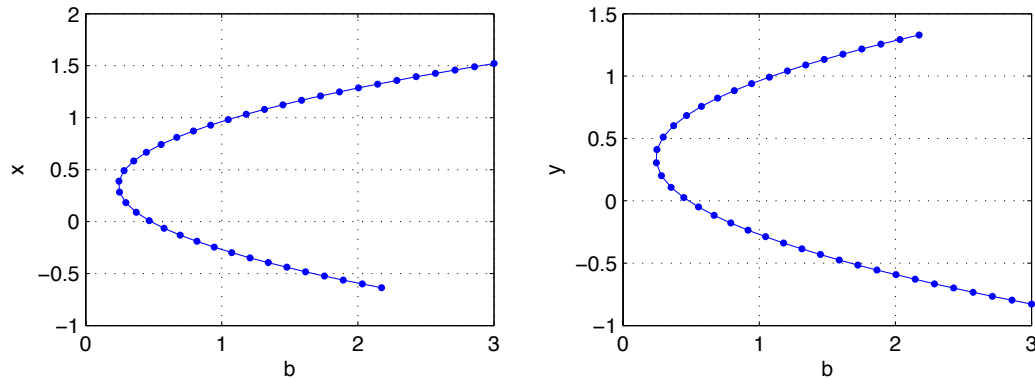


Figure 12.5: Solutions  $x$  and  $y$  as a function of  $b$  computed using a predictor-corrector continuation method.

using the Jacobian defined in (12.8). The MATLAB script to compute a vector of `naemodel`-derived objects is as follows:

```
ds = -0.15;
bifparam = 'b';
for i = 2:40
    if i == 2
        [Apred(i),dvds] = predictor(A(i-1),bifparam,ds);
    else
        [Apred(i),dvds] = predictor(A(i-1),bifparam,ds,dvds);
    end
    A(i) = newton(Apred(i),A(i-1),bifparam,ds); % the corrector step
end
```

Note that in prediction steps subsequent to the first that the previous derivatives are used to improve the speed of convergence of the predictor step. The results are plotted by extracting the parameter and corresponding solution values from the vector of `naetemplate` objects as shown below; results are plotted in Fig. 12.5 where the ability of the continuation algorithm to follow solution arcs around turning points is demonstrated.

```
bval = get(A(1),'param',bifparam);
xval = get(A(1),'var','x');
yval = get(A(1),'var','y');
for i = 2:length(A)
    bval(i) = get(A(i),'param',bifparam);
    xval(i) = get(A(i),'var','x');
    yval(i) = get(A(i),'var','y');
end
```

Note how the location of the turning point matches our “visual” observations made at the beginning of this chapter.



## 12.8 Review problems

1. Consider the problem of determining the roots of the function  $20 = 1 + 2x + 3x^2$ . For this homework problem, write a function that

- (a) plots the values of

$$f(x) = -19 + 2x + 3x^2$$

versus  $x$  over a specific interval  $A \leq x \leq B$  using a total of  $N$  points inside this interval; be sure to put a grid on the plot;

- (b) calling the grid points of this interval  $x_n, n = 1, \dots, N$ , add a Newton procedure to the function in which a solution to  $f(x) = 0$  is computed using each of the  $x_n$  as an initial guess to the Newton procedure;
- (c) plot the converged solution values as a function of the initial guess values  $x_n$  in this interval; report how many of the  $x_n$  converged to any valid solution (even if they are outside the interval).
- (d) write your function so that the input parameter list to the function consists of  $A$ ,  $B$ , and  $N$ ;
- (e) test your function for the cases

$A=0, B=1, N=10$   
 $A=0, B=10, N=100$   
 $A=-3, B=5, N=1000$   
 $A=-3, B=5, N=10000$   
 $A=30, B=5, N=100$

2. Recall how the heat capacity for many materials can be approximated by the polynomial curve fit

$$C_p(T) = a + bT + cT^2 + dT^3$$

where  $T$  is given in  $^{\circ}\text{C}$  and  $C_p$  in  $\text{J/mol}\cdot\text{K}$ . Data for several chemical compounds are given below; these curve fits are valid over the temperature range  $0\text{--}1200^{\circ}\text{C}$

Substance	$a$	$b$	$c$	$d$
acetone	71.96	$20.10 \times 10^{-2}$	$-12.78 \times 10^{-5}$	$34.76 \times 10^{-9}$
air	28.94	$0.4147 \times 10^{-2}$	$0.3191 \times 10^{-5}$	$-1.965 \times 10^{-9}$
methane	34.31	$5.496 \times 10^{-2}$	$0.3661 \times 10^{-5}$	$-11.00 \times 10^{-9}$

Given that the total amount of energy  $Q$  needed to raise one mole of each of these substances from  $T_0$  to  $T_1$  is

$$Q = \int_{T_0}^{T_1} C_p(T) dT$$

- (a) Integrate the equation above by hand and write a MATLAB function that takes as input the coefficients  $a$ ,  $b$ ,  $c$ , and  $d$ , the initial temperature  $T_0$ , and the value of  $Q$  and computes the corresponding final temperature  $T_1$  using a Newton procedure.
- (b) Given  $T_0 = 20^{\circ}\text{C}$  and  $Q = 10,000 \text{ J/mol}$ , compute the corresponding value for  $T_1$  for each substance.

3. Write a MATLAB script to find solutions to the model problem

$$\begin{aligned}f(x, y) &= \exp(x + y) - 2 \\g(x, y) &= -x^2 - y^2 + b\end{aligned}$$

using a Newton-Raphson technique where elements of the Jacobian array are computed using centered finite differences with a step size  $\delta = 1 \times 10^{-6}$ ; compute the value of both solutions for  $b = 1.5$ .

4. Solve the 8-equation, 7-unknown flash drum material balance below subject to  $z_M = 0.1$  and  $z_O = 0.9$  directly using a Newton-Raphson method.

$$\begin{aligned}x_M + x_O &= 1 \\y_M &= 220x_M \\y_O &= 0.5x_O \\y_M + y_O &= 1 \\V + L &= F \\y_M V + x_M L &= z_M F \\z_M F &= 10 \\y_O V + x_O L &= z_O F\end{aligned}$$

Prove your computed solution is a true solution to the modeling equations; is this an over-determined (least-squares) system, or is there a redundant equation?

## Chapter 13

# Case Study: A CVD reactor thermal dynamics model

Microelectronic devices are manufactured in a sequence of processing steps that include numerous thin-film deposition and etching operations. As one example, consider the reaction process where thin films of tungsten are deposited using a Chemical Vapor Deposition (CVD) system [21].



Figure 13.1: A two-chamber ULVAC tungsten CVD cluster tool.

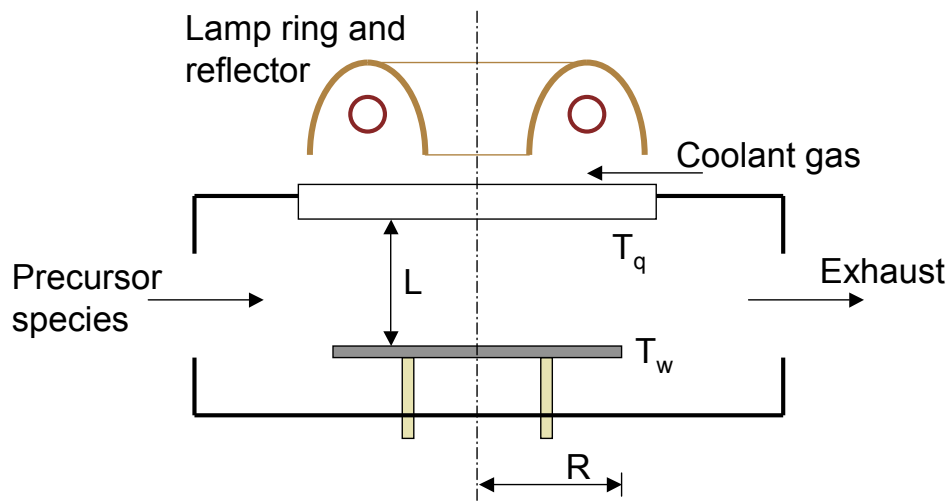


Figure 13.2: A schematic diagram of one of the ULVAC single-wafer cross flow chemical vapor deposition reaction chambers (bottom).

Wafer temperature control is critical to producing uniform films across the wafer as well as from wafer-to-wafer. To illustrate representative issues in reactor modeling, we develop a lumped dynamic wafer thermal model. Reactor gas phase temperature is assumed to be at quasi steady-state with respect to the wafer and showerhead thermal dynamics, with a further simplification in that we assume a linear temperature profile from the wafer to the cooled reactor wall. We develop a model describing the dynamics of the wafer temperature  $T_w$  and showerhead window temperature  $T_q$  under the following assumptions:

1. Wafer temperature  $T_w$  does not vary with  $z$  (thin wafer assumption); and for now we neglect any radial variations in temperature;
2. The process operates at sufficiently low deposition rates so that the contribution of heat of chemical reaction is negligible compared to the radiation emitted from the wafer and absorbed from the heating lamps; and
3. Radiation is emitted from both wafer sides and the wafer edge to a blackbody enclosure (the RTP chamber) maintained at  $T_a$  in this cold-walled process.
4. The wafer diameter is small compared to the quartz window, and so heat transfer by gas phase thermal conduction between the wafer and quartz window is considered important to the former, minor for the latter.

Table 13.1: Physical constant table (top) and RTP furnace design parameters (bottom).

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
$\rho_w$	2330 kg/m <sup>3</sup>	Si density
$C_{pw}$	2300 J/(kg K)	Si heat capacity
$\epsilon_w$	0.7	Si emissivity
$\rho_q$	2200 kg/m <sup>3</sup>	quartz density
$C_{pq}$	660 J/(kg K)	quartz heat capacity
$\epsilon_q$	0.5	quartz emissivity (high temperature)
$a$	0.01	quartz absorptivity
$\epsilon_s$	0.07	steel emissivity (room temperature)
$\sigma$	$5.670 \times 10^{-8}$ J/(K <sup>4</sup> m <sup>2</sup> s)	Stefan-Boltzmann constant
$T_a$	300 K	ambient temperature
$k$	0.3 W/(m K)	H <sub>2</sub> gas thermal conductivity
$h$	300 W/m <sup>2</sup>	showerhead/cooling gas heat transfer coeff
$R$	0.076 m	wafer radius
$L$	0.05 m	nominal wafer/quartz window distance
$Q$	2500 W/m <sup>2</sup>	lamp radiation flux
$\delta_w$	0.0005 m	wafer thickness
$\delta_q$	0.01 m	quartz window thickness

$$\begin{aligned}
\delta_w \frac{d\rho_w C_{pw} T_w^*}{dt} & \quad \text{accumulation of wafer thermal energy} \\
= & \quad \sigma F \epsilon_w \epsilon_q (T_q^{*4} - T_w^{*4}) \quad \text{wafer/showerhead radiation} \\
& + \sigma (1 - F) \epsilon_s \epsilon_q (T_a^4 - T_w^{*4}) \quad \text{wafer top/chamber heat transfer} \\
& + \sigma \epsilon_q \epsilon_s (T_a^4 - T_w^{*4}) \quad \text{wafer bottom/chamber heat transfer} \\
& + \frac{k}{L} (T_q^* - T_w^*) \quad \text{wafer/showerhead gas phase conduction} \\
& + (1 - a) Q u \quad \text{lamp heating of wafer} \\
\delta_q \frac{d\rho_q C_{pq} T_q^*}{dt} & \quad \text{accumulation of showerhead thermal energy} \\
= & \quad \sigma F \epsilon_w \epsilon_q (T_w^{*4} - T_q^{*4}) \quad \text{showerhead/wafer radiation} \\
& + \sigma (1 - F) \epsilon_s \epsilon_q (T_a^4 - T_q^{*4}) \quad \text{showerhead/chamber heat transfer} \\
& + h (T_a - T_q^*) \quad \text{showerhead coolant gas} \\
& + a Q u \quad \text{lamp heating of showerhead}
\end{aligned}$$

with the configuration factor given as

$$F = \frac{2 + \alpha^2 - \sqrt{(2 + \alpha^2)^2 - 4}}{2}$$

and  $\alpha = L/R$ . The dimensionless quantity  $u \in [0, 1]$  represents the percentage of full lamp power.

Table 13.2: Definition and numerical values of parameters in the CVD reactor model.

Parameter	Definition	Numerical value
$\epsilon_w^q$	$\tau \sigma \epsilon_w \epsilon_q T_a^4 / \rho_w C_{pw} \delta_w$	0.06
$\epsilon_w^s$	$\tau \sigma \epsilon_w \epsilon_s T_a^4 / \rho_w C_{pw} \delta_w$	0.0084
$\kappa_w$	$\tau k / R \rho_w C_{pw} \delta_w$	0.0015
$q_w$	$\tau (1 - a) Q / \rho_w C_{pw} \delta_w$	0.75
$\epsilon_q^w$	$\tau \sigma \epsilon_w \epsilon_q T_a^4 / \rho_q C_{pq} \delta_q$	0.011
$\epsilon_q^s$	$\tau \sigma \epsilon_s \epsilon_q T_a^4 / \rho_q C_{pq} \delta_q$	0.0011
$\nu_q$	$\tau h / \rho_q C_{pq} \delta_q$	0.021
$q_q$	$\tau a Q / \rho_q C_{pq} \delta_q$	0.034
$\alpha$	L/R	0.66
$F$		0.52

Defining the dimensionless temperature  $T$  and time  $t$  as

$$T = \frac{T^*}{T_a} \quad t = \frac{t^*}{\tau}$$

and assuming  $C_{pw}$ ,  $\rho_w$ ,  $C_{pq}$ , and  $\rho_q$  are not functions of temperature (a poor assumption, but necessary to keep the modeling equations as simple as possible), we obtain

$$\frac{dT_w}{dt} = \epsilon_w^q F (T_q^4 - T_w^4) + \epsilon_w^s (2 - F) (1 - T_w^4) + \frac{\kappa_w}{\alpha} (T_q - T_w) + q_w u \quad (13.1)$$

$$\frac{dT_q}{dt} = \epsilon_q^w F (T_w^4 - T_q^4) + \epsilon_q^s (1 - F) (1 - T_q^4) + \nu_q (1 - T_q) + q_q u \quad (13.2)$$

## 13.1 Steady-state solution using Newton's method

At steady state

$$\frac{dT_w}{dt} = \frac{dT_q}{dt} = 0$$

and  $u(t) = \text{constant}$  giving a set of two nonlinear algebraic equations to solve for the steady state showerhead and wafer temperatures. Consider the simplifying case where  $T_q$  is known and is fixed in time; the problem reduces to finding the steady-state solutions to

$$g(T_w) = \epsilon_w^q F (T_q^4 - T_w^4) + \epsilon_w^s (2 - F) (1 - T_w^4) + \frac{\kappa_w}{\alpha} (T_q - T_w) + q_w u = 0 \quad (13.3)$$

We now solve for solutions to  $g(T_w) = 0$  using the iterative Newton's method for a specified set of  $T_q$  and other parameters. If  $T_w^0$  is our guess for a solution, a more refined solution  $T_w^1$

$$T_w^1 = T_w^0 - \frac{g(T_w^0)}{dg/dT_w|_{T_w^0}}$$

with

$$\frac{dg}{dT_w} = -4\epsilon_w^q F T_w^3 - 4\epsilon_w^s (2 - F) T_w^3 - \frac{\kappa_w}{\alpha} \quad (13.4)$$

for this example. It is interesting to observe that because (13.4) is negative for all parameter values, the problem  $g(T_w) = 0$  must have a unique solution.

We consider the following portion of a MATLAB script written to solve the single-equation version of the simplified CVD thermal dynamics problem (13.3)

```
% single equation newton procedure applied the the CVD reactor
% wafer energy balance

model_param_script % set dimensionless parameter values for this model
Tq = 2; % fix value of Tq to 600K
Tw = 4; % initial solution guess
for iters = 1:8
    g = e_wq*F*(Tq^4-Tw^4) + e_ws*(2-F)*(1-Tw^4) ...
        + k_w/alf*(Tq-Tw) + q_w*u;
    dg = -4*e_wq*F*Tw^3 - 4*e_ws*(2-F)*Tw^3 - k_w/alf;
    update = -g/dg;
    Tw = Tw + update;
end
```

Given an initial guess of  $T_w^0 = 4$ , the updates converge with the following behavior to the solution  $T_w = 2.3160$  (695K) for  $u = 1$  (full lamp power)

Iteration i	Update	$T_w^i$
0	-	4
1	-8.8777e-01	3.1122
2	-5.3955e-01	2.5727
3	-2.2076e-01	2.3519
4	-3.5076e-02	2.3168
5	-8.1187e-04	2.3160
6	-4.2666e-07	2.3160
7	-1.1770e-13	2.3160
8	5.0929e-17	2.3160
9	5.0929e-17	2.3160

and so the exponent of the update indeed nearly doubles during each iteration demonstrating quadratic convergence until the round-off limit of  $1 \times 10^{-16}$ . A graphical view of the Newton iterations is shown in Fig. 13.3, where the red-line segments are plots of the linear function  $f(x)$  while the blue curve is  $g(x)$ . For this example, the parameters are computed using the script `model_param_script.m`:

```

% model_param_script - define dimensionless parameter values for the
%                       CVD lumped thermal dynamics model
rho_w = 2330;
Cp_w = 2300;
ew = 0.7;
...
e_qs = tau*sig*eq*es*Ta^4/(rho_q*Cp_q*delq)
nu_q = tau*h/(rho_q*Cp_q*delq)
q_q = tau*a*Q/(rho_q*Cp_q*delq)

```

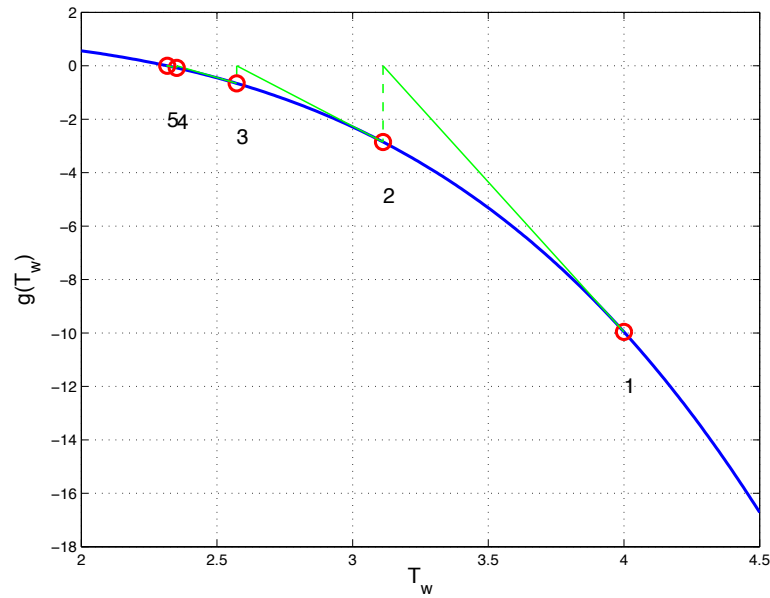


Figure 13.3: Convergence of the Newton scheme.

## 13.2 Multiple equations: The Newton-Raphson method

Returning to the CVD reactor thermal modeling equations (13.1, 13.2), at steady state and written in vector form

$$\mathbf{0} = \mathbf{g}(\mathbf{z}) \quad \text{with} \quad \mathbf{z} = \begin{bmatrix} T_w \\ T_q \end{bmatrix}$$

and

$$\begin{aligned}
 g_1(T_w, T_q) &= \epsilon_w^q F(T_q^4 - T_w^4) + \epsilon_w^s (2 - F)(1 - T_w^4) + \frac{\kappa_w}{\alpha} (T_q - T_w) + q_w u \\
 g_2(T_w, T_q) &= \epsilon_q^w F(T_w^4 - T_q^4) + \epsilon_q^s (1 - F)(1 - T_q^4) + \nu_q (1 - T_q) + q_q u
 \end{aligned}$$



For the CVD reactor lumped thermal model, the Jacobian matrix  $\mathbf{J}$  is found to be

$$\mathbf{J} = \begin{bmatrix} J_{1,1} & J_{1,2} \\ J_{2,1} & J_{2,2} \end{bmatrix} = \begin{bmatrix} -4\epsilon_w^q F T_w^3 - 4\epsilon_w^s (2-F) T_w^3 - \frac{\kappa_w}{\alpha} & 4\epsilon_w^q F T_q^3 + \frac{\kappa_w}{\alpha} \\ 4\epsilon_q^w F T_w^3 & -4\epsilon_q^w T_q^3 - 4\epsilon_q^s (1-F) T_q^3 - \nu_q \end{bmatrix}.$$

We now consider solving both the wafer and quartz window steady state energy balance equations to compute  $T_w$  and  $T_q$  for full lamp power  $u = 1$  using the script newton2eqn.m:

```
% Newton-Raphson procedure applied the the CVD reactor
% wafer energy balance

model_param_script % set dimensionless parameter values for this model
Tw = 2; % initial solution guess
Tq = 2; % initial solution guess
u = 1;

for iters = 1:8
    g1 = e_wq*F*(Tq^4-Tw^4) + e_ws*(2-F)*(1-Tw^4) ...
        + k_w/alf*(Tq-Tw) + q_w*u;
    g2 = e_qw*F*(Tw^4-Tq^4) + e_qs*(1-F)*(1-Tq^4) ...
        + nu_q*(1-Tq) + q_q*u;
    J11 = -4*e_wq*F*Tw^3 - 4*e_ws*(2-F)*Tw^3 - k_w/alf;
    J12 = 4*e_wq*F*Tq^3 + k_w/alf;
    J21 = 4*e_qw*F*Tw^3;
    J22 = -4*e_qw*F*Tq^3 - 4*e_qs*(1-F)*Tq^3 - nu_q;

    g = [g1; g2];
    J = [J11 J12; J21 J22];
    update = -J\g;
    normup = sqrt(update'*update)
    Tw = Tw + update(1)
    Tq = Tq + update(2)
end
```

For this case we compute the length (2-norm) of the update, and again observe the quadratic convergence to the solution  $T_w = 2.67$  (802K) and  $T_q = 2.62$  (786K):

Iteration i	Update	$T_w^i$	$T_q^i$
0	-	2	2
1	1.3652e+00	3.0365	2.8885
2	3.8702e-01	2.7310	2.6509
3	6.4840e-02	2.6745	2.6191
4	1.8667e-03	2.6727	2.6186
5	1.7971e-06	2.6727	2.6186
6	1.7987e-12	2.6727	2.6186
7	1.2215e-16	2.6727	2.6186
8	1.2215e-16	2.6727	2.6186

### 13.3 CVD thermal model derived from the neqmodel class

To use our object-oriented approach to model building, we define the class `cvdtherm2eqn` by creating the directory `cvdtherm2eqn` and in it, defining the constructor method `cvdtherm2eqn.m` in derived from the `naemodel` class, and overloading its `rhs.m` method.

Inside the constructor method `cvdtherm2eqn.m`, the problem variables and parameters are defined and the *state* of the object created using this constructor method is initialized to the initial solution estimates and parameter values. Note how the constructor method takes a lamp power level  $u$  as input:

```
function B = cvdtherm2eqn(u)
% cvdtherm2eqn.m A nonlinear equation model object constructor, derived
%               from the neqmodel class. Called as
%
%               A = cvdtherm2eqn

model_param_script % set dimensionless parameter values for this model
Tw = 2; Tq = 2;
if nargin < 1 % default power input
    u = 1;
end

var = assocarray({'Tw' Tw 'Tq' Tq});
param = assocarray({ 'e_wq' e_wq 'e_ws' e_ws 'k_w' k_w ...
                    'q_w' q_w 'e_qw' e_qw 'e_qs' e_qs ...
                    'nu_q' nu_q 'q_q' q_q 'u' u ...
                    'F' F 'alf' alf ...
                    });
A = naemodel(var,param);
B = struct([]);
B = class(B,'cvdtherm2eqn',A);
```

In the `residual.m` method, the “right-hand sides” of the modeling equations are defined; in computing the steady-state solutions, these equations define the residual values that are to be driven to zero. Inside the `residual.m` method, the variables and parameter values are unpacked from the modeling object and are placed in the workspace of the `residual.m` function. The residual functions then are evaluated and the residual values, stored as a MATLAB cell array, are used to update the modeling object `resid` field.

```
function g = rhs(A)
% rhs.m Nonlinear equation model object residual update.
%
% INPUT/OUTPUT PARAMETERS
%     A : an object derived from the naemodel class

unpack(A)
g{1} = e_wq*F*(Tq^4-Tw^4) + e_ws*(2-F)*(1-Tw^4) ...
      + k_w/alf*(Tq-Tw) + q_w*u;
g{2} = e_qw*F*(Tw^4-Tq^4) + e_qs*(1-F)*(1-Tq^4) ...
      + nu_q*(1-Tq) + q_q*u;
```

And now we interactively solve the problem in a MATLAB session:

```
>> A = cvdtherm2eqn;
>> A = newton(A);
Update/residual norm = 0.84874 / 0.053809
Update/residual norm = 0.071781 / 0.0024428
Update/residual norm = 0.00071572 / 0.00011191
Update/residual norm = 3.2778e-05 / 5.0809e-06
Update/residual norm = 1.4882e-06 / 2.3078e-07
Update/residual norm = 6.7596e-08 / 1.0482e-08
Update/residual norm = 3.0702e-09 / 4.7609e-10
Update/residual norm = 1.3945e-10 / 2.1625e-11
Update/residual norm = 6.3339e-12 / 9.8171e-13
Update/residual norm = 2.8746e-13 / 4.5271e-14
```

again observing the slightly reduced (less than quadratic, but not much) convergence rate.

At this point, the modeling object `A` has stored in its `var` field the `assocarray` object containing the variable values corresponding to the converged solution. We can view the solution values using the `naemodel/display.m` method:

```
>> A
cvdtherm2eqn object "A"

Variables
-----
Tw:
    2.6727
Tq:
    2.6186

Parameters
-----
e_wq:
    0.0600
e_ws:
    0.0084
...
```

We can unpack the parameters and variables from the converged model object, placing copies in the current workspace and translating them into dimensional quantities:

```
>> unpack(A)
>> 300*Tw
ans =
    801.8108

>> 300*Tq
ans =
    785.5657
```

## 13.4 Continuation

We can use our numerical continuation technique to effectively study the behavior of the reactor system as a function of power input  $u$  over a range of values.

We start the algorithm at  $u = 0$  (lamps off) so that  $T_w = 1$ , and  $T_q = 1$  (recall that an exact solution is not necessary to begin the continuation - in fact, only a good initial guess is needed to start the Newton-Raphson procedure to find the initial point). The MATLAB script to compute a vector of `naemodel`-derived objects is as follows:

```
A = cvdtherm2eqn(0); % note lamp power setting
A = newton(A);

ds = 0.5;
bifparam = 'u';
for i = 2:8
    if i == 2
        [Apred(i),dvds] = predictor(A(i-1),bifparam,ds);
    else
        [Apred(i),dvds] = predictor(A(i-1),bifparam,ds,dvds);
    end
    A(i) = newton(Apred(i),A(i-1),bifparam,ds); % the corrector step
end
```

The results are plotted by extracting the parameter and corresponding solution values from the vector of `cvdtherm2eqn`; results are plotted in Fig. 13.4 where the accuracy of the predictor steps is demonstrated.

## 13.5 Linearized system dynamics

As a final analysis of the reactor system, we examine its dynamic behavior under small perturbations. This is accomplished by linearizing the system at a converged steady state solution, computing the eigenvalues and eigenvectors of the Jacobian array, and superimposing a set of particular solutions to the linear ordinary differential equations.

We can write the Taylor's series expansion of  $\mathbf{g}$  at  $(\bar{T}_w, \bar{T}_q)$  as

$$\begin{aligned} dT_w/dt &= g_1(\bar{T}_w, \bar{T}_q) + \left. \frac{\partial g_1}{\partial T_w} \right|_{\bar{T}_w, \bar{T}_q} (T_w - \bar{T}_w) + \left. \frac{\partial g_1}{\partial T_q} \right|_{\bar{T}_w, \bar{T}_q} (T_q - \bar{T}_q) \\ dT_q/dt &= g_2(\bar{T}_w, \bar{T}_q) + \left. \frac{\partial g_2}{\partial T_w} \right|_{\bar{T}_w, \bar{T}_q} (T_w - \bar{T}_w) + \left. \frac{\partial g_2}{\partial T_q} \right|_{\bar{T}_w, \bar{T}_q} (T_q - \bar{T}_q) \end{aligned}$$

or, neglecting the second and higher-order terms

$$dz/dt = \mathbf{J}(\mathbf{z} - \bar{\mathbf{z}}).$$

We consider the case of the system operated at full power  $u = 1$ ; we construct a model object `A`, determine the converged solution, and then use the `jacobian.m` method of the `naemodel` class:

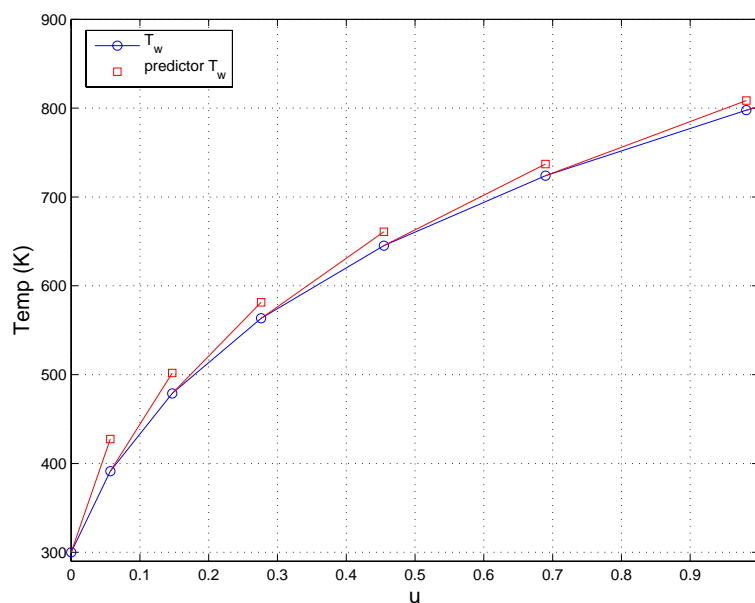


Figure 13.4: Predictor (red squares) / corrector (blue circles) continuation results for the two-equation CVD thermal model showing the wafer temperature as a function of lamp power input.

```
>> A = cvdtherm2eqn;
>> A = newton(A);
>> J = jacobian(A);
>> [U,S] = eig(J)
```

```
U =
   -0.9905   -0.5782
    0.1376   -0.8159

S =
   -3.6628         0
         0   -0.1611
```

We see that one eigenvalue is much larger in magnitude than the other; the two time scales are attributable to the large differences in thermal mass of the wafer and shower head. We show this by plotting the eigenvectors and the solution trajectories; results are shown in Fig. 13.5.

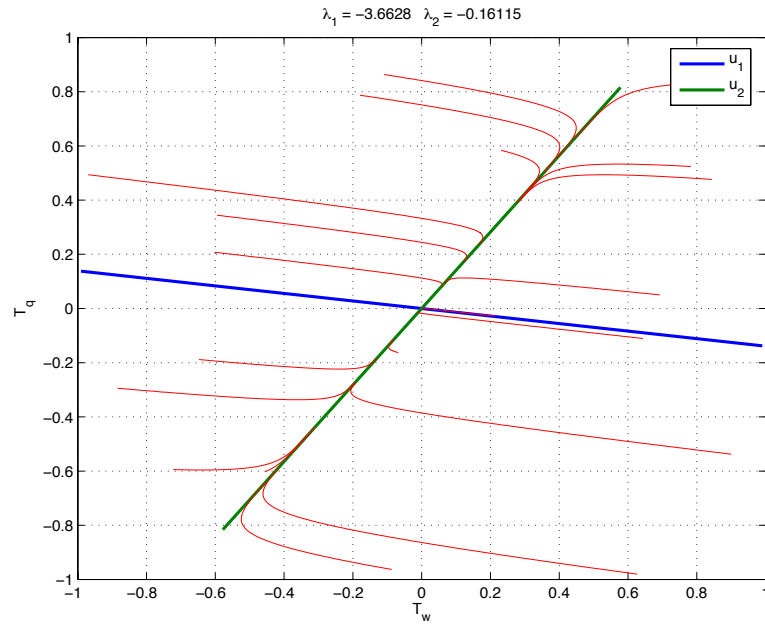


Figure 13.5: The linearized dynamics of the 2 ODE CVD reactor thermal dynamics illustrating the relatively slow dynamics of the showerhead compared to the much thinner wafer.

```
figure(1), clf
plot([-U(1,1) U(1,1)],[-U(2,1) U(2,1)], ...
      [-U(1,2) U(1,2)],[-U(2,2) U(2,2)], 'Linewidth',2)
legend('u_1','u_2')
xlabel('T_w'), ylabel('T_q')
grid on
title(['\lambda_1 = ',num2str(S(1,1)), ' \lambda_2 = ',num2str(S(2,2))])

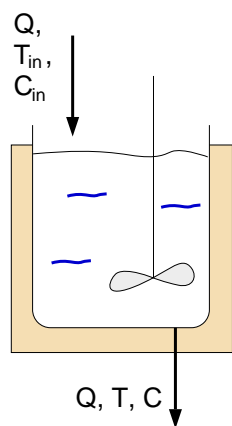
t = [0:0.005:2];
hold on
for i = 1:20
    z0 = 2*rand(2,1)-[1;1];
    z = lodesolver(J,t,z0);
    plot(z(1,:),z(2,:), 'r')
end
hold off
```

## Chapter 14

# Case Study: Adiabatic CSTR with first-order reaction

Consider the well-insulated (adiabatic) Continuous-flow Stirred-Tank Reactor (CSTR) where the single, exothermic, mole-conserving reaction  $A \rightarrow B$  takes place. The material and energy balances on the reactor give

$$\begin{aligned} \text{Quantity In} - \text{Quantity Out} &= \text{Quantity Consumed} + \text{Accumulated} \\ QC_{in} - QC &= Vk_0 e^{-E_a/RT} C + V \frac{dC}{dt} \\ Q\rho C_p(T_{in} - T_{ref}) - Q\rho C_p(T - T_{ref}) &= -V\Delta_{H_R} k_0 e^{-E_a/RT} C + V\rho C_p \frac{dT}{dt} \end{aligned}$$



Parameter	Description	Units	Value
$C_{in}$	inlet conc	kgmol/m <sup>3</sup>	3.0
$Q$	feed/effluent flow rate	m <sup>3</sup> /s	$6 \times 10^{-5}$
$\rho$	liquid density	kg/m <sup>3</sup>	1000
$C_p$	heat capacity	J/kg K	4190
$V$	reactor volume	m <sup>3</sup>	$1.8 \times 10^{-2}$
$\Delta_{H_R}$	exothermic heat of rxn	J/kgmole	$2.09 \times 10^8$
$k_0$	preexp constant	1/s	$4.48 \times 10^6$
$E_a$	activation energy	J/gmol	62,800
$R$	gas constant	J/gmole/K	8.314
$T_{in}$	inlet temp	K	300

Figure 14.1: Adiabatic CSTR with representative parameter values.

Defining the dimensionless variables and constants

$$x = \frac{C}{C_{in}}, \quad y = \frac{T}{T_{in}}, \quad \tau = \frac{Q}{V}t, \quad \gamma = \frac{E_a}{RT_{in}}, \quad \alpha = \frac{Vk_0}{Q}, \quad \beta = \frac{\Delta_{H_R} C_{in}}{\rho C_p T_{in}}$$

with data taken from [24] and presented in Fig. 14.1, we find that the modeling equations can be written in much simpler form

$$\begin{aligned}\frac{dx}{d\tau} &= 1 - x - \alpha x e^{-\gamma/y} \\ \frac{dy}{d\tau} &= 1 - y + \alpha \beta x e^{-\gamma/y}\end{aligned}$$

with

$$\gamma = 25.2, \quad \alpha = 1.344 \times 10^9, \quad \beta = 0.5$$

and subject to specified initial conditions.

Solutions of the set of coupled nonlinear equations

$$1 - x - \alpha x e^{-\gamma/y} = 0 \quad (14.1)$$

$$1 - y + \alpha \beta x e^{-\gamma/y} = 0 \quad (14.2)$$

define the steady-state solutions to this CSTR model. While not generally possible for sets of nonlinear equations, (14.1)-(14.2) can be simplified to a set of two equivalent (one nonlinear and one linear) equations with the property that the nonlinear equation is decoupled from the second equation; furthermore, because the second equation is linear, it can be solved exactly given the solution to the first. This transformation effectively performs the equivalent of the forward elimination factorization to the set of nonlinear equations.

This transformation is performed by multiplying (14.1) by  $\beta$  and subsequently subtracting (14.2) to find

$$\beta(1 - x) + 1 - y = 0 \quad \text{or} \quad y = \beta(1 - x) + 1.$$

This function can be substituted into the material balance equation (14.1) to find

$$1 - x - \alpha x \exp \left\{ \frac{-\gamma}{\beta(1 - x) + 1} \right\} = g(x) = 0 \quad (14.3)$$

The steady-state solutions to the CSTR model are now defined in terms of the roots of the nonlinear equation  $g(x) = 0$ ; because this is a nonlinear equation, it is possible there are no, one, several, or infinite solutions. It is possible to gain some insight into the behavior of  $g(x) = 0$  by considering how the solution(s) change as  $\alpha$  is varied; because  $\alpha$  is inversely proportional to  $Q$  at steady state, and by using (14.3), we can see that

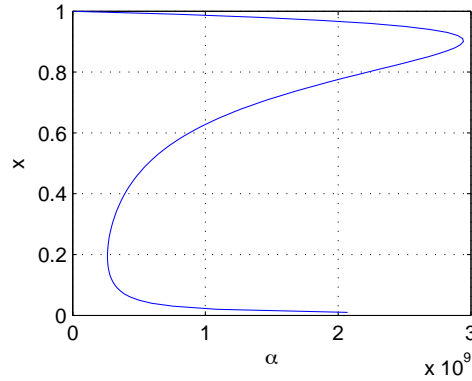
$$\begin{aligned}\alpha \rightarrow 0 \ (Q \rightarrow \infty) &\implies x \rightarrow 1 \quad \text{no conversion of reactant} \\ \alpha \rightarrow \infty \ (Q \rightarrow 0) &\implies x \rightarrow 0 \quad \text{complete conversion}\end{aligned}$$

It is possible to take advantage of another unique property of (14.3) to determine all possible solutions to (14.3): while it is not possible to immediately assess the number of solutions to (14.3), one method for computing all solutions is to switch the solution procedure around: instead of computing  $x$  for some  $\alpha$ , compute  $\alpha$  for some  $x$ :

$$\alpha = \frac{1 - x}{x} \exp \left\{ \frac{\gamma}{\beta(1 - x) + 1} \right\}. \quad (14.4)$$

This takes advantage of the range of  $x$  being limited to the unit interval. We then plot  $x$  as a function of  $\alpha$  (Fig. 14.2).



Figure 14.2: Steady-states for the adiabatic CSTR as a function of  $\alpha$ .

## 14.1 Turning-point bifurcations

It is interesting to consider the solutions which satisfy the modeling equation (14.3) and that make the first derivative condition (14.5) vanish.

$$\frac{dg}{dx} = -1 - \alpha \exp \left\{ \frac{-\gamma}{\beta(1-x) + 1} \right\} - \alpha x \exp \left\{ \frac{-\gamma}{\beta(1-x) + 1} \right\} \left[ \frac{-\gamma\beta}{[\beta(1-x) + 1]^2} \right] \quad (14.5)$$

This gives two equations from which potentially either a variable or a parameter can be eliminated. For example, by using the modeling equation in the form (14.4), if we set the derivative computed in (14.5) equal to zero and substitute in the equation for  $\alpha$ , we find the zero-derivatives can be calculated from the algebraic equation

$$\begin{aligned} -1 + \frac{x-1}{x} + (x-1) \left[ \frac{-\gamma\beta}{[(1-x)\beta + 1]^2} \right] &= 0 \\ -x[(1-x)\beta + 1]^2 + (x-1)[(1-x)\beta + 1]^2 - \gamma\beta(x-1)x &= 0 \\ (1-x)^2\beta^2 + 2(1-x)\beta + 1 + \gamma\beta(x^2 - x) &= 0 \\ (\gamma\beta + \beta^2)x^2 - (2\beta^2 + 2\beta + \gamma\beta)x + \beta^2 + 2\beta + 1 &= 0 \end{aligned}$$

so

$$x = \frac{\beta(2\beta + 2 + \gamma) \pm \sqrt{\beta^2(2\beta + 2 + \gamma)^2 - 4\beta(\gamma + \beta)(\beta + 1)^2}}{2\beta(\gamma + \beta)} = 0.9035, 0.1938.$$

Compare these points to the values observed in left figure of Fig. 14.2. We note that multiple solutions will not be found when

$$\beta^2(2\beta + 2 + \gamma)^2 - 4\beta(\gamma + \beta)(\beta + 1)^2 < 0.$$

This illustrates an important characteristic of nonlinear systems: that multiple solutions can exist and can correspond to true physical solutions – for the adiabatic CSTR, only two of these are normally seen due to the saddle-type stability of the intermediate state (the dynamics of the system near the steady states will be discussed in the next chapter). We note that the Newton procedure can compute all of the steady states, regardless of their stability characteristics.

## 14.2 Multistability

As a first step in understanding the dynamical behavior of *nonlinear* systems, let us put some of the facts we already understand together: if we calculate the three steady-state solutions  $\mathbf{z}_{si}$ ,  $i = 1, 2, 3$  of the nonadiabatic CSTR model

$$\frac{d\mathbf{z}}{d\tau} = \mathbf{g}(\mathbf{z})$$

for the parameter values given previously, and linearize the modeling equations at these solutions, the three steady-state solutions and the corresponding parameter values for the ODE solutions valid in the neighborhood of the steady solution points are given in the following table.

Variable	Low Conv	Medium Conv	High Conv
$x_s$	0.9810	0.6902	0.0158
$y_s$	1.0095	1.1549	1.4921
$\lambda_1, \lambda_2$	-1.00, -0.78	-1.00, 1.47	-57.59, -1.00
$\mathbf{U}$	$\begin{bmatrix} -0.99 & 0.89 \\ 0.04 & -0.44 \end{bmatrix}$	$\begin{bmatrix} -0.99 & 0.89 \\ 0.07 & -0.44 \end{bmatrix}$	$\begin{bmatrix} -0.89 & 0.17 \\ 0.44 & -0.98 \end{bmatrix}$

At this point, we make the following observations:

1. The three steady states demonstrate different aspects of stability and instability – the upper and lower steady states attracting all initial conditions, and the middle demonstrating saddle stability, where only initial conditions on the “stable eigenvector” reach the steady state;
2. The phase space is organized by the saddle-unstable manifolds.

The issues of stability raise the question of how one determines which initial conditions are attracted to each of the stable steady states. This requires numerical integration of the fully nonlinear system. Specifically, we compute numerically the *invariant manifolds* which form the skeleton organizing the dynamics in the phase space.

**Stable Manifold  $W^s$ :** Points, which when followed forward in time, asymptotically approach the fixed point –  $W^s$  is tangent to the space spanned by the stable eigenvectors of the fixed point.

**Unstable Manifold  $W^u$ :** Points, which when followed in reverse time, asymptotically approach the fixed point –  $W^u$  is tangent to the space spanned by the unstable eigenvectors of the fixed point.

## 14.3 Review problems

1. For the adiabatic CSTR modeling equations, using a Newton-Raphson method, solve for the steady-states of the 2-equation problem for  $\gamma = 25.2$ ,  $\beta = 0.5$ , and  $\alpha = 1.344 \times 10^9 * [0.05 : 0.1 : 4]$ ; plot the roots in  $x$  vs.  $\alpha$  and  $y$  vs.  $\alpha$  plots; compute the eigenvalues of the converged Jacobian arrays to assess the stability of each solution, and in the plots, denote stable solutions by green o markers, unstable by red \* markers.

## Chapter 15

# Case Study: Flash distillation for a non-ideal mixture

Following the description of Doherty and Malone [11], we consider a case where a binary system of components is in vapor-liquid equilibrium where the vapor phase is considered to be an ideal gas and the liquid is a nonideal mixture. Expressing the equilibrium relationship in terms of liquid-phase activity coefficients

$$Py_i = P_i^{sat} \gamma_i x_i \quad i = 1, 2$$

where the notation is defined in Fig. 15.1.

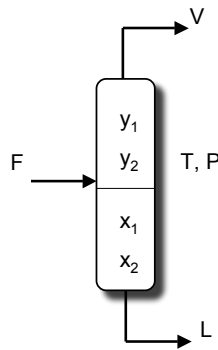


Figure 15.1: The single-stage separator.

We consider the two components to be methanol (component 1) and water (component 2); for this system, we can use the Wilson equation to define the activity coefficients:

$$\begin{aligned} \ln \gamma_1 &= -\ln(x_1 + \Lambda_{12}x_2) + x_2 \left( \frac{\Lambda_{12}}{x_1 + \Lambda_{12}x_2} - \frac{\Lambda_{21}}{\Lambda_{21}x_1 + x_2} \right) \\ \ln \gamma_2 &= -\ln(x_2 + \Lambda_{21}x_1) - x_1 \left( \frac{\Lambda_{12}}{x_1 + \Lambda_{12}x_2} - \frac{\Lambda_{21}}{\Lambda_{21}x_1 + x_2} \right) \end{aligned}$$

with

$$\Lambda_{12} = \frac{v_2^L}{v_1^L} \exp(-A_{12}/RT)$$

$$\Lambda_{21} = \frac{v_1^L}{v_2^L} \exp(-A_{21}/RT)$$

where  $v_i^L$  is the pure component molar volume of component  $i$ . More information can be found in [11]; parameter values are listed in the MATLAB functions below.

## 15.1 The variables and modeling equations

Our system consists of the six variables:

$$\text{variables: } x_m, y_m, x_w, y_w, T, P$$

and the four equations - two mole fraction definitions:

$$0 = x_m + x_w - 1$$

$$0 = y_m + y_w - 1$$

and the two phase relationships

$$0 = Py_m - P_m^{sat} \gamma_m x_m$$

$$0 = Py_w - P_w^{sat} \gamma_w x_w$$

This leaves two degrees of freedom that must be specified in order to find a unique solution to our problem (validate this with the Gibbs phase rule).

## 15.2 Definition of the model class meohwaterwilson

First, let us examine the constructor method of this new model class, which is derived from our naemodel class:

```

ffunction B = meohwaterwilson(T,P)

% meohwaterwilson.m A nonlinear equation model object constructor, derived
%                     from the neqmodel class. Called as
%
%                     A = meohwaterwilson(T,P)
%
%                     where P = total pressure (atm), T = temp (K)

A12 = 19.2547;
A21 = 554.0494;
vLm = 40.73;
vLw = 18.07;
R = 1.98721;

% Antoine eqn coeffs, from NIST webbook
Am = 5.20409; % T in K, P in bar; valid from 288K to 357K
Bm = 1581.341;
Cm = -33.50;
Aw = 4.65430; % valid from 56K to 373K
Bw = 1435.264;
Cw = -64.848;

xm = 0.5; ym = 0.5; % variables and initial guesses
xw = 0.5; yw = 0.5;
var = assocarray({'xm' xm 'ym' ym 'xw' xw 'yw' yw});
param = assocarray({'T' T 'P' P 'A12' A12 'A21' A21 ...
                    'vLm' vLm 'vLw' vLw 'R' R ...
                    'Am' Am 'Bm' Bm 'Cm' Cm ...
                    'Aw' Aw 'Bw' Bw 'Cw' Cw });

A = naemodel(var,param);
B = class(struct([]),'meohwaterwilson',A);

```

The residual method of this class computes the vapor pressure of the two pure components at the current value of temperature  $T$

```

function res = residual(A)

% residual.m Nonlinear equation model object residual update.
%
% INPUT/OUTPUT PARAMETERS
%      A : an object derived from the neqmodel class

unpack(A)

% MeOH = 1   Water = 2

% saturated liquid vapor pressure, bar

Psatm = 10^(Am - Bm/(T+Cm) );
Psatw = 10^(Aw - Bw/(T+Cw) );

% Convert from bar to atm

Psatm = 0.986923267*Psatm;
Psatw = 0.986923267*Psatw;

Lam12 = vLw/vLm*exp(-A12/(R*T));
Lam21 = vLm/vLw*exp(-A21/(R*T));

lng1 = -log(xm+Lam12*xw) ...
        + xw*( Lam12/(xm+Lam12*xw) - Lam21/(xw+Lam21*xm) );
lng2 = -log(xw+Lam21*xm) ...
        - xm*( Lam12/(xm+Lam12*xw) - Lam21/(xw+Lam21*xm) );

gm = exp(lng1);
gw = exp(lng2);

res{1} = 1 - xm - xw;
res{2} = 1 - ym - yw;
res{3} = ym - Psatm*gm*xm/P;
res{4} = yw - Psatw*gw*xw/P;

```

## 15.3 Representative calculations

### 15.3.1 Computing phase compositions for fixed temperature and pressure

```

P = 1;          % atm
T = 273+90; % K
A = meohwaterwilson(T,P);
A = newton(A)

```

After the newton procedure converges, we find the physically reasonable values of

$$\begin{aligned}x_m &: 0.07887 \\y_m &: 0.36594 \\x_w &: 0.92113 \\y_w &: 0.63406\end{aligned}$$

### 15.3.2 Flash drum material balance

In this problem, we use the results of the previous computation and define a flash drum feed composition and flowrate as below

```
zm = 0.2; % feed stream composition
F = 1; % feed basis, mole/min
zw = 1-zm;
```

and extract the composition data from the converged model object; the composition information then is used with a species material balance on the flash drum:

```
xm = get(A,'var','xm')
ym = get(A,'var','ym')
xw = get(A,'var','xw')
yw = get(A,'var','yw')

lv = [xm ym; xw yw] \ [F*zm; F*zw];
L = lv(1)
V = lv(2)
```

to find the following vapor and liquid product stream molar flowrates:

$$V = 0.4220 \quad L = 0.5780$$

### 15.3.3 Computing temperature for specified liquid composition

We can now specify a liquid phase composition of one component (e.g.,  $x_m$ ) and compute the corresponding flash drum temperature  $T$ ; this can be used to create a  $y$  vs.  $x$  composition plot. We first redefine  $T$  from a parameter to a variable and switch  $x_m$  from variable status to a specified parameter:

```
A = param2var(A,'T');
A = var2param(A,'xm');
```

Then, we step through a loop to compute the corresponding gas phase composition and drum temperature as  $x_m$  is varied over its entire range; results are plotted in Fig. 15.2.

```

xm = [0:0.05:1];
for i = 1:length(xm)
    A = set(A,'param',xm(i),'xm');
    A = newton(A);
    ym(i) = get(A,'var','ym');
    T(i) = get(A,'var','T');
end

figure(1), clf
subplot(1,2,1)
plot(xm,ym,'-o')
grid on
xlabel('x_m'), ylabel('y_m')
subplot(1,2,2)
plot(xm,T,'-o')
grid on
xlabel('x_m'), ylabel('T (K)')

```

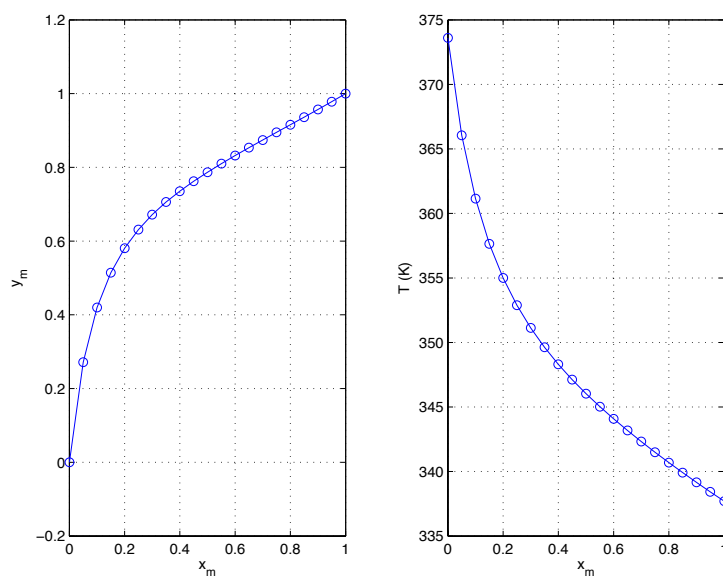


Figure 15.2:  $y$  vs.  $x$  plot for the methanol/water system (left) and the corresponding temperatures (right)



## Chapter 16

# Nonlinear ODE methods

Consider the single differential equation mode:

$$\frac{dx}{dt} = g(x) = ax - b$$

with  $a = -0.5$  and  $b = 1$ , subject to the initial condition  $x(t = 0) = 5$ . We can immediately see that the steady state solution is  $x_{st.st.} = -2$  and it is stable because of the negative eigenvalue  $\lambda = a = -0.5$ . The corresponding particular solution is:

$$x(t) = -2 + 7e^{-0.5t}$$

We wish to compare the exact solution computed over  $0 \leq t \leq 10$  to the solution approximated with the (forward) Euler method:

$$\begin{aligned} \left. \frac{dx}{dt} \right|_{t=t_n} &= g(x_n) && \text{(exact)} \\ &\approx \frac{x_{n+1} - x_n}{t_{n+1} - t_n} \\ \text{and so} \quad x_{n+1} &= x_n + (t_{n+1} - t_n)g(x_n) \\ &= x_n + \Delta_t g(x_n). \end{aligned}$$

If we consider  $x_n = x(t_n)$  to be a point on the exact solution to the differential equation, we can write the Taylor's series approximation to the time-dependent behavior of  $x$  (the exact solution) in the neighborhood of  $x_n$  as

$$\begin{aligned} x(t) &= x(t_n) + \left. \frac{dx}{dt} \right|_{t_n} (t - t_n) + \frac{1}{2} \left. \frac{d^2x}{dt^2} \right|_{t_n} (t - t_n)^2 + \text{h.o.t.} \\ x(t_{n+1}) &= x(t_n) + \Delta_t \left. \frac{dx}{dt} \right|_{t_n} + \frac{1}{2} \Delta_t^2 \left. \frac{d^2x}{dt^2} \right|_{t_n} + \text{h.o.t.} \\ &= x(t_n) + \Delta_t g(x_n) + \frac{1}{2} \Delta_t^2 \left. \frac{dg(x)}{dt} \right|_{t_n} + \text{h.o.t.} \end{aligned}$$

and so subtracting the results from one step of the Euler integrator and retaining only terms of order  $\Delta_t^2$  or less gives an estimate of the error  $d_{n+1}$  generated by one step of the Euler integrator:

$$\begin{aligned}
 d_{n+1} &= x(t_{n+1}) - x_{n+1} \\
 &= x(t_n) + \Delta_t g(x_n) + \frac{1}{2} \Delta_t^2 \frac{dg(x)}{dt} \Big|_{t_n} - x_n - \Delta_t g(x_n) \\
 &= \frac{1}{2} \Delta_t^2 \frac{dg(x)}{dt} \Big|_{t_n} \\
 &= \frac{1}{2} \Delta_t^2 [a^2 x_n - ab] \\
 &= \frac{1}{2} \Delta_t^2 [(-0.5)^2 (-2 + 7e^{-0.5t}) + 0.5] \\
 &< \frac{1}{2} \Delta_t^2 [1.75] \\
 \text{so } d_{n+1} &< 0.875 \Delta_t^2 \quad \text{for our particular problem.}
 \end{aligned}$$

This means halving the step size decreases the local error by a factor of 1/4, however, twice as many steps are required to integrate the system over the original time step and so if we assume errors accumulate in an additive manner over short time intervals, we see *the net effect of halving the step size when using the Euler method is to halve the error*. This is why the Euler method is called a **first** order method, even though it is accurate to order 2. A plot of the true solution to our scalar ODE problem and the predictions of the Euler integrator are shown in Fig. 16.1. We notice that for this particular example, both the local and global error decrease with time – the first is due to the exponential term  $e^{-0.5t_n}$  shrinking with time and the second is due to the Euler approximation and the true system being attracted to the same, steady-state solution.

There are other issues that have not been discussed, but should be mentioned:

- The Euler method works for sets of equations by simply employing vector notation:

$$\mathbf{z}_{n+1} = \mathbf{z}_n + \Delta_t \mathbf{g}(\mathbf{z}_n) \quad \text{subject to } \mathbf{z}_0$$

- The method works with mixed algebraic/ODE systems provided the AEs are solved at each time step for some subset  $\mathbf{y}$  the state variables:

$$\begin{aligned}
 \mathbf{z}_{n+1} &= \mathbf{z}_n + \Delta_t \mathbf{g}(\mathbf{z}_n, \mathbf{y}_n) \\
 \mathbf{f}(\mathbf{z}_n, \mathbf{y}_n) &= 0
 \end{aligned}$$

subject to  $\mathbf{z}_0$ ;

- Stability issues set an upper practical limit on the time step sizes for most numerical integration techniques, because instabilities can cause “runaway” to infinitely large global errors in systems which in reality display bounded, long-time behavior.

### 16.0.1 An isothermal chemical reactor model

Let us consider the simple  $2A \rightarrow B$  reaction taking place in a continuous-flow, stirred-tank reactor.

We make the following two major assumptions:

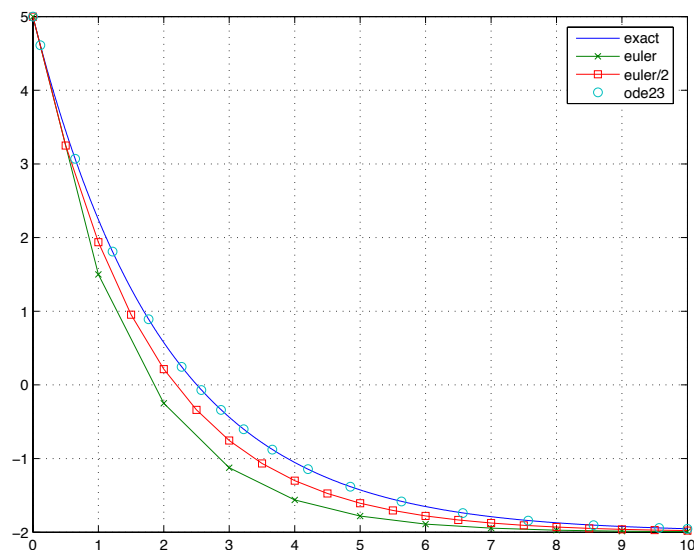


Figure 16.1: Euler method (marked by x's) vs. exact solution (solid curve) for step size  $\Delta_t = 1$ . Also shown are the results from halving the step size (squares) and the 2nd-order accurate Runge-Kutta method (circles), demonstrating its much greater accuracy with fewer time steps.

1. The system is isothermal
2. No volume change results from the chemical reaction

The transient material balance on species A is derived from

$$A_{in} - A_{out} = \text{rate of A accumulation} + \text{rate of A consumption}$$

With  $Q$  as the volumetric flow of reactant feed into the reactor in  $\text{m}^3/\text{s}$ ,  $C$  as the concentration of A in  $\text{mol}/\text{m}^3$ ,  $V$  the reactor volume in  $\text{m}^3$

$$QC_{in} - QC = V \frac{dC}{dt} + 2VkC^2$$

where

$$kC^2 = \text{rate of production of B, mol}/(\text{m}^3\text{s})$$

Defining the dimensionless quantities

$$x = \frac{C}{C_{in}} \quad \tau = \frac{Q}{V}t$$

we find the dimensionless nonlinear ODE

$$1 - x = \frac{dx}{d\tau} + 2D_a x^2$$

subject to initial condition  $x(0) = x_0$  and Damköhler number

$$D_a = \frac{VkC_{in}}{Q}$$

For this problem, we observe that the steady-state, corresponding to  $dx/dt = 0$  is found from the solution of

$$0 = 1 - x_{stst} - 2D_a x_{stst}^2$$

a quadratic equation with two real roots

$$x_{stst} = -\frac{1 \pm \sqrt{1 + 8D_a}}{4D_a}.$$

It is not hard to see that one root always is positive and corresponds to the true solution. The negative solution is physically meaningless; however, it demonstrates saddle-type stability and so initial values less than this root value will diverge to negative infinity.

## 16.0.2 Inert species

Now consider the case where the reactor is initially filled with an inert material and the feed is a combination of A and the inert. Simply extending the modeling approach described above is problematic in that a model where the mole fractions do not sum to unity can result. Therefore, we relax the assumption that no volume change takes place during the reaction and we now assume the species act as an ideal gas. Therefore, molar balances are in order where

$\dot{m}_{in}$  = molar feed flow, mol/s

$\dot{m}_{out}$  = molar product flow, mol/s

$m$  = moles of gas in the reactor

$\phi(t)$  = feed fraction of A (remainder is the inert species)

$x_1, x_2, x_3$  = mole fractions of A, B, and inert, respectively

We note that under the ideal gas assumption and constant reactor volume  $V$ ,  $m$  must then be constant, but  $\dot{m}_{in}$  is likely to be greater than  $\dot{m}_{out}$ .

Using these definitions, we write each of the species material balances as

$$\begin{aligned}\phi \dot{m}_{in} - x_1 \dot{m}_{out} &= m \frac{dx_1}{dt} + 2Vk \left( \frac{x_1 m}{V} \right)^2 \\ 0 - x_2 \dot{m}_{out} &= m \frac{dx_2}{dt} - Vk \left( \frac{x_1 m}{V} \right)^2 \\ (1 - \phi) \dot{m}_{in} - x_3 \dot{m}_{out} &= m \frac{dx_3}{dt}\end{aligned}$$

We can now compute  $\dot{m}_{out}$  by requiring the system to preserve  $x_1 + x_2 + x_3 = 1$  simply by adding the equations up to find

$$\dot{m}_{in} - \dot{m}_{out} = Vk \left( \frac{x_1 m}{V} \right)^2 \quad \text{or} \quad \dot{m}_{out} = \dot{m}_{in} - \frac{km^2}{V} x_1^2$$

Making this substitution and dividing through by  $\dot{m}_{in}$ ,

$$\begin{aligned}\phi - x_1 \left[ 1 - \frac{km^2}{\dot{m}_{in} V} x_1^2 \right] &= \frac{m}{\dot{m}_{in}} \frac{dx_1}{dt} + 2 \frac{km^2}{\dot{m}_{in} V} x_1^2 \\ x_2 \left[ 1 - \frac{km^2}{\dot{m}_{in} V} x_1^2 \right] &= \frac{m}{\dot{m}_{in}} \frac{dx_2}{dt} - \frac{km^2}{\dot{m}_{in} V} x_1^2 \\ (1 - \phi) - x_3 \left[ 1 - \frac{km^2}{\dot{m}_{in} V} x_1^2 \right] &= \frac{m}{\dot{m}_{in}} \frac{dx_3}{dt}\end{aligned}$$

Now after defining our new dimensionless time and Damköhler number as

$$\tau = \frac{\dot{m}_{in}}{m} t \quad D_a = \frac{km^2}{\dot{m}_{in}V}$$

we obtain our set of nonlinear ODEs

$$\begin{aligned} \frac{dx_1}{d\tau} &= \phi - x_1 [1 - D_a x_1^2] - 2D_a x_1^2 \\ \frac{dx_2}{d\tau} &= x_2 [1 - D_a x_1^2] + D_a x_1^2 \\ \frac{dx_3}{d\tau} &= (1 - \phi) - x_3 [1 - D_a x_1^2] \end{aligned}$$

subject to initial condition

$$\mathbf{x}_0 = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{\tau=0} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

### 16.0.3 Time-varying feed composition

An upstream process disturbance affects the concentration of species A in the feed stream. In previous problems,  $C_{in}$  was taken as a constant; we now define

$$\phi(\tau) = \text{mole fraction of A in feed stream} = \frac{1 - \sin 2\pi\tau}{2}$$

where the remainder of the feed stream consists of the inert component. Our reactor modeling equations as well as initial conditions remain the same.

## 16.1 Our Euler integrator

```
function [tout,yout] = euler(fun,del,tfinal,y0,p)
% EULER A forward-Euler integrator, Called as
% [tout,yout] = euler(fun,del,tfinal,y0,p)
% where fun = function handle, del = time step, tfinal = final time,
% y0 is the initial condition, and p contains the parameter(s), if
% any. Note that the function follows the standard MATLAB
% format: dydt = fun(t,y,p)

if nargin < 5, p = []; end % set default as empty
t = 0; y = y0; i = 1; % initial conditions
tout(1) = t; yout(:,1) = y0; % store results to return
```

```

while t <= tfinal % integration loop
    y = y + del*fun(t,y,p);
    t = t + del;
    i = i + 1;
    % check for endpoint
    if t >= tfinal
        slope = (y-yout(:,i-1))/del;
        yout(:,i) = slope*(tfinal-tout(i-1)) + yout(:,i-1);
        tout(i) = tfinal;
        break
    else
        tout(i) = t; yout(:,i) = y;
    end
end
end

```

The function containing that computes the time-derivatives for the ODEs to be solved is in the standard MATLAB format (recall the nonlinear algebraic equations in Chapter 12):

```
dxdt = funname(t,x,p)
```

where  $t$  is the current time (if used explicitly in the differential equations),  $x$  is a column vector of the current state variable values, and  $p$  is a vector (or other data structure) of parameter values. The Euler integrator typically would be used in the following manner:

```

>> f = @funname;
>> [t,y] = euler(f,del,tfinal,y0,p);
>> plot(t,y)
>> % compute steady state using the final point
>> % in time as initial Newton guess
>> [ystst,conflag] = nr(y(:,end),p,f);
>> if conflag, disp('Converged solution xstst = '), end
>> disp(xstst)

```

## 16.2 Nonlinear pendulum example

Consider the pendulum equation

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \sin \theta \quad (16.1)$$

which for small amplitude oscillations can be linearized ( $\sin \theta \approx \theta$ ) to obtain

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \theta$$

to find the classic (i.e., Galileo's) result that the frequency of oscillation is

$$\omega = \sqrt{g/L}$$

which is independent of the amplitude of the oscillations. We wish to consider the fully nonlinear system (16.1) using our Euler integrator, introducing two new concepts:

1. How one can convert (16.1) into initial-value-problem form;
2. How one can examine the dynamics of a system in the *phase plane*.

Before proceeding, we introduce the dimensionless time  $\tau = t\sqrt{g/L}$  to find the dimensionless nonlinear pendulum equation:

$$\frac{d^2\theta}{d\tau^2} = -\sin \theta$$

Defining the *angular velocity* as

$$\Omega = \frac{d\theta}{d\tau}$$

allows us to split (16.1) into two equations

$$\begin{aligned}\frac{d\theta}{d\tau} &= \Omega \\ \frac{d\Omega}{d\tau} &= -\sin \theta\end{aligned}$$

now subject to the initial conditions

$$\mathbf{x}_0 = \begin{bmatrix} \theta(0) \\ \Omega(0) \end{bmatrix}$$

We understand that this problem much conserve energy; therefore, we defined the *Hamiltonian*  $H$  as

$$H = \frac{1}{2}\Omega^2 - \cos \theta$$

which consist of the sum of kinetic and potential energy for the dimensionless system. It is simple to prove  $H$  is a content by differentiating with repeat to dimensionless time:

$$\begin{aligned}\frac{dH}{d\tau} &= \Omega \frac{d\Omega}{d\tau} + \sin \theta \frac{d\theta}{d\tau} \\ &= \Omega(-\sin \theta) + (\sin \theta)\Omega \\ &= 0\end{aligned}$$

Examining the pendulum equations in the phase plane reveals the following dynamical behaviors corresponding to different values of  $H \geq -1$ :

$H = -1$	vertically downward, motionless
$H = 0$	results from initial condition $\Omega_0 = 0, \theta_0 = \pi/2$
$H = 1$	defines the separatrix
$H > 1$	continuous revolutions about the pivot

## 16.3 Computing time step sizes – the Runge-Kutta methods

The motivation for developing higher-order integration schemes can be justified by the following argument: if we compare a third-order integrator to the first-order Euler, we will see that the 3rd-order method requires two additional evaluations of the right-hand-side function  $g$  in order to cover a single time-step

$\Delta_t$ . This means we should allow our Euler to take step sizes which are one third this value, and so the accumulated discretization error of the Euler scheme  $d_E$  might be

$$d_E = 3c_E \left( \frac{1}{3} \Delta_t \right)^2 = \frac{1}{3} c_E \Delta_t^2.$$

The 3rd-order method, however, generates the following error

$$d_T = c_T \Delta_t^4$$

and so given that the two constants are bounded, we should be able to find a suitably small time step  $\Delta_t$  such that the third-order method is more accurate than first, with both methods requiring the same amount of computational work.

Let us consider deriving a second-order accurate integrator, a numerical technique which generates errors of order  $\Delta_t^3$  (written as  $\mathcal{O}(\Delta_t^3)$ ) during each step

$$\begin{aligned} x(t_{n+1}) &= x(t_n) + \Delta_t \frac{dx}{dt} \Big|_{t_n} + \frac{1}{2} \Delta_t^2 \frac{d^2x}{dt^2} \Big|_{t_n} + \frac{1}{6} \Delta_t^3 \frac{d^3x}{dt^3} \Big|_{t_n} + \mathcal{O}(\Delta_t^3) \\ &= x(t_n) + \Delta_t g(x_n, t_n) + \frac{1}{2} \Delta_t^2 \frac{d}{dt} g(x, t) \Big|_{t_n} + \frac{1}{6} \Delta_t^3 \frac{d^2}{dt^2} g(x, t) \Big|_{t_n} + \mathcal{O}(\Delta_t^3) \\ &= x(t_n) + \Delta_t g(x_n, t_n) + \frac{1}{2} \Delta_t^2 \left[ \frac{\partial g}{\partial x} \frac{dx}{dt} + \frac{\partial g}{\partial t} \right]_{t_n} + \mathcal{O}(\Delta_t^3) \\ &= x(t_n) + \Delta_t g(x_n, t_n) + \frac{1}{2} \Delta_t^2 [g_x g + g_t]_{t_n} + \mathcal{O}(\Delta_t^3) \end{aligned}$$

If we rewrite the Euler method in “update” notation:

$$x_{n+1} = x_n + u \quad \text{with} \quad u = \Delta_t g(x_n, t_n),$$

and let  $u_1$  be our first estimate of the true update

$$u_1 = \Delta_t g(x_n, t_n)$$

and use this to compute the update at some fraction of the total time step

$$\begin{aligned} u_2 &= \Delta_t g(x_n + \beta u_1, t_n + \alpha \Delta_t) \\ &\approx \Delta_t g + \Delta_t [g_x u_1 (\beta - 0) + g_t \Delta_t (\alpha - 0)]_{t_n} + \mathcal{O}(\Delta_t^3) \\ &= \Delta_t g + \Delta_t^2 [g_x g \beta + g_t \alpha]_{t_n} + \mathcal{O}(\Delta_t^3) \quad \text{near } (x_n, t_n) \quad (\text{small } \alpha, \beta) \end{aligned}$$

We use the subscript notation  $g_x$  and  $g_t$  to denote partial derivatives with respect to  $x$  and  $t$ , and choose  $0 < \alpha < 1$  and  $0 < \beta < 1$ . We can now use some linear combination of the two updates to compute the state of the system at the end of the time step  $\Delta_t$ :

$$x_{n+1} = x_n + \gamma_1 u_1 + \gamma_2 u_2. \tag{16.2}$$

The coefficients  $\gamma_1$  and  $\gamma_2$  are determined by substituting in the expressions for  $u_1$  and the Taylor's series of  $u_2$  at the current time step  $(x_n, t_n)$  to (16.2)

$$\begin{aligned} x_{n+1} &= x_n + \gamma_1 \Delta_t g + \gamma_2 [\Delta_t g + \Delta_t^2 (g_x g \beta + g_t \alpha)] + \mathcal{O}(\Delta_t^3) \\ &= x_n + \Delta_t [\gamma_1 g + \gamma_2 g] + \Delta_t^2 [\gamma_2 g_x g \beta + \gamma_2 g_t \alpha] + \mathcal{O}(\Delta_t^3) \end{aligned}$$



and so comparing the expression above term-by-term with the expansion of the true solution gives

$$\begin{aligned}\gamma_1 g + \gamma_2 g &= g \\ \gamma_2 g_x g \beta &= \frac{1}{2} g_x g \\ \gamma_2 g_t \alpha &= \frac{1}{2} g_t\end{aligned}$$

so

$$\gamma_1 + \gamma_2 = 1 \quad \gamma_2 \beta = \frac{1}{2} \quad \gamma_2 \alpha = \frac{1}{2}.$$

If we choose  $\gamma_2 = 0$  we recover our single step Euler integrator. A typical choice for  $\alpha$  (truly optimal values require more extensive calculations) to give us a unique solution to the set of three equations is  $\alpha = 1/2$ . This means we can write our integration procedure as follows

1. With an initial guess or the most recently used value of  $\Delta_t$ , compute the value of the state at the next time step, accurate to  $\mathcal{O}(\Delta_t^3)$  (resulting in a 2nd-order accurate integrator) using  $\alpha = \beta = 1/2$ ,  $\gamma_2 = 1$ ,  $\gamma_1 = 0$ :

$$\begin{aligned}u_1 &= \Delta_t g(x_n, t_n) \\ u_2 &= \Delta_t g(x_n + u_1/2, t_n + \Delta_t/2) \\ x_{n+1} &= x_n + u_2.\end{aligned}$$

2. Compute the difference between the 1st-order and 2nd-order accurate solutions

$$\begin{aligned}d_{n+1} &= (x_n + u_2) - (x_n + u_1) \\ &= u_2 - u_1 \\ &= C \Delta_t^2\end{aligned}$$

3. Compute the time step size which guarantees a specified error tolerance  $d_{max}$  of a 1st-order accurate solution by first evaluating the constant above and then computing

$$\Delta_t = \sqrt{d_{max}/C}$$

The solution accurate to  $\mathcal{O}(\Delta_t^3)$  is then recomputed; we also do not allow the step size to grow beyond a certain practical constraint.

A comparison of the 1st-order accurate Euler integrator and the 2nd-order accurate Runge-Kutta integrator (with  $d_{max} = 0.02$ ) are shown in Fig. 16.1.

## 16.4 Numerical integration of ordinary differential equations

Let us first test some computational methods for solving ODEs on a scalar system that has an exact solution.

```

a = -0.5; b = 1;
x0 = 5;
tfinal = 10;
texact = [0:0.01:tfinal];
xexact = (a*x0-b)*exp(a*texact)/a + b/a;

delt = 1; % Euler integrator for the fixed stepsize
t = 0;
x = x0;
i = 1;

while t <= tfinal
    teuler(i) = t;
    xeuler(i) = x;
    dxdt = odetrivial(t,x);
    x = x + delt*dxdt;
    i = i + 1;
    t = t + delt;
end

```

Now, halve the step size

```

delt = delt/2;
t = 0;
x = x0;
i = 1;

while t <= tfinal
    teuler_half(i) = t;
    xeuler_half(i) = x;
    dxdt = odetrivial(t,x);
    x = x + delt*dxdt;
    i = i + 1;
    t = t + delt;
end

```

Now use a MATLAB Runge-Kutta numerical integration routine (ode23)

```

[tode23,xode23] = ode23('odetrivial',[0, tfinal],x0);

plot( texact,xexact,teuler,xeuler,'x-',teuler_half, ...
      xeuler_half,'s-',tode23,xode23,'o')
grid on
legend('exact','euler','euler/2','ode23')

```

and the function odefun.m called by MATLAB's ode23 method:

```

function dxdt = odetrivial(t,x)
a = -0.5; b = 1;
dxdt = a*x - b;

```

### 16.4.1 A nonlinear problem

Create a file called `lorfun.m` and include the following function:

```
function f = lorfun(t,s)
% Lorenz attractor
sigma = 10; rho = 28; beta = 8/3;
x = s(1,1);
y = s(2,1);
z = s(3,1);
f(1,1) = -beta*x + y*z;
f(2,1) = sigma*(z-y);
f(3,1) = -x*y + rho*y - z;
```

and then try

```
[T,Y] = ode45(@lorfun,[0 40],[5; 20; -10]);
figure(1)
subplot(2,1,1)
plot(T,Y)
xlabel('time')
legend('x','y','z')
grid on
axis tight

figure(2)
subplot(2,2,1)
plot3(Y(:,1),Y(:,2),Y(:,3))
grid on
xlabel('x'), ylabel('y'), zlabel('z')
axis tight
```

The dynamic solutions are plotted in Fig. 16.2.

## 16.5 Review problems

1. The concentration of a chemical species in a tubular reactor is denoted  $x$  and it is a function of position within the reactor  $x(z)$ . Given the inlet concentration  $x(0) = 1$ , find the length of reactor  $z = L$  required to achieve 99% conversion of the reactant if the reactor can be modeled by the differential equation:

$$\frac{dx}{dz} = -0.1x^2$$

Use your own forward Euler method.

2. Diffusion and reaction in a catalyst pellet can be described by

$$0 = 0.5 \frac{d^2x}{dz^2} - 2x^2$$

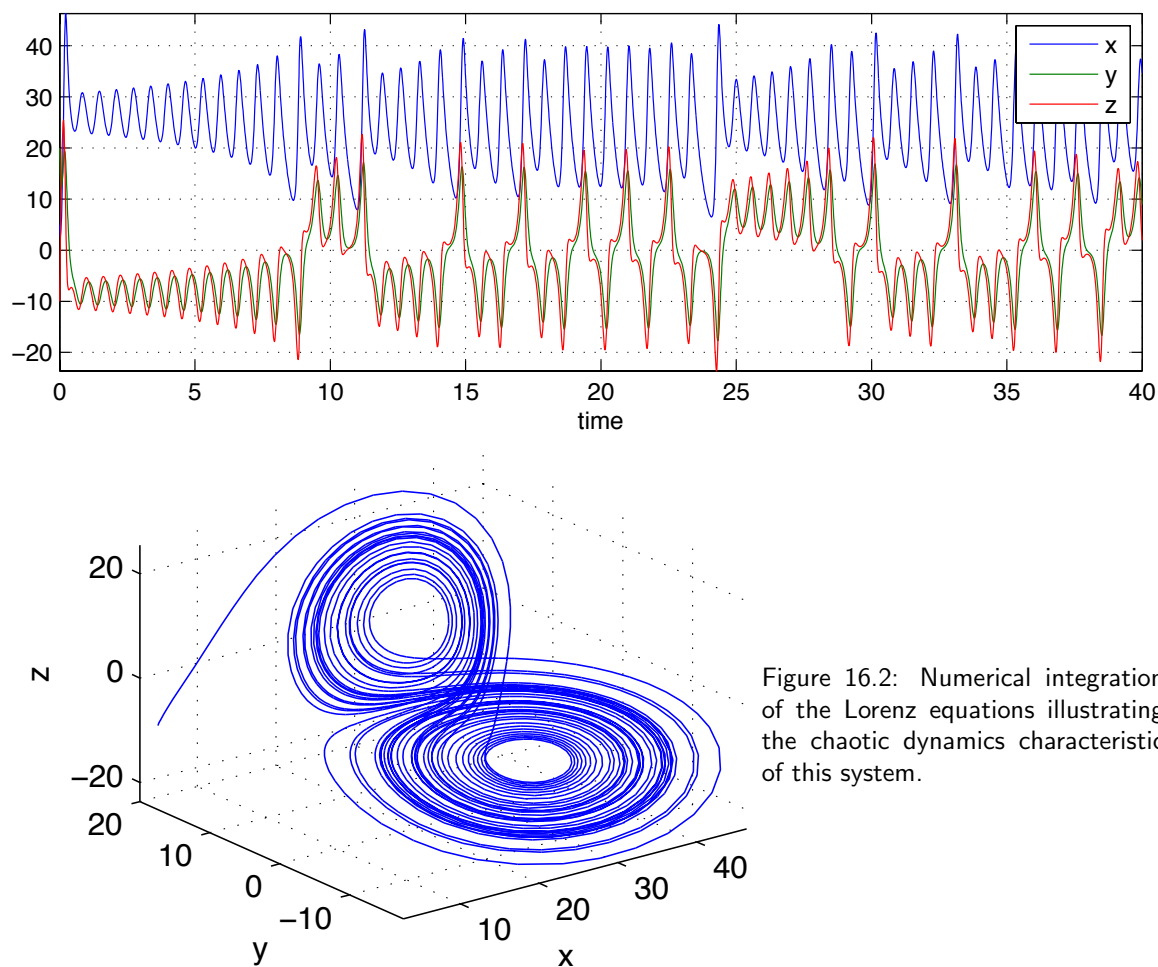


Figure 16.2: Numerical integration of the Lorenz equations illustrating the chaotic dynamics characteristic of this system.

subject to boundary conditions

$$\frac{dx}{dz} = 0 \text{ at } z = 0 \text{ and } x = 1 \text{ at } z = 1.$$

Transform this into a system of 2 equations by defining  $y = dx/dz$ , and develop a Newton procedure to compute the reactant profile  $x(z)$  solution by guessing a value of  $x(0)$  and integrating the system of equations to  $z = 1$  using MATLAB's `ode45.m`

3. Consider the system of equations:

$$\begin{aligned} dx/dt &= v \\ dy/dt &= -2(x^2 - 1)y - x \end{aligned}$$

By hand, determine the steady state(s) for this system; at this steady state, determine the value of the elements in the Jacobian array and using MATLAB's `eig.m` method, determine if the system is stable at this point. Then, using MATLAB's `ode45.m`, compute solutions in time and plot them to assess the true long-term behavior of this system.

## Chapter 17

# Orthogonal function sequences

### 17.1 Norms, function spaces, and orthogonal functions

In a lumped-parameter system, we have a clearly defined notion of distance as a difference between two values of a scalar function, e.g., the temperature difference between two objects of uniform temperature. In a distributed parameter system, this idea is not as straightforward. For example, in Fig. 17.1, how do we measure the difference between curves  $f(x)$  and  $T(x)$  in such a way that the computation produces a *scalar* value? How do we then use this to fit our general solution to a particular solution determined by an initial condition?

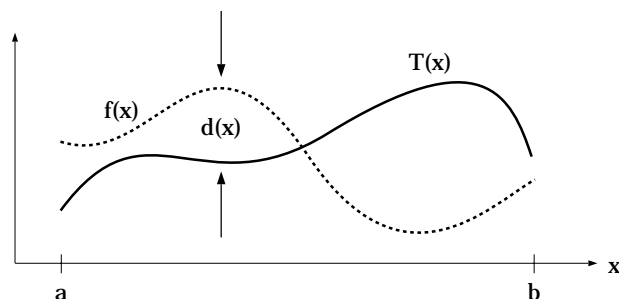


Figure 17.1: *The difference between two functions of a single variable  $x$ .*

The difference between two real-valued scalar functions,  $f$  and  $T$ , over the range  $b \geq x \geq a$  can be measured in a number of ways:

$$\begin{aligned}\int_a^b |f(x) - T(x)| dx &= \|f(x) - T(x)\|_1 \\ \left( \int_a^b [f(x) - T(x)]^2 dx \right)^{1/2} &= \|f(x) - T(x)\|_2 \\ \sup_{b \geq x \geq a} |f(x) - T(x)| &= \|f(x) - T(x)\|_\infty\end{aligned}$$

Functions defined in the domain  $\Omega$  whose norms are bounded are defined in the function spaces

$$L^1(\Omega) \quad L^2(\Omega) \quad L^\infty(\Omega)$$

These function spaces are vector spaces; we will be primarily interested in real-valued functions in Hilbert spaces  $X = L^2(\Omega)$ , and so, in general, we will use the norm  $\|\cdot\|$  definition

$$\|f\|_2 = \left( \int_{\Omega} f^2 d\omega \right)^{1/2}$$

where  $\Omega$  is the (spatial) domain of function  $f$  and  $d\omega$  is a differential element in that domain.

### 17.1.1 The optimal projection

In later chapters, we frequently will encounter the problem of finding the best approximation  $T(x)$  of a known function  $f(x)$  where  $T(x)$  is defined in terms of the *trial function expansion*

$$T(x) = \sum_{j=0}^N a_j \psi_j(x)$$

in domain  $\Omega$ . We can give some meaning to the equivalence of  $T$  and  $f$  in the 2-norm sense by defining the distance function  $d$  as

$$d(x) = f(x) - T(x) = 0 \quad \text{in } \Omega$$

and so the two curves are considered equal<sup>1</sup> when  $\|d\|_2^2 = 0$ . Thus, dropping the 2 subscript,

$$\begin{aligned} \|f(x) - T(x, 0)\|^2 &= \|d\|^2 \\ \text{so } \int_{\Omega} (f^2 - 2fT_0 + T_0^2) d\omega &= R \end{aligned}$$

Our goal is to find a sequence  $a_j$  such that the constant  $R$  (the residual) is minimized. This condition becomes

$$\begin{aligned} \frac{\partial R}{\partial a_j} &= \frac{\partial}{\partial a_j} \int_{\Omega} \left[ f^2 - 2f \sum_{m=0}^{\infty} a_m \psi_m + \left( \sum_{m=0}^{\infty} a_m \psi_m \right)^2 \right] d\omega \\ &= - \int_{\Omega} 2f \psi_j d\omega + \int_{\Omega} 2 \left( \sum_{m=0}^{\infty} a_m \psi_m \right) \psi_j d\omega \\ &= -2 \int_{\Omega} f \psi_j d\omega + 2a_j \int_{\Omega} \psi_j \psi_j d\omega \quad \text{when} \quad \int_{\Omega} \psi_j \psi_k d\omega = 0 \text{ for } j \neq k. \end{aligned}$$

Notice how the integrals are simplified when the  $\psi_m$  belong to an *orthogonal* sequence of functions – we will return to this point in the following section. Thus,

$$\frac{\partial R}{\partial a_j} = 0 \quad \text{when} \quad a_j = \frac{\langle f, \psi_j \rangle}{\langle \psi_j, \psi_j \rangle} \quad (17.1)$$

<sup>1</sup>It is important to stress that this measurement of the equivalence of two curves is restricted to the particular norm used – it is possible to find cases where curves deemed equal in one norm are found to be different under a different norm.

where the *inner product* of two functions  $\langle f, g \rangle$  is defined as

$$\langle f, g \rangle = \int_{\Omega} fg \, d\omega.$$

Because  $\partial^2 R / \partial a_j^2 = 2\langle \psi_j, \psi_j \rangle > 0$ , this does indeed correspond to a minimum. This is the classic Fourier coefficient calculation. Notice, also, the intimate ties of this method for computing the “best” initial mode amplitude coefficients with the least-squares solution procedure.

### 17.1.2 Orthogonal function sequences

Two functions  $\psi_i(x)$  and  $\psi_j(x)$  are defined as orthogonal over a domain  $\Omega$  if they satisfy

$$\langle \psi_j, \psi_k \rangle = \int_{\Omega} \psi_j \psi_k \, d\omega = \begin{cases} c_j & \text{if } j = k \\ 0 & \text{otherwise.} \end{cases}$$

This property is guaranteed by the Sturm-Liouville theorem, which will be discussed in more detail in section 17.3 which is dedicated to these problems. Furthermore, if the constants  $c_j = \langle \psi_j, \psi_j \rangle = 1$ , the sequence of functions  $\psi_j$  is called an *orthonormal* sequence. Orthonormal sequences then simplify computations of the coefficients in determining a particular solution from a general solution, i.e.,

$$a_j = \langle f, \psi_j \rangle.$$

As an example of an orthogonal function sequence, consider

$$\psi_j = \cos \pi j x \quad \text{in } 0 \leq x \leq 1.$$

Computing the inner product over the domain gives

$$\begin{aligned} \langle \cos \pi j x, \cos \pi k x \rangle &= \int_0^1 \cos \pi j x \cos \pi k x \, dx \\ &= \int_0^1 \frac{1}{2} [\cos \pi(j-k)x + \cos \pi(j+k)x] \, dx \\ &= \int_0^1 \frac{1}{2} [1 + \cos 2\pi j x] \, dx \quad \text{for } j = k \\ &= \frac{1}{2} \quad \text{for } j = k \\ &= 0 \quad \text{for } j \neq k. \end{aligned}$$

It is important to point out that orthogonality is defined over a specified physical domain. Consider, for example the terms of the Fourier series

$$1 \quad \cos x \quad \sin x \quad \cos 2x \quad \sin 2x \quad \dots$$

These functions are orthogonal over the interval  $0 \leq x \leq 2\pi$ , but are not all orthogonal over  $0 \leq x \leq \pi$ .

### 17.1.3 The projection operator

We have now established that it is possible to represent a time and spatially varying state in terms of a time-dependent linear combination of spatially-varying, but time-independent trial functions  $\psi_j(x)$ :

$$T(x, t) = a_0(t)\psi_0(x) + a_1(t)\psi_1(x) + \dots, \quad (17.2)$$

If this series has a finite number of terms, can define the *projection operator*  $\mathcal{P}^N$  as

$$\mathcal{P}^N T(x, t) = \langle T, \psi_0 \rangle \psi_0 + \langle T, \psi_1 \rangle \psi_1 + \dots + \langle T, \psi_N \rangle \psi_N.$$

It is interesting to relate this projection of some function  $T$  onto the trial functions  $\psi_i$  to computing the power spectrum of a signal, or quantifying the contribution (e.g., kinetic energy) of each mode to the function  $T$  by its *mode amplitude coefficient*  $a_i = \langle T, \psi_i \rangle$ .

We will sometimes find it convenient to write the trial function expansions in vector notation. Defining the vector of trial functions as

$$\boldsymbol{\psi} = [\psi_0(x) \ \psi_1(x) \ \dots \ \psi_N(x)]$$

and the mode amplitude coefficients as

$$\mathbf{a} = [a_0(t) \ a_1(t) \ \dots \ a_N(t)]^T,$$

the truncated  $N$ -term trial function expansion (17.2) can be written as

$$T^N(x, t) = \boldsymbol{\psi} \mathbf{a}.$$

### 17.1.4 Gram-Schmidt orthogonalization

Let us consider a sequence of *trial* functions

$$\phi_0, \phi_1, \phi_2, \dots \quad \text{and norm} \quad \|\phi_i\| = \sqrt{\langle \phi_i, \phi_i \rangle}.$$

We can create an orthonormal basis  $\psi_i$  which *spans* the subspace of  $X$  spanned by  $\phi_i$  with the Gram-Schmidt procedure. This technique is useful for eliminating redundant trial functions and simplifies the process of projecting arbitrary functions onto these trial functions.

The arrangement of the initial sequence  $\phi_i$  does not matter, and so we choose the first function and normalize it over its domain

$$\psi_0 = \frac{\phi_0}{\|\phi_0\|} \quad \text{so} \quad \langle \psi_0, \psi_0 \rangle = 1$$

Now compute the “portion” of the next function  $\phi_1$  which is orthogonal to  $\psi_0$ :

$$\mathbf{v}_1 = \phi_1 - \langle \psi_0, \phi_1 \rangle \psi_0$$

so the inner product can be thought of as the mode amplitude or contribution of  $\psi_0$  to  $\phi_1$ . We normalize the remainder to obtain the next function in our orthonormal sequence

$$\psi_1 = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}.$$

We can continue this process

$$\mathbf{v}_2 = \phi_2 - \langle \psi_1, \phi_2 \rangle \psi_1 - \langle \psi_0, \phi_2 \rangle \psi_0$$

and so obtain the Gram-Schmidt orthogonalization procedure

$$\psi_n = \frac{\mathbf{v}_n}{\|\mathbf{v}_n\|} \quad \text{with} \quad \mathbf{v}_n = \phi_n - \sum_{k=0}^{n-1} \langle \psi_k, \phi_n \rangle \psi_k = \phi_n - \mathcal{P}^{n-1} \phi_n.$$



### 17.1.5 SVD based orthogonalization

Consider the set of functions  $\{\phi_i(x)\}_{i=1}^I$ . We would like to find a new set of functions  $\{\psi_j(x)\}_{j=1}^J$  with  $J \leq I$  that form a basis for the space spanned by the  $\phi_i$ , where the  $\psi_j$  are orthogonal under inner product

$$\langle \psi_m, \psi_n \rangle = \int_{\Omega} \psi_m(x) \psi_n(x) w(x) dx.$$

To being, we recall the singular value decomposition of matrix  $\mathbf{A}$ :

$$\mathbf{U} \mathbf{\Sigma} \mathbf{V}^T = \mathbf{A}^{m \times n}$$

with  $\mathbf{U}^{m \times m}$ ,  $\mathbf{\Sigma}^{m \times n}$ , and  $\mathbf{V}^{n \times n}$ . Writing out the arrays explicitly for the decomposition of a representative  $3 \times 3$  case

$$\begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ U_{2,1} & U_{2,2} & U_{2,3} \\ U_{3,1} & U_{3,2} & U_{3,3} \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{1,2} & V_{1,3} \\ V_{2,1} & V_{2,2} & V_{2,3} \\ V_{3,1} & V_{3,2} & V_{3,3} \end{bmatrix}^T = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

and so multiplying through by the singular values  $\sigma_i$

$$\begin{bmatrix} \sigma_1 \begin{bmatrix} U_{1,1} \\ U_{2,1} \\ U_{3,1} \end{bmatrix} & \sigma_2 \begin{bmatrix} U_{1,2} \\ U_{2,2} \\ U_{3,2} \end{bmatrix} & \sigma_3 \begin{bmatrix} U_{1,3} \\ U_{2,3} \\ U_{3,3} \end{bmatrix} \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{2,1} & V_{3,1} \\ V_{1,2} & V_{2,2} & V_{3,2} \\ V_{1,3} & V_{2,3} & V_{3,3} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

Denoting

$$[\mathbf{v}^T]_i = \begin{bmatrix} V_{i,1} \\ V_{i,2} \\ V_{i,3} \end{bmatrix}$$

we see that  $[\mathbf{v}^T]_i$  are the linear combinations of the basis vectors  $\mathbf{u}_i$ , weighted by their corresponding singular values  $\sigma_i$ , that produce the original vectors  $\mathbf{a}_i$ . Likewise, because of the orthogonality of the  $\mathbf{v}_i$  (i.e.,  $\mathbf{V}^T \mathbf{V} = \mathbf{I}$ )

$$\begin{bmatrix} \sigma_1 \begin{bmatrix} U_{1,1} \\ U_{2,1} \\ U_{3,1} \end{bmatrix} & \sigma_2 \begin{bmatrix} U_{1,2} \\ U_{2,2} \\ U_{3,2} \end{bmatrix} & \sigma_3 \begin{bmatrix} U_{1,3} \\ U_{2,3} \\ U_{3,3} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{1,2} & V_{1,3} \\ V_{2,1} & V_{2,2} & V_{2,3} \\ V_{3,1} & V_{3,2} & V_{3,3} \end{bmatrix}$$

we observe that the  $\mathbf{v}_i$  are the linear combinations of the original column vectors  $\mathbf{a}_i$  that give the  $i$ th basis  $\mathbf{u}_i$  vector multiplied by its corresponding singular value  $\sigma_i$

We reiterate important observation is that the eigenvectors  $\mathbf{u}_i$  corresponding to small-magnitude singular values  $\sigma_i$  contribute little to the original vectors  $\mathbf{a}_j$ ; we note that because  $\langle \mathbf{u}_i, \mathbf{u}_i \rangle = 1$ , the magnitude of elements must remain on the order of 1.

Now consider representing one of the new functions we see  $\psi_j(x)$  in terms of the functions we are given  $\phi_j(x)$

$$\psi_j(x) = a_{j,1}\phi_1(x) + a_{j,2}\phi_2(x) + a_{j,3}\phi_3(x).$$

Taking the inner product with  $\phi_1$  gives

$$\langle \psi_j, \phi_1 \rangle = a_{j,1} \langle \phi_1, \phi_1 \rangle + a_{j,2} \langle \phi_2, \phi_1 \rangle + a_{j,3} \langle \phi_3, \phi_1 \rangle$$

so if

$$\mathbf{A} = \begin{bmatrix} \langle \phi_1, \phi_1 \rangle & \langle \phi_1, \phi_2 \rangle & \langle \phi_1, \phi_3 \rangle \\ \langle \phi_2, \phi_1 \rangle & \langle \phi_2, \phi_2 \rangle & \langle \phi_2, \phi_3 \rangle \\ \langle \phi_3, \phi_1 \rangle & \langle \phi_3, \phi_2 \rangle & \langle \phi_3, \phi_3 \rangle \end{bmatrix}$$

we can write

$$\begin{bmatrix} \sigma_1 \begin{bmatrix} \langle \psi_1, \phi_1 \rangle \\ \langle \psi_1, \phi_2 \rangle \\ \langle \psi_1, \phi_3 \rangle \end{bmatrix} & \sigma_2 \begin{bmatrix} \langle \psi_2, \phi_1 \rangle \\ \langle \psi_2, \phi_2 \rangle \\ \langle \psi_2, \phi_3 \rangle \end{bmatrix} & \sigma_3 \begin{bmatrix} \langle \psi_3, \phi_1 \rangle \\ \langle \psi_3, \phi_2 \rangle \\ \langle \psi_3, \phi_3 \rangle \end{bmatrix} \end{bmatrix} =$$

$$\begin{bmatrix} A_{1,1} V_{1,1} + A_{1,2} V_{2,1} + A_{1,3} V_{3,1} & A_{1,1} V_{1,2} + A_{1,2} V_{2,2} + A_{1,3} V_{3,2} & A_{1,1} V_{1,3} + A_{1,2} V_{2,3} + A_{1,3} V_{3,3} \\ A_{2,1} V_{1,1} + A_{2,2} V_{2,1} + A_{2,3} V_{3,1} & A_{2,1} V_{1,2} + A_{2,2} V_{2,2} + A_{2,3} V_{3,2} & A_{2,1} V_{1,3} + A_{2,2} V_{2,3} + A_{2,3} V_{3,3} \\ A_{3,1} V_{1,1} + A_{3,2} V_{2,1} + A_{3,3} V_{3,1} & A_{3,1} V_{1,2} + A_{3,2} V_{2,2} + A_{3,3} V_{3,2} & A_{3,1} V_{1,3} + A_{3,2} V_{2,3} + A_{3,3} V_{3,3} \end{bmatrix}$$

which, by comparing each array element-by-element finally implies

$$\begin{aligned} \sigma_j \langle \psi_j, \phi_i \rangle &= A_{i,1} V_{1,j} + A_{i,2} V_{2,j} + A_{i,3} V_{3,j} \\ &= \langle \phi_i, \phi_1 \rangle V_{1,j} + \langle \phi_i, \phi_2 \rangle V_{2,j} + \langle \phi_i, \phi_3 \rangle V_{3,j} \\ \text{so } \sigma_j \psi_j &= \phi_1 V_{1,j} + \phi_2 V_{2,j} + \phi_3 V_{3,j} \end{aligned}$$

Likewise

$$\begin{bmatrix} \sigma_1 \begin{bmatrix} \langle \psi_1, \phi_1 \rangle \\ \langle \psi_1, \phi_2 \rangle \\ \langle \psi_1, \phi_3 \rangle \end{bmatrix} & \sigma_2 \begin{bmatrix} \langle \psi_2, \phi_1 \rangle \\ \langle \psi_2, \phi_2 \rangle \\ \langle \psi_2, \phi_3 \rangle \end{bmatrix} & \sigma_3 \begin{bmatrix} \langle \psi_3, \phi_1 \rangle \\ \langle \psi_3, \phi_2 \rangle \\ \langle \psi_3, \phi_3 \rangle \end{bmatrix} \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{2,1} & V_{3,1} \\ V_{1,2} & V_{2,2} & V_{3,2} \\ V_{1,3} & V_{2,3} & V_{3,3} \end{bmatrix} =$$

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

so again, comparing each side element-by-element,

$$\begin{aligned} \sigma_1 \langle \psi_1, \phi_i \rangle V_{j,1} + \sigma_2 \langle \psi_2, \phi_i \rangle V_{j,2} + \sigma_3 \langle \psi_3, \phi_i \rangle V_{j,3} &= A_{i,j} \\ &= \langle \phi_i, \phi_j \rangle \\ \text{so } \sigma_1 \psi_1 V_{j,1} + \sigma_2 \psi_2 V_{j,2} + \sigma_3 \psi_3 V_{j,3} &= \phi_j. \end{aligned}$$

### The SVD method for class scalarfield

The MATLAB command

$$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A}) \quad \parallel$$

returns a vector of scalarfield objects  $\mathbf{U}$  representing the orthogonal basis for the vector of scalarfield objects  $\mathbf{A}$  of length  $m$ ;  $\mathbf{S}$  is a vector of singular values with the same dimensions as vector  $\mathbf{A}$  and  $\mathbf{V}$  is an  $m \times m$  array such that if  $\mathbf{A}$  is a row vector of scalarfield objects,

$$\mathbf{U} = \mathbf{A} * \mathbf{V} \quad \parallel$$

Note that the orthogonal basis functions are normalized using the scalarfield method

$$\mathbf{U} = \text{normalize}(\mathbf{U}) \quad \parallel$$

## 17.2 Numerical representations of basis function sequences

In our computational methods, the one-dimensional basis function components  $\phi_i(x)$ ,  $\psi_j(y)$ , are represented as vectors of the function values at a set of quadrature points  $\hat{\mathbf{x}}$ ,  $\hat{\mathbf{y}}$ , respectively. For example, in the case of the function sequence  $\{\phi_i(x)\}_{i=1,\dots,I}$ , the  $n$  quadrature points are defined as the combination of the unit interval endpoints and the  $n-2$  roots of a shifted orthogonal Jacobi polynomial  $J_{n-2}^{\alpha+1,\beta+1}(x)$ , a polynomial sequence orthogonal with respect to inner product weight  $x^\alpha(1-x)^\beta$  where  $J_0^{\alpha+1,\beta+1} = 1$ ;  $\alpha = 0, 1$ , or  $2$ , and corresponds to the slab, cylindrical, or spherical geometries, respectively ( $\beta = 0$  in this work). Equidistant points  $\hat{x}_k = k/n$  are used for periodic physical domains. In all cases, it is required that  $n \geq I + 2$ .

Fundamental to spectral projection methods is the definition of the basis functions. Orthogonal polynomial sequences can be readily generated on the quadrature grid using recurrence formulas, and subsequently can be normalized numerically by quadrature. Likewise, we find it convenient to define a basis function sequence by the solutions to These functions are orthogonal with respect to the quadrature-based inner product operations and are typically normalized prior to use. Eigenfunctions of this form are automatically generated in the BasisFun object constructor.

### 17.2.1 basisfun: basis function class

One of the key elements of implementing a global spectral projection discretization method is defining the basis functions. The basisfun object class was created to store basis function sequences, eigenvalue arrays (if the basis functions are eigenfunctions), and the physical domain over which the basis functions are defined (a quadgrid object). Each basis function in each sequence is discretized at the quadrature points of the quadgrid object; the basis functions themselves can be approximate solutions to a Sturm-Liouville problem or can be generated from a recurrence relation or other means.

In multidimensional applications, a single basisfun object is used to store all basis function sequence components, and this basisfun object is created using the overloaded  $*$  operator. For example, if

$$\begin{aligned} \text{object Phi} &= \{\phi_i(x)\}_{i=1,\dots,I} \\ \text{object Psi} &= \{\psi_j(y)\}_{j=1,\dots,J} \\ \text{object P} &= \text{objects Phi} * \text{Psi} = \{\phi_i(x)\psi_j(y)\}_{i=1,\dots,I, j=1,\dots,J} \end{aligned}$$

Having described methods for creating basisfun objects, the most important method of the class is the weighted inner product method wip.m. This method is used to project scalar field (scalarfield) objects onto basis function sequences, and to compute inner products of basis function sequences with other sequences; further information can be found in the table below and in the representative applications found in the remainder of this chapter.

<b>basisfun</b>
quadgrid : quadgrid or relquad fun : double eigv : double name : char cell array
basisfun(X,fun,eigv,name) bs2sf(A,indices) display(A) eig(A) even(A) get(A,field) modes(A,modeno) mtimes(A,B) odd(A) truncate(A,N) truncno(A) wip(A,B)

<b>basisfunlegendre</b>
basisfun : basisfun option : char N : double
basisfunlegendre(X,coname,option,N) display(A)

<b>basisfunperiodic</b>
basisfun : basisfun
basisfunperiodic(X,coname) display(A)

## 17.3 Sturm-Liouville problems

We encounter the eigenvalue problem

$$\mathbf{A}\mathbf{u} = \lambda\mathbf{u} \quad (17.3)$$

as part of the solution procedure for time-invariant linear sets of ODEs. Recall that the eigenvectors  $\mathbf{u}$  are found as nontrivial solutions to (17.3) and that the  $\mathbf{u}$  define an orthogonal basis for the vector space where snapshots of solutions to the ODE system exist. In this chapter we consider the analogous eigenvalue problem

$$Av(x) + \lambda v(x) = 0 \quad (17.4)$$

which defines the eigenfunctions  $v$  for specific forms of the linear operator  $A$  and (homogeneous) boundary conditions for  $v(x)$ .

Eigenfunction expansions are used to solve time-dependent, linear, nonhomogeneous boundary value problems. The method is simple and intuitively straightforward to implement, and the eigenfunction and eigenvalues frequently have physically meaningful interpretations. Eigenfunctions also provide a natural basis for trial function expansions when used in conjunction with the Galerkin and other projection methods to solve nonlinear problems.

An important set of eigenvalue problems in the form of (17.4) are defined by the class of second-order, regular, Sturm-Liouville problems defined by

$$\frac{1}{x^\alpha v(x)} \frac{d}{dx} \left( x^\alpha p(x) \frac{d\psi}{dx} \right) + q(x) \frac{d\psi}{dx} + g(x)\psi = \lambda\psi \quad x \in (0, 1). \quad (17.5)$$

The solutions  $\psi(x)$  are subject to boundary conditions

$$\begin{aligned} a \frac{d\psi(0)}{dx} + b\psi(0) + a_1 \frac{d\psi(1)}{dx} + b_1\psi(1) &= 0, \\ c \frac{d\psi(1)}{dx} + d\psi(1) + c_0 \frac{d\psi(0)}{dx} + d_0\psi(0) &= 0. \end{aligned} \quad (17.6)$$

Therefore, homogeneous forms of the Dirichlet (zero value), Neumann (zero diffusive flux), mixed, periodic, and semiperiodic boundary conditions can be satisfied. Eigenvalue problems in this form are considered regular in the unit interval if

1.  $p(x)$  and  $v(x)$  do not vanish inside the unit interval;
2.  $1/p(x)$ ,  $g(x)$ , and  $v(x)$  are locally integrable near the endpoints.

The format of the Sturm-Liouville problems to be solved (17.5) was chosen to reflect the form commonly encountered when solving boundary-value problems generated from conservation equations describing the transport of mass, momentum, or energy in engineering modeling problems. In this form,  $p(x)$  can represent a spatially-dependent diffusivity,  $q(x)$  the contribution of a convective flux,  $g(x)$  a spatially nonuniform potential or heat transfer rate, and  $v(x)$  a nonuniform capacitive term. The exponent  $\alpha$  is related to the problem symmetry:  $\alpha = 0$  for slabs,  $\alpha = 1$  for cylindrical geometries, and  $\alpha = 2$  for spherical geometries. It is possible for  $g(x)$  to have more than one finite-jump discontinuity. Finally, we note that the boundary-value problem (17.5) is defined over the unit interval; all problems can be scaled in this way and so this definition poses no limitations.

### 17.3.1 Orthogonality of eigenfunctions

Eigenfunctions computed as the nontrivial solutions to (17.5) satisfying boundary conditions (17.6) will be orthogonal with respect to weighted inner product

$$\langle \psi_i(x), \psi_j(x) \rangle_W = \int_0^1 \psi_i(x) \psi_j(x) W(x) x^\alpha dx \quad (17.7)$$

with

$$W(x) = v(x) \exp \left\{ \int_0^x [v(x') q(x') / p(x')] dx' \right\}.$$

### 17.3.2 A simple example

To prove the orthogonality of the eigenfunctions, consider one of the simplest forms of the eigenvalue problem (17.4):

$$\frac{d^2 v}{dx^2} - \lambda v = 0 \quad \text{over } 0 < x < 1$$

subject to boundary conditions

$$dv(0)/dx = dv(1)/dx = 0.$$

This corresponds to the case of  $\alpha = 0$  and  $W(x) = 1$  in (17.7). It is not hard to show that the solutions  $v(x) = \psi_n(x)$  to the eigenvalue problem are the eigenfunctions  $\psi_n(x) = \cos(\sqrt{-\lambda_n}x)$  with  $\lambda_n = -n^2\pi^2$ ; it has been shown previously (Section 17.1.2) that the eigenfunctions are orthogonal under (17.7) using the inner product definition.

The orthogonality property can be obtained directly from the eigenvalue problem itself: consider the two solutions  $\psi_i$  and  $\psi_j$

$$\begin{aligned}\frac{d^2\psi_i}{dx^2} - \lambda_i\psi_i &= 0 \\ \frac{d^2\psi_j}{dx^2} - \lambda_j\psi_j &= 0\end{aligned}$$

Multiplying the first equation by  $\psi_j$  and the second by  $\psi_i$  and subtracting the two gives

$$\begin{aligned}(\lambda_i - \lambda_j)\psi_i\psi_j &= \psi_j \frac{d^2\psi_i}{dx^2} - \psi_i \frac{d^2\psi_j}{dx^2} \\ &= \frac{d}{dx} \left( \psi_j \frac{d\psi_i}{dx} \right) - \frac{d\psi_j}{dx} \frac{d\psi_i}{dx} - \frac{d}{dx} \left( \psi_i \frac{d\psi_j}{dx} \right) + \frac{d\psi_i}{dx} \frac{d\psi_j}{dx} \\ &= \frac{d}{dx} \left[ \psi_j \frac{d\psi_i}{dx} - \psi_i \frac{d\psi_j}{dx} \right] \\ (\lambda_i - \lambda_j) \int_0^1 \psi_i\psi_j dx &= \left[ \psi_j \frac{d\psi_i}{dx} - \psi_i \frac{d\psi_j}{dx} \right]_0^1 \\ &= 0\end{aligned}$$

The final equality is true because

$$d\psi_i(0)/dx = d\psi_j(0)/dx = d\psi_i(1)/dx = d\psi_j(1)/dx = 0.$$

Because we have assumed that  $\lambda_i \neq \lambda_j$ , the functions  $\psi_i$  and  $\psi_j$  are orthogonal in terms of the inner product (17.7):

$$(\lambda_i - \lambda_j) \int_0^1 \psi_i\psi_j dx = \langle \psi_i, \psi_j \rangle = 0, \quad i \neq j.$$

### 17.3.3 The case $q(x) = 0$

We now consider the eigenvalue problem (17.5-17.6) when  $q(x) = 0$ . Again, the two solutions  $\psi_i$  and  $\psi_j$  correspond to distinct values of the eigenvalues  $\lambda_i$  and  $\lambda_j$ , i.e.,

$$\begin{aligned}\frac{d}{dx} \left( x^\alpha p(x) \frac{d\psi_i}{dx} \right) + x^\alpha v(x)g(x)\psi_i - x^\alpha v(x)\lambda_i\psi_i &= 0 \\ \frac{d}{dx} \left( x^\alpha p(x) \frac{d\psi_j}{dx} \right) + x^\alpha v(x)g(x)\psi_j - x^\alpha v(x)\lambda_j\psi_j &= 0\end{aligned}$$

Multiplying the first equation by  $\psi_j$  and the second by  $\psi_i$  and subtracting the two gives

$$\begin{aligned} (\lambda_j - \lambda_i)x^\alpha v(x)\psi_i\psi_j &= \psi_j \frac{d}{dx} \left( x^\alpha p \frac{d\psi_i}{dx} \right) - \psi_i \frac{d}{dx} \left( x^\alpha p \frac{d\psi_j}{dx} \right) \\ &= \frac{d}{dx} \left[ x^\alpha p \psi_j \frac{d\psi_i}{dx} - x^\alpha p \psi_i \frac{d\psi_j}{dx} \right]. \end{aligned}$$

This can be shown by carrying out the differentiation of the left sides of each of the above equations. Integrating over the unit interval then gives

$$(\lambda_j - \lambda_i) \int_0^1 \psi_i \psi_j x^\alpha v(x) dx = \left[ x^\alpha p \left( \psi_j \frac{d\psi_i}{dx} - \psi_i \frac{d\psi_j}{dx} \right) \right]_0^1$$

For this problem, consider the case  $a_1 = b_1 = c_0 = d_0 =$ , a simplification that eliminates only periodic boundary conditions. In the case of  $a = c = 0$ , the right-hand-side of the equation above vanishes as-is. For  $p, a, c \neq 0$ , solving the boundary conditions for the derivative terms gives

$$\psi'(0) = -\frac{b}{a}\psi(0) \quad \psi'(1) = -\frac{d}{c}\psi(1)$$

and so evaluating the right-hand-side of the integrated equation gives

$$\begin{aligned} &\left[ \psi_j(1) \left( -\frac{d}{c}\psi_i(1) \right) - \psi_i(1) \left( -\frac{d}{c}\psi_j(1) \right) \right] [x^\alpha p]_1 \\ &- \left[ \psi_j(0) \left( -\frac{b}{a}\psi_i(0) \right) - \psi_i(0) \left( -\frac{b}{a}\psi_j(0) \right) \right] [x^\alpha p]_0 = 0. \end{aligned}$$

Because we have assumed that  $\lambda_i \neq \lambda_j$ , the functions  $\psi_i$  and  $\psi_j$  are orthogonal relative to the weight function

$$W(x) = v(x).$$

This concludes the proof that  $\phi_i$  and  $\psi_j$  are orthogonal under the weighted inner product for distinct  $\lambda_i$  and  $\lambda_j$  :

$$(\lambda_j - \lambda_i) \int_a^b \psi_i \psi_j W(x) x^\alpha dx = \langle \psi_i, \psi_j \rangle_W = 0.$$

We will see that weight functions frequently have physical meaning. For example,  $W(r) = v(x)$  represents the fluid velocity field in the Graetz problem, a case study to be discussed later in this text.

### 17.3.4 Completeness

Having established the orthogonality of  $\psi_n$ , an even stronger statement can be made

**Theorem (Sturm-Liouville)** *Let  $p(x)$ ,  $q(x)$ , and  $W(x)$  be regular functions (can be represented pointwise by a convergent power series) for  $x \in (a, b)$ , and let  $p(x)$  and  $W(x)$  be positive for  $x \in [a, b]$ . For  $\psi_i$  satisfying homogeneous boundary conditions, if function  $f(x)$  is bounded and piecewise differentiable for  $x \in (a, b)$ , the series*

$$f(x) = \sum_{n=0}^{\infty} a_n \psi_n(x) \quad \text{with} \quad a_n = \frac{\langle f, \psi_n \rangle_W}{\langle \psi_n, \psi_n \rangle_W} = \frac{\langle f, \psi_n \rangle_W}{\|\psi_n\|_W^2}$$

converges to  $f(x)$  in  $(a, b)$  where  $f$  is continuous, and to  $\frac{1}{2}[f(x^-) + f(x^+)]$  where a finite jump occurs.

Thus, a consequence of this theorem is that the sequence  $\{\psi_n\}_{n=0}^{\infty}$  is said to be *complete*, because we can find no nontrivial function  $f(x)$  whose Fourier constants  $a_n$  all vanish.

### 17.3.5 Basis functions computed as solutions to a Sturm-Liouville problem

Consider computing solutions to Sturm-Liouville problems over  $x_0 < x < x_1$  in the form

$$\frac{1}{x^\alpha} \frac{d}{dx} x^\alpha \frac{d\phi}{dx} = \lambda \phi$$

subject to

$$\begin{aligned} a \frac{d\phi(x_0)}{dx} + b\phi(x_0) &= 0 \\ c \frac{d\phi(x_1)}{dx} + d\phi(x_1) &= 0. \end{aligned}$$

A class derived from the `basisfun` class has been developed to create a `basisfun` object containing basis functions in the form of eigenfunctions corresponding to the problem above.

<b>basisfunsl</b>
<code>basisfun : basisfun</code> <code>a : double</code> <code>b : double</code> <code>c : double</code> <code>d : double</code>
<code>basisfunsl(X,coname,a,b,c,d)</code> <code>display(A)</code>

We can demonstrate the use of this class with the example given in Section 17.3.2; we first set up the physical domain (that includes the endpoints):  $0 \leq x \leq 1$  and then compute and truncate the basis (eigen)functions:

```
>> X = quadgrid('slab',40,'x',[0,1]);
>> P = basisfunsl(X,'x',1,0,1,0);
>> P = truncate(P,5);
>> plot(P)
```

As can be seen in Fig. 17.2, the first five eigenfunctions clearly are  $\cos n\pi x$ ,  $n = 0, \dots, 4$ .

Furthermore, we can assess the accuracy of the eigenvalues computed using the numerical procedure by dividing each by  $\pi^2$ ; the result should be  $-n^2$ ,  $n = 0, \dots, 4$ :

```
>> eig(P)/pi^2
ans =
-0.0000
-1.0000
-4.0000
-9.0000
-16.0000
```



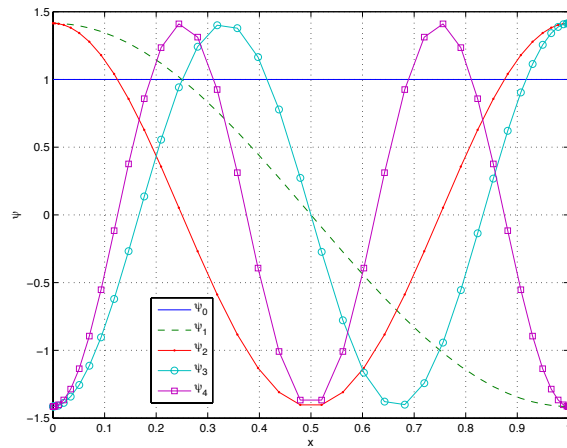


Figure 17.2: First five normalized eigenfunctions computed using basisfunsl.m.

Finally, the orthogonality of the eigenfunctions is established by computing the weighted inner product of all the eigenfunctions with all eigenfunctions:

```
>> wip(P,P)
ans =
    1.0000e+00    3.1764e-14   -1.4319e-14   -6.8658e-15   -3.1489e-16
    3.1781e-14    1.0000e+00    4.7296e-15   -2.0335e-15   -1.1423e-16
   -1.4335e-14    4.7774e-15    1.0000e+00   -1.3995e-14   -2.9135e-15
   -6.9046e-15   -2.0388e-15   -1.3963e-14    1.0000e+00   -1.3745e-14
   -2.9882e-16   -1.3842e-16   -2.9059e-15   -1.3737e-14    1.0000e+00
```

From the matrix above, we observe that all off-diagonal elements approach round-off error and all diagonal elements are unity, indicating the eigenfunctions are indeed orthonormal.

### 17.3.6 Another example

Consider the eigenvalue problem

$$\frac{d^2\psi}{dz^2} + \lambda^2\psi = 0 \quad \text{over } 0 < z < L$$

subject to boundary conditions

$$P_e\psi(0) - \frac{d\psi(0)}{dz} = 0 \quad \text{and} \quad \frac{d\psi(L)}{dz} = 0.$$

For this problem it is easy to see that

$$\psi_n(z) = A_n \sin \lambda_n z + B_n \cos \lambda_n z$$

satisfies the eigenvalue differential equation. Substituting this function into each of the boundary conditions gives

$$\begin{aligned} P_e B_n \cos \lambda_n + A_n \lambda_n \cos \lambda_n &= 0 \\ -A_n \lambda_n \cos \lambda_n L + B_n \lambda_n \sin \lambda_n L &= 0 \end{aligned}$$

Defining  $C_n = A_n/B_n$  we find from the first equation  $C_n = -P_e/\lambda_n$  and so the second equation provides the nonlinear relationship for the eigenvalues

$$P_e \cos \lambda_n L + \lambda_n \sin \lambda_n L = 0 \implies P_e + \lambda_n \tan \lambda_n L = 0.$$

Computing solutions to the equation above gives the sequence of orthogonal basis functions

$$\psi_n = A_n \left[ \sin \lambda_n z - \frac{\lambda_n}{P_e} \cos \lambda_n z \right].$$

## 17.4 The Galerkin projection

Applications of the eigenfunction expansion technique are limited to certain linear distributed parameter systems. The Galerkin technique introduced in this section can be seen as an extension of the eigenfunction expansion to nonlinear systems. Its applicability is increase by choosing to express the solution in terms of trial functions other than the eigenfunctions of the linearized system.

Consider a truncated eigenfunction expansion solution to the nonhomogeneous, linear problem

$$\frac{\partial u}{\partial \tau} = L(u) + F(x, \tau)$$

subject to the homogeneous boundary conditions. If we think of  $F$  as resulting from the linearization of a nonlinear function  $N(u)$  at some solution profile  $u(x, \hat{\tau})$ , the eigenfunctions  $\psi_n$  will not change and  $F$  will exist in the function space spanned by the  $\psi_n$ . The conclusion we reach is that if the linearized system possesses a particular solution (determined by the initial condition  $u(x, \hat{\tau})$ ), the nonlinear system should have a solution that can be expressed in the form

$$u^N(x, \tau) = \sum_{n=1}^N a_n(\tau) \psi_n(x). \quad (17.8)$$

Substituting the truncated eigenfunction function expansion  $u^N$  into the nonhomogeneous problem and approximating the forcing function by the eigenfunction expansion  $F = \sum f_n \psi_n$  gives the residual

$$\mathcal{R}^N(x, \tau) = \sum_{n=1}^N \dot{a}_n(\tau) \psi_n(x) + \sum_{n=1}^N \lambda_n a_n(\tau) \psi_n(x) - \sum_{n=1}^N f_n(\tau) \psi_n(x).$$

Minimizing the square of the residual 2-norm  $\|\mathcal{R}^N\|_2^2$  means minimizing

$$\sum_{n=1}^N [\dot{a}_n(\tau) + \lambda_n a_n(\tau) - f_n(\tau)]^2$$

(by the orthogonality of the  $\psi_n$ ) which gives the conditions

$$\dot{a}_n(\tau) + \lambda_n a_n(\tau) - f_n(\tau) = 0 \quad n = 1, \dots, N$$

that must be solved to drive the residual to zero. We note that this formulation is equivalent to

$$\begin{aligned} \langle \dot{u}^N, \psi_m \rangle_w &= \langle L(u^N), \psi_m \rangle_w + \langle F, \psi_m \rangle_w \\ \dot{a}_m &= -\lambda_m a_m + \langle F, \psi_m \rangle_w / \langle \psi_m, \psi_m \rangle_w. \end{aligned}$$

We can generalize the last idea by considering particular solutions to

$$\frac{\partial u}{\partial \tau} = G(u)$$

of the form (17.8). Because the trial functions  $\psi_n$  are chosen as the eigenfunctions of the homogeneous, linearized problem with linear, homogeneous boundary conditions, *any linear combination of the trial functions  $\psi_n$  will also satisfy the boundary conditions of the problem  $\dot{u} = G(u)$* . This means we need not further consider the role of boundary conditions in this problem. The coefficients  $a_n(\tau)$  are governed by the set of (possibly nonlinear) ordinary differential equations in time

$$\frac{da_n}{d\tau} = g(\mathbf{a}, \tau), \quad n = 0, \dots, M, \quad \mathbf{a} = [a_1, \dots, a_M]^T$$

determined so that the *residual*

$$\mathcal{R}(x, \tau) = \frac{\partial}{\partial \tau} \sum_{n=1}^N a_n(\tau) \psi_n(x) - G \left( \sum_{n=1}^N a_n(\tau) \psi_n(x) \right)$$

is minimized, subject to the initial conditions  $a_n(\tau = 0)$ . This minimization is in the 2-norm sense, and so we wish to determine

$$\min_{a_1, \dots, a_N} \|\mathcal{R}\|_2^2$$

for any specified  $M$ . We develop the Galerkin procedure in two steps. First, because the  $\psi_n$  are complete, they can approximate any function in the function space they span, and so we can write the *convergent* series

$$\mathcal{R}^N(x, \tau) = \sum_{i=1}^N b_i(\tau) \psi_i(x).$$

It is important to stress that if we can drive the truncated, trial function expansion approximation to the residual to zero, we will also force the true residual  $\mathcal{R}$  to zero.

We can now determine the coefficients  $b_n$  with the standard procedure of minimizing the norm

$$\mathcal{D} = \left\| \sum_i b_i \psi_i - \mathcal{R} \right\|_2^2$$

over  $b_n$ , giving

$$b_n c_n = \langle \mathcal{R}, \psi_n \rangle_w \quad \text{or} \quad b_n = \frac{\langle \mathcal{R}, \psi_n \rangle_w}{\langle \psi_n, \psi_n \rangle_w}.$$

This residual can be minimized by driving as many of the coefficients  $b_n$  to zero as possible, and so with the first  $N$  eigenfunctions we can write

$$\begin{aligned} \langle \mathcal{R}, \psi_n \rangle_w &= \frac{\partial}{\partial \tau} a_n c_n - \langle L(u), \psi_n \rangle_w - \langle N(u), \psi_n \rangle_w \\ &= \frac{\partial}{\partial \tau} a_n c_n + \lambda_n a_n c_n - \langle N(u), \psi_n \rangle_w \\ &= 0 \end{aligned}$$

and so integrating the ordinary differential equations in time

$$\frac{da_n}{d\tau} = -\lambda_n a_n + \frac{\langle N(u), \psi_n \rangle_w}{\langle \psi_n, \psi_n \rangle_w} \quad n = 0, \dots, N$$

subject to initial conditions  $\langle u, \psi_n \rangle_w = \langle u_0, \psi_n \rangle_w$  minimizes the residual.

### 17.4.1 Accuracy

Consider the particular solution to

$$\frac{\partial u}{\partial \tau} = G(u) \quad \text{in } \Omega \quad (17.9)$$

subject to boundary and initial conditions. If  $u^N = \sum_{n=0}^N a_n \psi_n$  and satisfies the boundary conditions for all combinations of the mode amplitudes  $\mathbf{a}$ , two sources of simulation error can be identified:

1. A residual can be defined as  $\mathcal{R}(x, t) = \dot{u}^N - G(u^N)$ . This measures the time rate of change of the error in the state profile  $u(x, t)$ . If the scalar quantity

$$\hat{\mathcal{R}}(t) = \|\dot{u}^N - G(u^N)\|$$

is significantly smaller than a representative quantity, such as  $\|u(x, t)\|$ , we can consider this to be an accurate *general* solution. Simulation error then depends on the accuracy of the integrator.

2. An accurate *particular* solution has an accurate general solution and accurately captures the initial condition  $u^0(x)$

$$\|u^N(x, t=0) - u^0(x)\| \rightarrow 0 \quad \text{as} \quad N \rightarrow \infty$$

In addition to the accuracy criteria above, we can pose other measures of simulation accuracy, such as whether the solution explicitly conserves certain quantities (e.g., mass or energy).

## 17.5 Assessing discretized solution accuracy

The overall process for assessing the accuracy of a numerical simulation is to

1. Determine the accuracy of the numerical solution relative to the original modeling equations; and if the numerical solution is accurate, then
2. Assess whether the modeling equations accurately represent the physical phenomena under study.

Unfortunately, many computational studies do not directly address the first step, or at least do not provide details justifying the assumption that computed solutions accurately portray solutions to the modeling equations. In this Chapter, we take a closer look at assessing the accuracy of some of our computed solutions.

## 17.6 The Gibb's phenomenon and filtering

Solutions to boundary-value problems (BVPs) with spatially discontinuous forcing terms can exhibit poor convergence performance when the solutions are represented by a smooth, continuous, globally defined basis function expansion. Likewise, initial conditions that do not satisfy the problem boundary conditions, or discontinuities in the boundary conditions themselves, can result in solutions with non-diminishing oscillations characteristic of the Gibbs phenomenon. Even in cases where satisfactory solutions are found, accurate computation of secondary quantities, such as a flux at a specified spatial location, can be affected by the Gibbs oscillations both near to and away from the discontinuity.

To reduce the effects Gibbs oscillations have on solutions to BVPs, we will incorporate recent developments in spectral filtering methods in collocation discretization procedures based on globally defined trial function expansions. We take advantage of the computational methods that make possible very high discretization degrees developed in previous chapters, and use the filtering methods as a means of improving discretization method convergence performance, rather than the more traditional application as a post-processing step. We consider global polynomial collocation for time integration of the nonlinear CVD thermal dynamics model as a representative test case for the numerical methods presented in this chapter.

To illustrate basic issues in spectral filtering and the convergence of global polynomial approximations, consider projecting the piece-wise continuous function defined on the unit interval:

$$f(t) = \begin{cases} 0 & 0 \leq t < 0.55 \\ 1 & 0.55 < t \leq 1 \end{cases} \quad (17.10)$$

onto the sequence of trial functions  $\psi_{n+1}(t) = P_n(t)/\sqrt{c_n}$  where the shifted Legendre polynomials  $P_n(t)$  are defined by the recurrence formula

$$(n+1)P_{n+1}(t) = (2n+1)(2t-1)P_n(t) - nP_{n-1}(t) \quad (17.11)$$

with  $P_0(t) = 1$  and  $P_1(t) = 2t - 1$ . The resulting sequence of functions is orthogonal with respect to inner product

$$\langle P_m, P_n \rangle = \int_0^1 P_m(t)P_n(t) dt = c_n \delta_{m,n}$$

with  $c_n = 1/(2n+1)$ . Examining the projection  $f_N(t)$  of  $f(t)$  onto the space spanned by the  $\psi_n(t)$ ,  $n = 1, \dots, N$

$$f_N(t) = \sum_{n=1}^N a_n \psi_n(t) \quad (17.12)$$

with

$$a_n = \langle f(t), \psi_n(t) \rangle,$$

we find the expected Gibbs phenomenon (Fig. 17.3). Three important features are observed in the projection results. First, oscillatory overshoot behavior near the discontinuity does not vanish with increasing truncation number  $N$ , but asymptotically approaches a constant value of approximately 8.949% of the jump magnitude [13] as  $N$  grows large. Second, the oscillatory effects are exhibited far from the discontinuity: for this problem, pointwise convergence away from the discontinuity and the interval endpoints is of order  $1/N$ , while at the endpoints convergence is proportional to  $1/\sqrt{N}$  (cf. the exponential convergence when  $f(t)$  is continuous). The slower convergence rate near the endpoints is directly attributable to the property of the polynomial sequence—that

$$|P_n(t)| < 1, \quad 0 < t < 1$$

but

$$P_n(0) = (-1)^n, \quad P_n(1) = 1.$$

A third observation comes from comparing consecutive projections  $f_{N-1}$  and  $f_N$  (Fig. 17.3). When the  $\psi_n$  are taken from an orthogonal sequence, the oscillations tend to become out of phase away from the discontinuity, and the features near the discontinuity show little change. Therefore, we can average the two consecutive projections in the hope of obtaining better pointwise convergence performance.

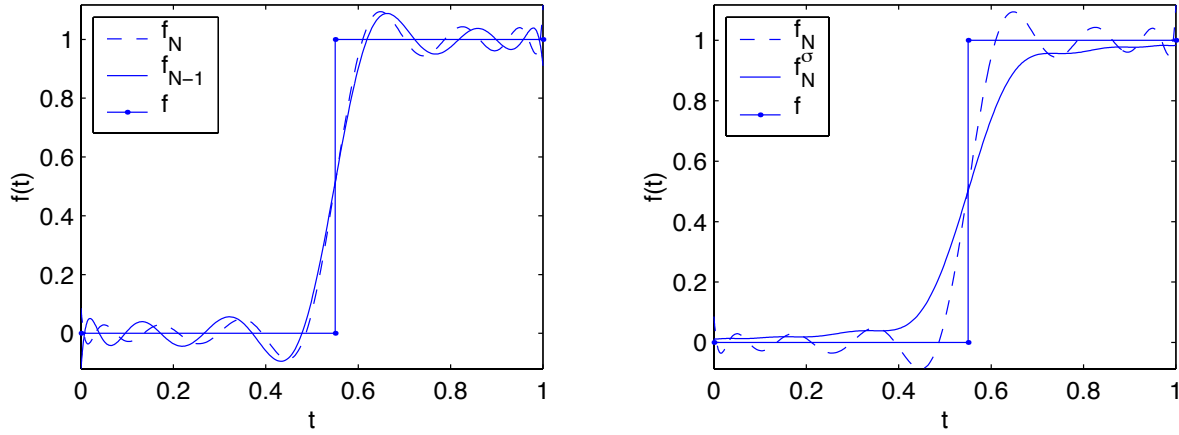


Figure 17.3: *Gibbs oscillations in two consecutive projections (left) for  $N = 15$ ; first-order filtered results (right).*

Continuing this averaging operation on the partial sums extends the smoothing behavior towards the discontinuity (Fig. 17.3, right). This averaging operation can be expressed by rewriting the partial sum (17.12) in terms of the spectrally filtered form

$$f_N^\sigma(t) = \sum_{n=1}^N a_n \sigma_n \psi_n(t)$$

with the  $\sigma_n$  derived to give the equivalent to the arithmetic mean of the partial sums  $f_n$ ,  $n = 1, \dots, N$  (the Cesaro sum):

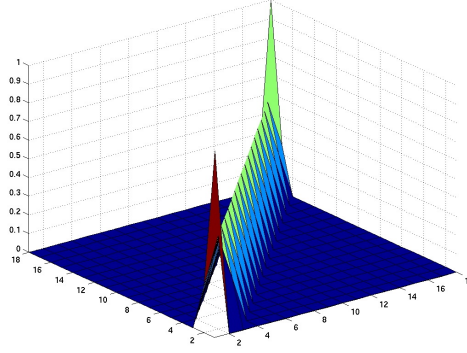
$$\sigma_n = 1 - \frac{n-1}{N} = 1 - \eta$$

with  $\eta = (n-1)/N$ . A more uniform, but still slow, pointwise convergence rate was observed and discussed in [1]. In fact, for the representative point chosen in that study, the average partial sum converges at essentially the same rate (but more uniformly) as the non-filtered projection.

### 17.6.1 Filters

The averaged partial sum filter belongs to the class of first-order filters; a filter of order  $p$  is a  $C^{p-1}$  function whose first  $p-1$  derivatives vanish at  $\eta = 0$ ; specific definitions can be found in [14]. In the cited reference, it was shown that higher-order filters can provide better point-wise convergence when the filter order grows with truncation number  $N$ . One such filter is the Vandeven filter [28], derived to produce exponential pointwise convergence of the filtered partial sums away from the discontinuity for periodic functions:

$$\begin{aligned} \sigma(\eta) &= 1 - \frac{(2p-1)!}{(p-1)!^2} \int_0^\eta [s(1-s)]^{p-1} ds \\ &= 1 - \frac{(2p-1)!}{(p-1)!^2} \left[ \frac{\eta^p}{p} + \sum_{n=1}^{p-1} \frac{(-1)^n (p-1)!}{n! (p-1-n)!} \frac{\eta^{n+p}}{n+p} \right]. \end{aligned}$$

Figure 17.4: A binomial physical-space filter for  $N = 18$ .

To study the potential improvements in pointwise convergence when using the higher-order filters, we examined  $|f(t_0) - f_N^g(t_0)|$  as a function of truncation number  $N$  and filter order  $p$  for the representative point  $t_0 = 0.4$  in [1]. It was observed that convergence for the second-order filter is, in general, better than the other filter orders for  $N < 65$ . For higher truncation number values, the eighth-order filter gives improved convergence at the point  $t_0$ , a result consistent with [14].

### 17.6.2 Physical-space filters

For use with the collocation discretization techniques discussed in previous chapters, it makes sense to consider the physical space equivalents to the spectral filters; these physical space equivalents and “composite” versions were discussed in detail in [1]. Likewise, physical space filters with no direct spectral filter counterpart were discussed, such as the simple binomial filter, commonly used for image processing applications [17]. It is defined by

$$P_{sp}^b(i, i) = 1 \quad \text{for } i = 1, N$$

$$P_{sp}^b(i, i-1) = P_{sp}^b(i, i+1) = 1/4, \quad P_{sp}^b(i, i) = 1/2 \quad \text{for } i = 2, \dots, N-1$$

The filter is shown in Fig. 17.4. We note that higher-order binomial filters, necessary for cases where stronger filtering is required, can be obtained by repeated applications of this filter, i.e.,  $(\mathbf{P}_{sp}^b)^2$ ,  $(\mathbf{P}_{sp}^b)^3$ , and so on. We will refer to  $(\mathbf{P}_{sp}^b)^n$  as a binomial filter of order  $n$ .

The binomial filter produced improved convergence near the interval endpoints and the reduced overshoot near the discontinuity. Results for the binomial filter are presented in Fig. 17.5

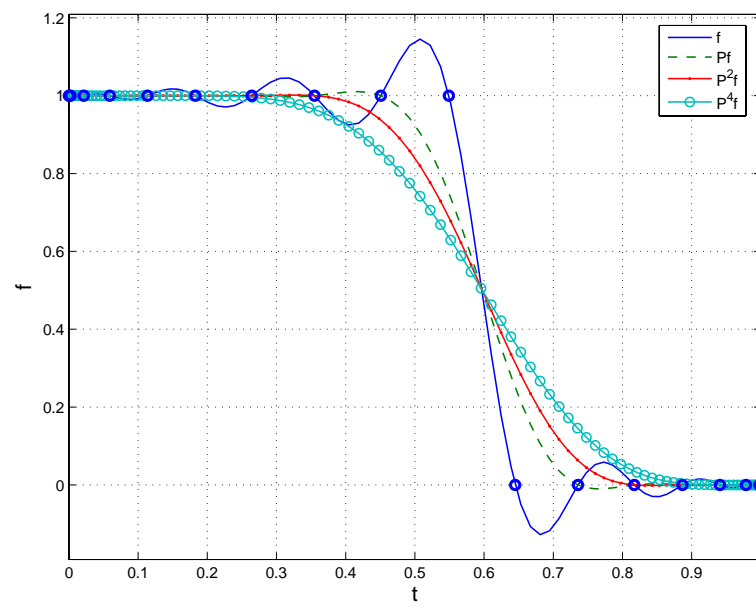


Figure 17.5: Unfiltered and filtered collocation approximations to the step function demonstrating the filtering achieved with various orders of binomial filters.



## Chapter 18

# Polynomial collocation for nonlinear/linear BVPs

Only global polynomial interpolation methods will be discussed in this chapter. While there are numerous methods for representing curves or interpolating data points with locally defined functions, (such as the spline methods), we will focus only on the global methods to be consistent with the later chapters on global spectral methods.

### 18.1 Polynomial interpolation and quadrature

An important application of polynomial interpolation methods is in the analysis and use of experimental data that is represented in the form of a set discrete measurements. Consider data tabulated in the following form:

$x$	$T$ (K)
$x_1$	$T_1$
$x_2$	$T_2$
$x_3$	$T_3$
$\vdots$	$\vdots$
$x_{N+1}$	$T_{N+1}$

Our objective is to find the interpolating polynomial

$$T(x) = a_0 + a_1x + a_2x^2 + \dots + a_Nx^N = \sum_{n=0}^N a_nx^n \quad (18.1)$$

using the experimental data. We note the following:

1. At least  $N + 1$  data points  $(x_i, T_i)$  are required to define (18.1);
2. The polynomial (18.1) is said to have **degree**  $N$ ; the number of roots of a (non-zero) polynomial and the degree of the polynomial are equal.

### 18.1.1 Uniqueness of the interpolating polynomial

A *unique* interpolating polynomial curve results in the case where  $N + 1$  independent experimental measurements are available to define (18.1). If this were not the case, it would be possible to define at least two different polynomials

$$T^a(x) = \sum_{n=0}^N a_n x^n \quad \text{and} \quad T^b(x) = \sum_{n=0}^N b_n x^n$$

that pass through the same set of points  $(x_i, T_i)$ ,  $i = 1, \dots, N + 1$ . Subtracting term-by-term gives a third polynomial

$$P^c(x) = \sum_{n=0}^N (a_n - b_n) x^n = \sum_{n=0}^N c_n x^n$$

which must now pass through the points  $(x_i, 0)$ ,  $i = 1, \dots, N + 1$ . Because a degree  $N$  non-zero polynomial can have only  $N$  zeros, the only possible polynomial for  $P^c$  is the zero polynomial, forcing  $a_n = b_n$  for all  $n$ .

### 18.1.2 The Vandermonde array

For relatively low-degree polynomials (e.g.,  $N > 10$ ), the polynomial coefficients can be calculated directly. Given  $M \geq N + 1$  data points,

$$\mathbf{V}^{M \times N+1} \mathbf{a}^{N+1 \times 1} = \mathbf{T}^{M \times 1} \quad (18.2)$$

with

$$V_{m,n} = x_m^{n-1} \quad m = 1, \dots, M, \quad n = 1, \dots, N + 1$$

and  $x_1 < x_2 < \dots < x_M$ . A least-squares solution procedure must be used when  $M > N + 1$ .

The array  $\mathbf{V}$  is a Vandermonde array; it is possible to write an explicit equation for the determinant of this array and to show that the determinant is never zero [27] when the points  $x_i$  are unique. This means  $\mathbf{V}$  is always invertible (for unique  $x_i$ ); however, for finite-precision computations, the array becomes numerically singular for (approximately)  $M > 12$ , making the direct computation of the coefficients  $a_i$  impractical for large  $M$ .

## 18.2 Lagrange interpolation

A more direct approach in the case  $M = N + 1$  involves the use of the Lagrange interpolation polynomials. Consider the following polynomial:

$$l_i(x) = \frac{(x - x_1)(x - x_2) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_M)}{(x_i - x_1)(x_i - x_2) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_M)} = \frac{\prod_{j=1, j \neq i}^M (x - x_j)}{\prod_{j=1, j \neq i}^M (x_i - x_j)} \quad (18.3)$$

which has the following property:

$$l_i(x) = \begin{cases} 1 & \text{if } x = x_i \\ 0 & \text{if } x = x_j, \quad j = 1, \dots, M, \quad j \neq i \end{cases}$$

The polynomial  $l_i(x)$  is known as the Lagrange building block polynomial. In this formulation, the interpolating polynomial can be written directly as

$$T^M(x) = \sum_{i=1}^M l_i(x) T_i. \quad (18.4)$$

The importance of the polynomial uniqueness theorem discussed in Section 18.1.1 is that for  $M = N + 1$  data points, the following polynomials are *identical*, both at the interpolation points and in between:

$$\begin{aligned} T(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_Nx^N \\ &= l_1(x)T_1 + l_2(x)T_2 + \cdots + l_M(x)T_M \end{aligned}$$

where the  $a_j$  are computed using the Vandermonde array (18.2).

### 18.2.1 Runge's divergence phenomenon

It can be observed that interpolation polynomials can behave extremely badly when the data points have equidistant abscissae. This can be traced directly to the building block polynomials; an example for  $M = 11$  interpolation points is shown in Fig. 18.1, where we note the large deviation in the building block polynomials towards the interval endpoints. Spacing the interpolation points in such a way that the density increases towards the interpolation interval endpoints produces considerably better polynomial behavior.

## 18.3 Neville's algorithm

It is computationally more efficient and accurate to use the interpolation methods in an iterated fashion. Following the derivation presented by Stroud [27], we begin the derivation of Neville's algorithm by defining the value of  $P(x)$  at  $x$  of an  $n$  degree polynomial that interpolates the points  $x_1, x_2, \dots, x_{n+1}$  as

$$P_n(x; x_1, x_2, \dots, x_{n+1}).$$

In general, this value can be computed by interpolating the two  $n - 1$  degree polynomials:

$$P_n(x; x_i, x_{i+1}, \dots, x_{i+n}) = \frac{(x_{i+n} - x)P_{n-1}(x; x_i, x_{i+1}, \dots, x_{i+n-1}) - (x_i - x)P_{n-1}(x; x_{i+1}, x_{i+2}, \dots, x_{i+n})}{x_{i+n} - x_i}$$

For example,

$$P_2(x; x_1, x_2, x_3) = \frac{(x_3 - x)P_1(x; x_1, x_2) - (x_1 - x)P_1(x; x_2, x_3)}{x_3 - x_1}$$

Therefore, if

$$\begin{aligned} x = x_1 & \quad P_2(x_1; x_1, x_2, x_3) = P_1(x_1; x_1, x_2) \quad \text{which is correct,} \\ x = x_3 & \quad P_2(x_3; x_1, x_2, x_3) = P_1(x_3; x_2, x_3) \quad \text{which is correct,} \\ x = x_2 & \quad P_2(x_2; x_1, x_2, x_3) = \frac{(x_3 - x_2)P_1(x_2; x_1, x_2) - (x_1 - x_2)P_1(x_2; x_2, x_3)}{x_3 - x_2}. \end{aligned}$$

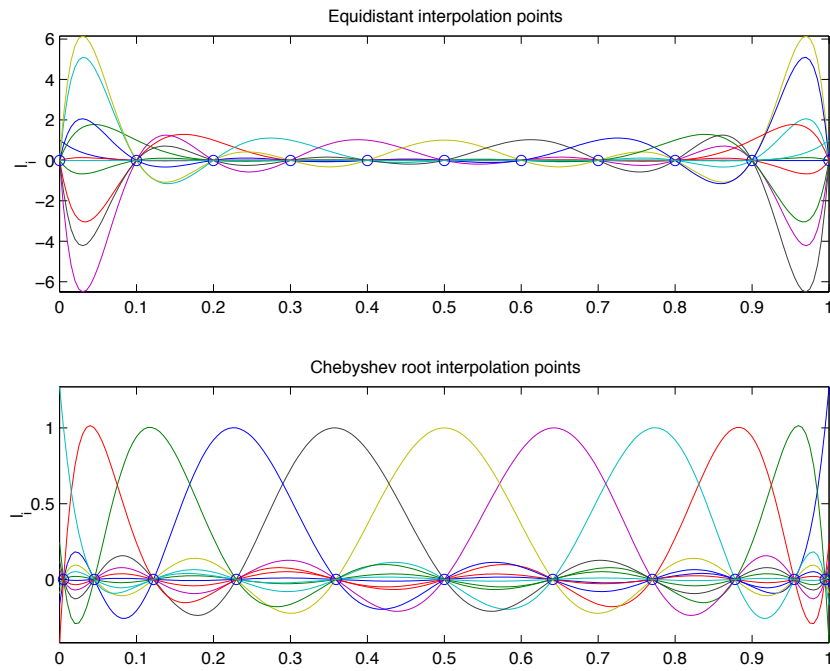


Figure 18.1: Comparison of building block polynomials for equidistant data points (top) and the polynomials produced when the roots of a Chebyshev polynomial are used for the abscissa points (bottom).

The last equality also is valid because  $P_1(x_2; x_1, x_2) = P_1(x_2; x_2, x_3)$ .

What is produced by this algorithm is a table that is filled in from left to right; for the CVD example, the table would look like the following:

$x_1$	$P_0(x; x_1) = T_1$	0	0	0	0
$x_2$	$P_0(x; x_2) = T_2$	$P_1(x; x_1, x_2)$	0	0	0
$x_3$	$P_0(x; x_3) = T_3$	$P_1(x; x_2, x_3)$	$P_2(x; x_1, x_2, x_3)$	0	0
$x_4$	$P_0(x; x_4) = T_4$	$P_1(x; x_3, x_4)$	$P_2(x; x_2, x_3, x_4)$	$P_3(x; x_1, x_2, x_3, x_4)$	0
$x_5$	$P_0(x; x_5) = T_5$	$P_1(x; x_4, x_5)$	$P_2(x; x_3, x_4, x_5)$	$P_3(x; x_2, x_3, x_4, x_5)$	$P_4(x; x_1, x_2, x_3, x_4, x_5)$

which for the case of  $x = 0.2$  gives:

0	672.6700				
0.4	626.6700	649.6700			
0.6	602.3300	651.0100	650.1167		
0.8	577.3300	652.3300	650.3500	650.1750	
1	553.3300	649.3300	655.3300	648.6900	649.8780

## 18.4 Discrete differentiation operations

There are many cases where the derivative of an interpolation polynomial is necessary (such as the Newton procedure described in the following section):

$$\frac{dT^M(x)}{dx} = \sum_{i=1}^M \frac{dl_i(x)}{dx} T_i.$$

Rather than computing  $dl_i/dx$  directly from the definition of the building block polynomials, we make use of the uniqueness theorem (Section 18.1.1) to simplify the computations. First, consider the case  $M = N + 1$  and writing

$$\begin{aligned} T^M(x) &= \sum_{m=1}^M l_m(x) T_m \\ T^M(x_i) &= \sum_{m=1}^M l_m(x_i) T_m \end{aligned}$$

or in matrix form

$$\begin{aligned} \mathbf{T} &= \mathbf{L}\mathbf{T} \\ &= \mathbf{I}\mathbf{T}. \end{aligned}$$

The equivalence of the  $\mathbf{L}$  and identity arrays follows from the definition of the building block polynomials. Using the uniqueness theorem, we can write the equivalent discretized system:

$$\mathbf{T} = \mathbf{L}\mathbf{T} = \mathbf{V}\mathbf{a}$$

so making use of the invertability of the Vandermonde array,

$$\mathbf{a} = \mathbf{V}^{-1}\mathbf{T}$$

Defining the column vector  $d\mathbf{T}/dx$  with the elements  $dT(x_i)/dx$  we can write *discrete ordinate formulation*:

$$\begin{aligned} \frac{d\mathbf{T}}{dx} &= \frac{d}{dx} \mathbf{V}\mathbf{a} \\ &= \left[ \frac{d}{dx} \mathbf{V} \right] \mathbf{V}^{-1}\mathbf{T} \\ &= \mathbf{A}\mathbf{T} \end{aligned}$$

where the elements of  $d\mathbf{V}/dx$  are defined by

$$\begin{aligned} \frac{dV_{i,j}}{dx} &= (j-1)x_i^{j-2}, \quad j \neq 1 \\ &= 0, \quad j = 1 \end{aligned}$$

We note that besides any roundoff errors, this computation is *exact* because the derivative operations reduce the degree of the polynomial, and so no accuracy is lost in the discrete representation of the derivative curve.

## 18.5 Quadrature

Just as with differentiation, we would like to be able to evaluate the integral

$$\begin{aligned} \int_{x_1}^{x_M} f(x) dx &= \sum_{i=1}^M \int_{x_1}^{x_M} l_i(x) dx f(x_i) \\ &= \sum_{i=1}^M w_i f(x_i) \end{aligned}$$

in a discrete fashion, where the  $w_i$  are known as the quadrature weights defined as

$$w_i = \int_{x_1}^{x_M} l_i(x) dx.$$

As in the case of the discretized differentiation operation, we want to avoid direct integration of the Lagrange polynomial. Returning to the CVD temperature example, the procedure follows the derivation of [12] and begins by defining the Taylor's series expansion for the temperature function  $T(x)$ :

$$T(x) = T(0) + xT'(0) + \frac{x^2}{2} T''(0) + \frac{x^3}{6} T'''(0) + \cdots + \frac{x^n}{n!} T^{(n)}(0) + \dots$$

where  $T^{(n)}(x)$  denotes the  $n$ th derivative of  $T(x)$ . Integrating over the range  $[x_1, x_M]$ , the exact and quadrature results can be written as

$$\begin{aligned} \int_{x_1}^{x_M} T(x) dx &= (x_M - x_1)T(0) + \frac{(x_M^2 - x_1^2)}{2} T'(0) + \frac{(x_M^3 - x_1^3)}{6} T''(0) + \cdots + \frac{(x_M^n - x_1^n)}{n!} T^{(n-1)}(0) + \dots \\ &= \sum_{m=1}^M w_m T(0) + \sum_{m=1}^M w_m x_m T'(0) + \sum_{m=1}^M w_m \frac{1}{2} x_m^2 T''(0) + \cdots + \sum_{m=1}^M w_m \frac{x_m^{n-1}}{(n-1)!} T^{(n-1)}(0) + \dots \end{aligned}$$

Equating like coefficients of  $T^n$  gives

$$\begin{aligned} \sum_{m=1}^M w_m &= x_M - x_1 \\ \sum_{m=1}^M w_m x_m &= \frac{1}{2}(x_M^2 - x_1^2) \\ &\vdots \\ \sum_{m=1}^M w_m \frac{x_m^{n-1}}{(n-1)!} &= \frac{(x_M^n - x_1^n)}{n!} \\ &\vdots \end{aligned}$$

or

$$\mathbf{V}^T \mathbf{w} = \begin{bmatrix} x_M - x_1 \\ \vdots \\ \frac{x_M^n - x_1^n}{n} \\ \vdots \\ \dots \end{bmatrix}$$

which can be solved for the quadrature weights  $\mathbf{w}$ .

## 18.6 High-degree interpolation

Using Lagrange interpolation directly or Neville's algorithm is impractical for high degree polynomials (e.g., greater than 10). Instead, we take the following approach, which is based on the uniqueness theorem. Consider representing a function  $T(x)$  over the interval  $x_0 \leq x \leq x_1$  by a sequence of Chebyshev polynomials of degree  $n$  ( $p_n(x)$ ):

$$\begin{aligned} T(x) &= a_0 p_0(x) + a_1 p_1(x) + a_2 p_2(x) + a_3 p_3(x) + \dots \\ T(x_i) &= a_0 p_0(x_i) + a_1 p_1(x_i) + a_2 p_2(x_i) + a_3 p_3(x_i) + \dots \\ &= \mathbf{p}^T(x_i) \mathbf{a} \end{aligned}$$

where the Chebyshev polynomials are generated by the recursive relationship

$$\begin{aligned} p_0(x) &= 1 \\ p_1(x) &= 2(x - x_0)/(x_1 - x_0) - 1 \\ p_n(x) &= (4(x - x_0)/(x_1 - x_0) - 2)p_{n-1}(x) - p_{n-2}(x) \end{aligned}$$

If we have  $x_i$ ,  $i = 1, \dots, I$ , we can write

$$\mathbf{T} = \mathbf{P}^T \mathbf{a} \quad \text{with} \quad P_{i,j} = p_j(x_i)$$

and so interpolation to an arbitrary point  $x_k \in [x_0, x_1]$  is

$$T(x_k) = \mathbf{p}^T(x_k) [\mathbf{P}^T]^{-1} \mathbf{T}$$

### 18.6.1 2-dimensional interpolation

Now consider the two-dimensional polynomial expansion

$$\begin{aligned} T(x, y) &= a_{0,0} p_0(x) q_0(y) + a_{0,1} p_0(x) q_1(y) + a_{0,2} p_0(x) q_2(y) + \dots \\ &\quad + a_{1,0} p_1(x) q_0(y) + a_{1,1} p_1(x) q_1(y) + a_{1,2} p_1(x) q_2(y) + \dots \\ &\quad + a_{2,0} p_2(x) q_0(y) + a_{2,1} p_2(x) q_1(y) + a_{2,2} p_2(x) q_2(y) + \dots \\ &= \mathbf{p}^T(x) \mathbf{A} \mathbf{q}(y) \quad A_{i,j} = a_{i,j} \end{aligned}$$

and so if our solution is known at the grid points  $(x_i, y_j)$ ,

$$\mathbf{T} = \mathbf{P}^T \mathbf{A} \mathbf{Q}$$

and so at the arbitrary point  $(x_k, y_l)$

$$T(x_k, y_l) = \mathbf{p}^T(x_k) [\mathbf{P}^T]^{-1} \mathbf{T} \mathbf{Q}^{-1} \mathbf{q}(y_l)$$

### 18.6.2 Collocation discretization

The quadrature point (collocation) grid is defined first

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{M-1} \\ z_M \end{bmatrix} = \begin{bmatrix} 0 \\ z_2 \\ \vdots \\ z_{M-1} \\ 1 \end{bmatrix}$$

on which the discretized dimensionless concentration vector  $\mathbf{x}$  is defined such that  $x_i = x(z_i)$ ; the discrete differentiation arrays associated with quadrature grid  $\mathbf{z}$  also are defined

$$\mathbf{A} = \frac{d}{dz} \quad \mathbf{B} = \frac{d^2}{dz^2}.$$

Finally, we denote the  $i$ -th row of arrays  $\mathbf{A}$  and  $\mathbf{B}$  as  $\mathbf{A}_i$  and  $\mathbf{B}_i$ , respectively.

The collocation procedure consists of defining  $M$  collocation-discretized equations and solving the resulting system for  $\mathbf{x}$ . The reactor inlet boundary condition can be approximated by the collocation-discretized representation

$$\frac{1}{P_e} \mathbf{A}_1 \mathbf{x} - x_1 = -1. \quad (18.5)$$

Similarly, the 2nd-order ordinary differential equation derived from the chemical species material balance can be approximated as

$$\frac{1}{P_e} \mathbf{B}_j \mathbf{x} - \mathbf{A}_j \mathbf{x} - Kx_j = 0 \quad j = 2, \dots, M-1 \quad (18.6)$$

and the outlet boundary condition by

$$\mathbf{A}_M \mathbf{x} = 0. \quad (18.7)$$

The set of equations (18.5-18.7) define a set of linear equations that can be solved directly for the steady-state conversion values at the collocation points.

### 18.6.3 Collocation solution error analysis

One method to assess the error generated by the collocation discretization approximation is to compute the residual function of the collocation solution on a finer-scale quadrature grid. For the collocation procedure above, let  $x^M(z)$  be the polynomial function representing the  $M$ -point collocation solution. If  $\bar{\mathbf{z}}$  is the  $N$ -point finer quadrature grid (where  $N > M$ ) and  $\bar{\mathbf{A}}$  and  $\bar{\mathbf{B}}$  are the corresponding discrete differentiation operations, the residual function on this finer grid is computed simply as

$$\mathbf{R} = \frac{1}{P_e} \bar{\mathbf{B}} \bar{\mathbf{x}} - \bar{\mathbf{A}} \bar{\mathbf{x}} - K \bar{\mathbf{x}}$$

where  $\bar{\mathbf{x}}$  is the  $M$ -point solution interpolated to the finer quadrature grid. This multiple-grid collocation method is discussed in further detail in [2].



## 18.7 Object classes for quadrature

### 18.7.1 quadgrid: quadrature grid class

When solving a problem using quadrature-based projection methods, one must recompute the differentiation and quadrature arrays if the number of quadrature points is changed. The fixed relationship between the quadrature points, differentiation and quadrature weight arrays, and coordinate axis names leads naturally to encapsulating these data into a single object; this motivated developing the quadgrid object class. Objects of this class contain a physical-space grid of quadrature points and the above-mentioned arrays. No methods were created to modify the data fields of quadgrid objects once they are constructed; a new quadgrid object must be created if any changes in geometry or grid size are needed. Methods of this class include accessor (`get.m`), grid visualization (`plot.m`), and constructor methods. A quadgrid object is aggregated into objects of every other class defined in this chapter.

<b>quadgrid</b>
geom : cell array of strings qp : cell array of doubles w : cell array of doubles Q : cell array of doubles Qinv : cell array of doubles name : cell array of strings d : cell array of doubles dd : cell array of doubles d2 : cell array of doubles dd2 : cell array of doubles
colmat(A) quadgrid(geom,ndisc,name,axlim,ptype) compare(A,B) display(A) get(A,field,coname) mask(A,shape,params) mtimes(A,B) normal(A,x) plot(A) qg2sf(A,coname) quadrotshift(A,coname,shift) size(A)

### 18.7.2 Computational example: creating objects of quadgrid class

On their own, objects of quadgrid class have very little utility; however, it is instructive to consider how one constructs basic quadrature grids. Consider, for example, setting up a 10 point quadrature grid in coordinate  $r$  with maximum and minimum values  $r \in [0, 0.5]$  with cylindrical symmetry:

```
>> R = quadgrid('cyln',10,'r',[0 0.5])
1st element of quadgrid object array "R" of size:
    10
coordinate name(s) [limits] (geom):
    r : [ 0 , 0.5 ] (cyln)
qp:
    [10x1 double]
```

Now we extract the actual locations of the quadrature points using the `quadgrid/get.m` method:

```
>> Rqp = get(R,'qp','r')
Rqp =
    0
    0.0048
    0.0421
    0.1111
    0.2012
    0.2988
    0.3889
    0.4579
    0.4952
    0.5000
```

The quadrature point location double arrays are stored inside a cell array to allow for higher-dimensional quadrature grids; if X, Y, and Z are 1-dimensional quadrature grid objects, 2- and 3-dimensional quadrature grids simply are constructed using the overloaded `*` (`mtimes.m`) method:

```
>> T = quadgrid('peri',31,'t',[0 2*pi]);
>> Z = quadgrid('slab',15,'z',[0 1]);
>> subplot(2,2,1), plot(R)
>> subplot(2,2,2), plot(R*Z)
>> subplot(2,2,3), plot(R*T)
>> subplot(2,2,4), plot(R*Z*T)
```

The plots produced by this script are shown in Fig. 18.2.

### 18.7.3 scalarfield: scalar field class

A `scalarfield` object represents a scalar function value defined on a quadrature grid; a `scalarfield` object has data fields consisting of the function value at the quadrature points and the corresponding `quadgrid` (or `relquad`) object itself. `scalarfield` objects frequently are used to represent distributed state variables and residual functions evaluated on the quadrature grid in MWR applications.

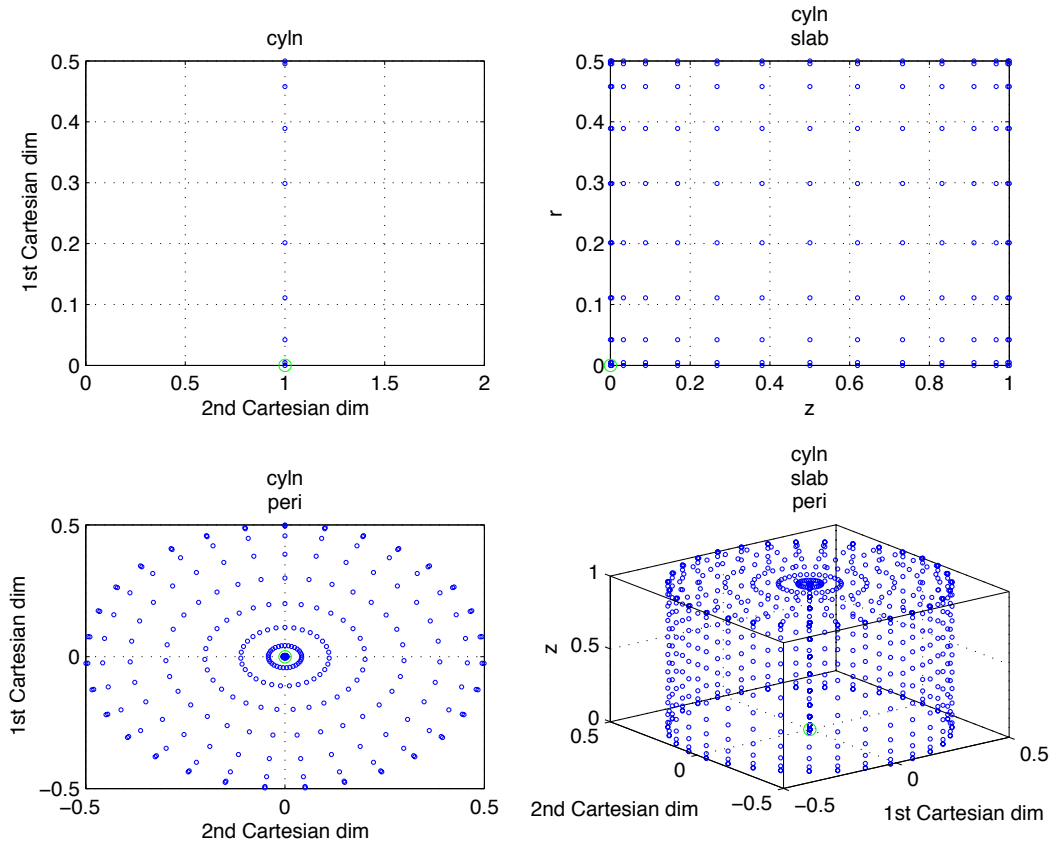


Figure 18.2: Plots indicating the locations of quadrature points for various geometries.

scalarfield	
pd : quadgrid	
val : double array	
abs(A)	mtimes(A,B)
bpsf(A,n,dir)	mwrsgs(phi,errtol)
cos(A)	normalize(A)
display(A)	orth(A,srmin)
exp(A)	plot(A,...,lwidth,marker)
expand(A,pdNew)	plus(A,B)
get(A,field)	rdivide(A,B)
getbval(A,coordname,minmax)	rq2qg(A)
getval(A,i,j,k,l)	scalarfield(pd,val)
interp(A,x,y)	setbval(A,B,coordname,minmax)
linterp(A,codir,locat)	setval(A,newval,i,j,k,l)
mean(A,coname)	sin(A)
minus(A,B)	substitute(A,B)
mpower(A,n)	svd(A,errtol)
mrdivide(A,B)	wip(A,B)

## 18.7.4 Computational example: creating objects of scalarfield class

```

>> X = quadgrid('slab',20,'x',[0 1]);
>> Y = quadgrid('slab',30,'y',[0 3]);
>> sx = qg2sf(X); % create scalarfield equivalent to f(x) = x
>> sy = qg2sf(Y);
>> f = cos(2*pi*sx);
>> g = sin(2*pi*sy);
>> f = expand(f,X*Y); % expand f and g to 2-dim and plot
>> g = expand(g,X*Y);
>> subplot(2,2,1), plot(f)
>> subplot(2,2,2), plot(g)
>> subplot(2,1,2), plot(f*g)
>> wip(f,g) % demonstrate orthogonality of f and g
ans =
-1.1471e-18

```

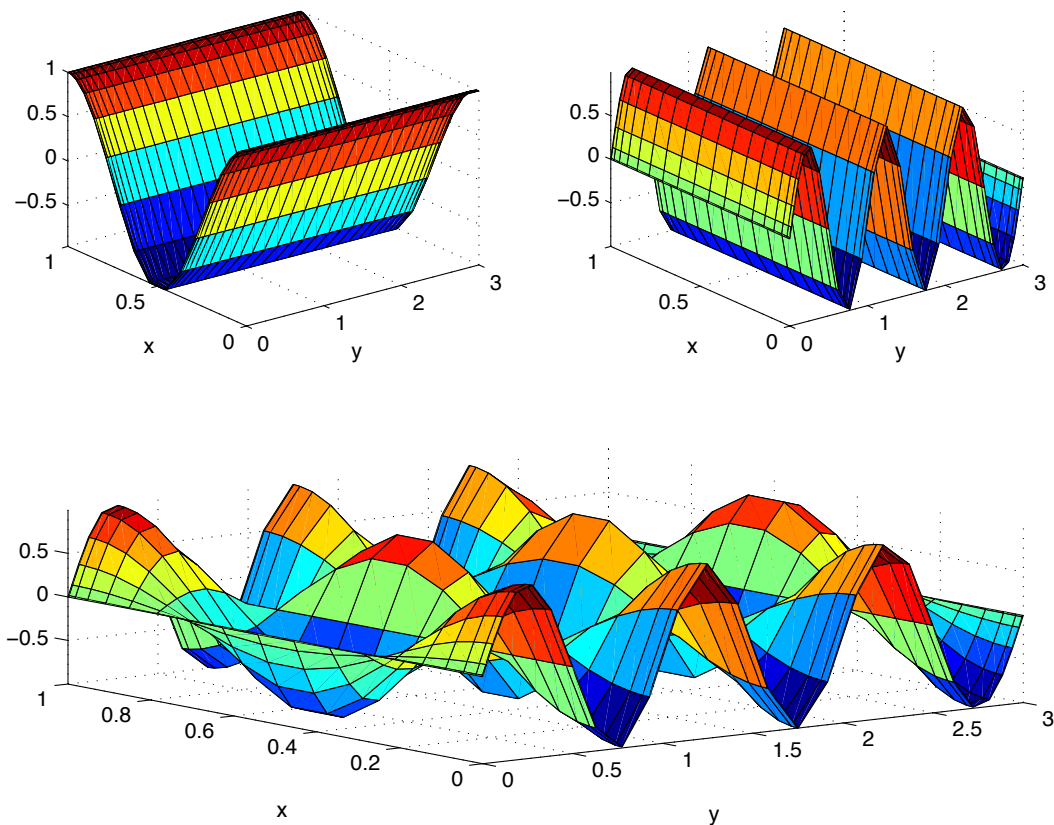


Figure 18.3: Creation ( $f$ , top left and  $g$ , top right) and multiplication of two (orthogonal) scalarfield objects  $f$  and  $g$ .

### 18.7.5 Computational example: using the linearoperator class

To demonstrate some basic properties and operations of the linearoperator class by continuing the previous computational example; we know that with  $f = \cos 2\pi x$

$$\frac{d}{dx} \cos 2\pi x = -2\pi \sin 2\pi x$$

and so

```
>> Dx = linearoperator(X,'d','x');
>> df = Dx*f;
>> subplot(2,2,1), plot(df)
>> ddf = Dx*df/(-4*pi^2); % will return original f = cos(2*pi*x)
>> subplot(2,2,2), plot(ddf)
>> dfg = Dx*(f*g);
>> subplot(2,1,2), plot(dfg)
```

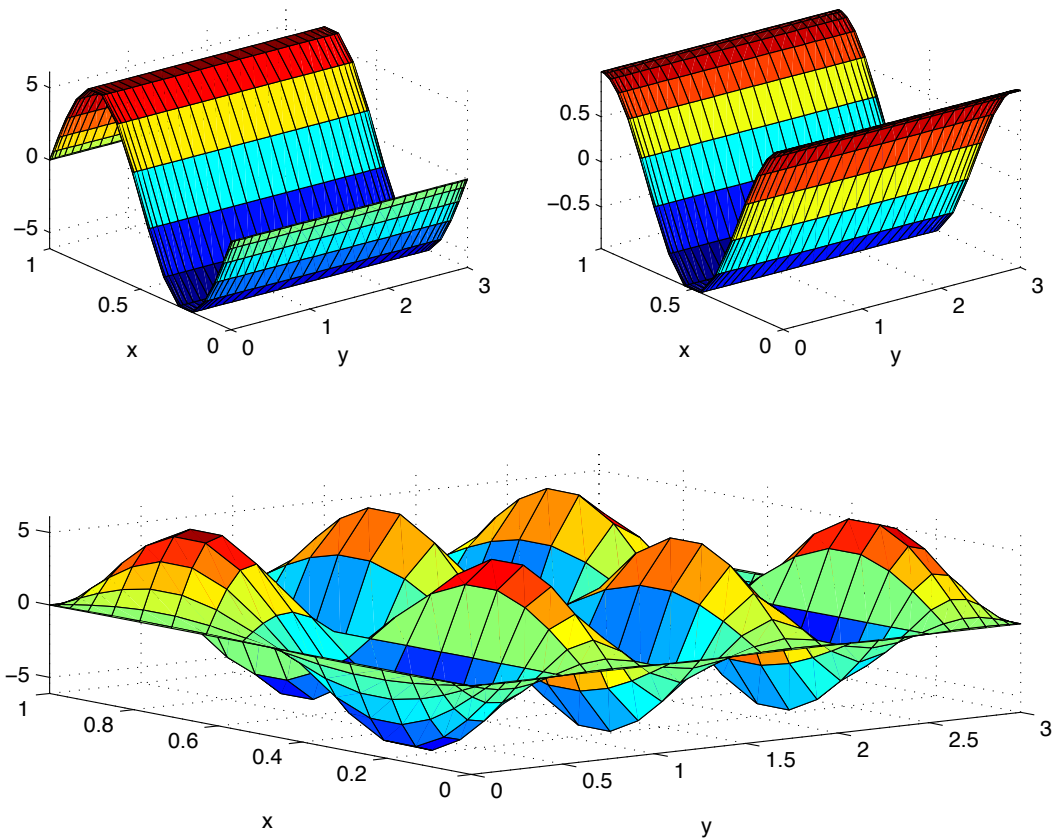


Figure 18.4:  $2\pi \sin 2\pi x$  computed by differentiating  $f$  with respect to  $x$  (top left); the original  $f$  returned by twice differentiation with respect to  $x$  (top right) and multiplication of two (orthogonal) scalarfield objects followed by differentiation with respect to  $x$  (bottom).

## Chapter 19

# Case Study: Tubular reactor boundary-value problem

Consider a system where a reacting species travels down a long tube (of length  $L$  m), transported by a combination of diffusion and convection due to the bulk fluid velocity  $v$  (in m/sec). Assuming the reactant concentration  $C$  (in moles/m<sup>3</sup>) is a function of only the axial position and time, we can write the reactant material balance over a differential element as

$$A(J|_z - J|_{z+\Delta z}) = A\Delta z R_c$$

with the reactant flux  $J$  (in mole/m<sup>2</sup> sec) written as

$$J = -D \frac{dC}{dz} + vC$$

and the reactant consumption rate per unit volume (moles/(sec m<sup>3</sup>)) term as proportional only to reactant concentration

$$R_c = k_c C.$$

This model is based on the assumptions of a homogeneous, first-order, isothermal reaction (the rate constant  $k_c$  has units sec<sup>-1</sup>). By taking  $\Delta z \rightarrow 0$  and combining terms, we find

$$\begin{aligned} 0 &= \frac{d}{dz}(-J) - k_c C \\ &= D \frac{d^2 C}{dz^2} - v \frac{dC}{dz} - k_c C \end{aligned}$$

The boundary conditions for both ends of the reactor are derived from the continuity of reactant flux across the boundaries. At the reactor inlet  $z = 0$ , the total flux must equal the reactant feed rate, so

$$\begin{aligned} vC_{in} &= J|_{z=0} \\ vC_{in} &= -D \frac{dC}{dz} \Big|_{z=0} + vC|_{z=0} \end{aligned}$$

so

$$-D \frac{dC}{dz} = v(C_{in} - C) \quad \text{at } z = 0.$$

Defining the flux as constant at the reactor outlet,

$$D \frac{dC}{dz} \Big|_{z=L-\epsilon} + vC|_{z=L-\epsilon} = D \frac{dC}{dz} \Big|_{z=L+\epsilon} + vC|_{z=L+\epsilon}$$

Without any other information on reactant transport beyond the reactor exit, it is impossible to simplify this equation. Under the assumption that there are no concentration gradients for  $z > 1$ ,

$$\frac{dC}{dz} = 0 \quad \text{at} \quad z = 1$$

because physically unrealistic concentration differences result when the derivative is not zero.

The modeling equations can be put in dimensionless form with the following definitions

$$\xi = \frac{z}{L} \quad x = \frac{C}{C_{in}} \quad P_e = \frac{Lv}{D} \quad K = \frac{k_c L}{v}$$

to find the homogeneous ODE model

$$0 = \frac{1}{P_e} \frac{d^2 x}{d\xi^2} - \frac{dx}{d\xi} - Kx$$

subject to the nonhomogeneous boundary conditions

$$\begin{aligned} \frac{1}{P_e} \frac{dx}{d\xi} &= x - 1 \quad \text{at} \quad \xi = 0 \\ \frac{dx}{d\xi} &= 0 \quad \xi = 1 \end{aligned}$$

If we split the second-order differential equation into the following two first-order ODEs

$$\begin{aligned} \frac{dx}{d\xi} &= y \\ \frac{dy}{d\xi} &= P_e(y + Kx) \end{aligned}$$

we can write the problem in matrix notation

$$\frac{d}{d\xi} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ KP_e & P_e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

subject to boundary conditions

$$\begin{aligned} y &= P_e(x - 1) \quad \text{at} \quad \xi = 0 \\ y &= 0 \quad \xi = 1 \end{aligned}$$

## 19.1 Exact solution

Choosing representative parameter values  $P_e = 4$  and  $K = 5/4$  gives the homogeneous set of ordinary differential equations

$$\frac{d}{d\xi} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 5 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{A} \begin{bmatrix} x \\ y \end{bmatrix}$$

subject to boundary conditions

$$y = 4(x - 1) \text{ at } \xi = 0 \quad \text{and} \quad y = 0 \text{ at } \xi = 1. \quad (19.1)$$

The eigenvalues of coefficient matrix  $\mathbf{A}$  can be computed to find  $\lambda_{1,2} = 5, -1$  along with their associated eigenvectors

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad \mathbf{u}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

This means the general solution takes the form

$$\begin{bmatrix} x \\ y \end{bmatrix} = q_1 e^{5\xi} \begin{bmatrix} 1 \\ 5 \end{bmatrix} + q_2 e^{-1\xi} \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

To determine  $\mathbf{q}$ , we solve for the coefficients using the boundary conditions. At  $\xi = 0$  and using (19.1) we find  $5q_1 - q_2 = 4q_1 + 4q_2 - 4$  and at  $\xi = 1$  we find  $5q_1 e^5 - q_2 e^{-1} = 0$ ; putting this in matrix form

$$\begin{bmatrix} 1 & -5 \\ 5e^5 & -e^{-1} \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} -4 \\ 0 \end{bmatrix}.$$

It is interesting to solve this system by hand using Gaussian Elimination because it shows the importance of pivoting. Using the partial pivoting rule and exchanging the two rows, performing the single elimination step, and neglecting small terms gives

$$\begin{bmatrix} 5e^5 & -e^{-1} \\ 0 & -5 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0 \\ -4 \end{bmatrix}.$$

which gives

$$q_2 = \frac{4}{5} \quad q_1 = \frac{4}{25} e^{-6}$$

from back-substitution. We now compare this solution to that which is obtained without the pivoting operation; carrying out the elimination procedure and neglecting relatively small terms gives

$$\begin{bmatrix} 1 & -5 \\ 0 & 25e^5 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} -4 \\ 20e^5 \end{bmatrix}.$$

which gives

$$q_2 = \frac{4}{5} \quad q_1 = 0.$$

While this solution is numerically close to the solution obtained with pivoting, they are qualitatively different in that the second does not satisfy the outlet boundary condition.

We now write the exact solution

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{4e^{-6}}{25} e^{5\xi} \begin{bmatrix} 1 \\ 5 \end{bmatrix} + \frac{4}{5} e^{-1\xi} \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

The solution is plotted in Fig. 19.1.



### 19.1.1 BVP collocation solution

We compute the solution to the hot-wall reactor problem using a  $M = 6$  point collocation method resulting in  $M$  linear discretized equations. The computed solution at the collocation points is compared to the exact solution in Fig. 19.1

```
M = 6; Pe = 4; K = 5/4;
Z = quadgrid('slab',M,'z',[0 1]); % quadgrid object
Zf = quadgrid('slab',4*M,'z',[0 1]); % finer grid for exact solution

[z, A, B] = colmat(Z); % extract collocation array info

C = zeros(M);
b = zeros(M,1);

C(1,:) = A(1,+)/Pe; % inlet boundary condition
C(1,1) = C(1,1) - 1;
b(1,1) = -1;

C(2:M-1,:) = B(2:M-1,+)/Pe - A(2:M-1,); % interior points
C(2:M-1,2:M-1) = C(2:M-1,2:M-1) - K*eye(M-2);

C(M,:) = A(M,); % outlet boundary condition

x = C\b; % compute mole fraction at collocation points

Zfsf = qg2sf(Zf);
% exact solution
xExact = 4*exp(-6)*exp(5*Zfsf)/25 + 4*exp(-Zfsf)/5;
```

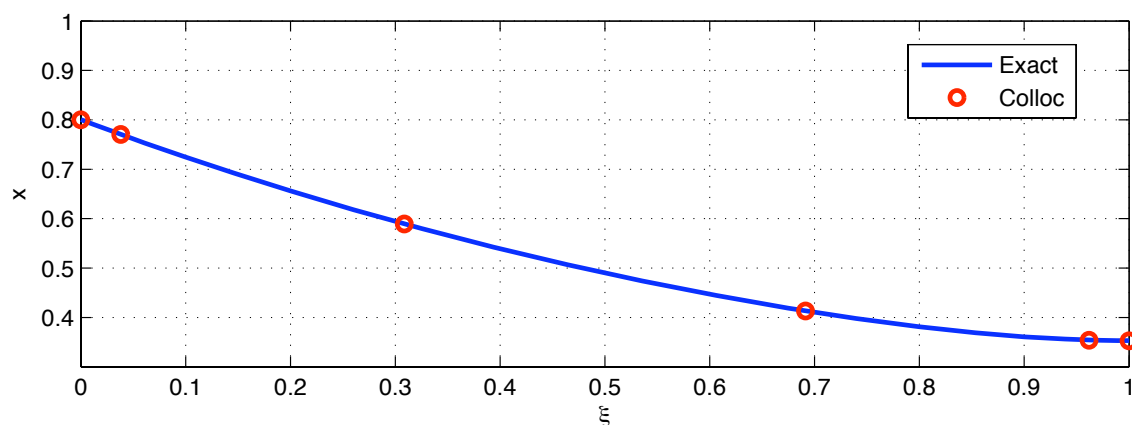


Figure 19.1: Exact solution (blue curve) compared to the  $M$ -point collocation solution; the latter is represented by the red circles.

### 19.1.2 Collocation solution error analysis

Finally, we can take a closer look at the errors generated by the collocation discretization procedure. As seen in Fig. 19.2, we re-plot the collocation and exact solutions in the uppermost plots. We can translate the collocation solution defined at the collocation points into a scalarfield object and then compute the residual at both the collocation points and in between by

```
xcolsf = scalarfield(Z,x);
residual = (1/Pe)*laplacian(xcolsf) - gradient(xcolsf) - K*x;
```

The significance of this operation is that because the scalarfield object represents the function value between the collocation points by the (unique) interpolating polynomial, we can observe that the polynomial representation of the residual between the collocation points is indeed non-zero, while it is zero at the interior collocation points (Fig. 19.2, center). Because the residual is computed with the differential equation, we should not expect it to vanish at the endpoints. Furthermore, direct comparison of the collocation solution to the exact solution reveals it is non-zero at the collocations points. This can be attributed to the polynomial representation of the collocation solution, compared to the exponential form of the true solution.

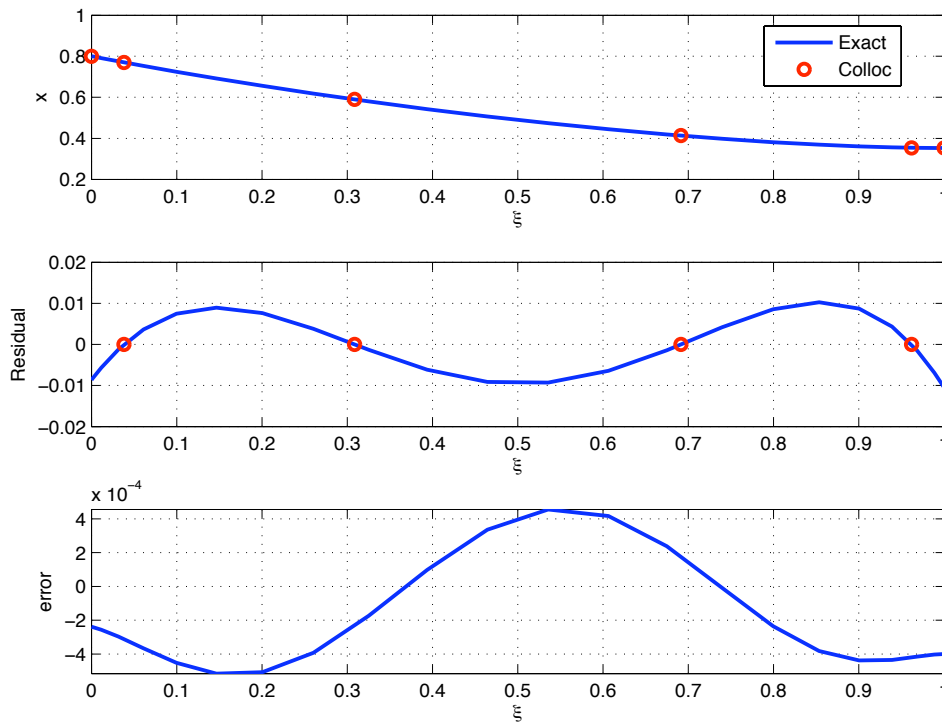


Figure 19.2: Exact solution (blue curve) compared to the  $M$ -point collocation solution (top); the latter is represented by the red circles. The residual of the function demonstrating the error inside  $\Omega$  (middle). Note how the residual function is zero at the interior collocation points (marked in red), but is not at the endpoints. Difference between exact and collocation solution over the entire physical domain (bottom).

## Chapter 20

# Case Study: One-dimensional transient heat transfer

Consider a material sandwiched between two rigid plates with temperatures  $T_A$  and  $T_B$  (as shown in Fig. 20.1).

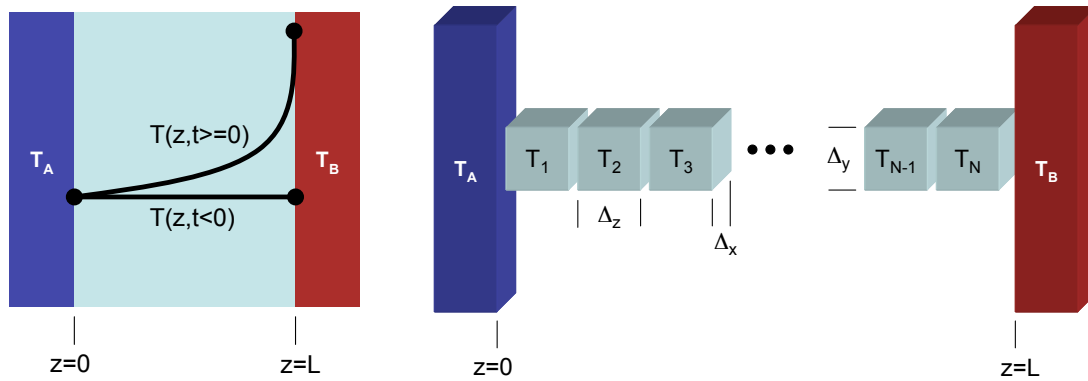


Figure 20.1: *Temperature profiles in a slab of material between plates of two different temperatures ( $T_A$  and  $T_B$ ) at  $t < 0$  and  $t \geq 0$  (left); finite volume discretization of the temperature profile (right).*

Initially, the fluid is at a constant temperature  $T(z, t < 0) = T_A$ , but precisely at  $t = 0$ , the temperature of wall B is raised to  $T_B(t \geq 0) > T_A$ . Intuition tells us that the temperature profile  $T(z, t)$  appears as the sample profile depicted in Fig. 20.1 for  $t \geq 0$ .

Our goal is to compute  $T(z, t)$  for  $t \geq 0$ , including the steady state corresponding to  $t \rightarrow \infty$ . We begin development of the solution procedure by breaking the fluid up into  $N$  equal volumes (see Fig. 20.1, right) where

$$\Delta_x = \Delta_y = \Delta_z = \frac{L}{N}.$$

Therefore the volume of each element is simply  $\Delta_x \Delta_y \Delta_z$ . Given the heat capacity of the material between the walls ( $C_p$  with units  $J/kg/K$ ), its density ( $\rho$  with units  $kg/m^3$ ), and assuming that each is

independent of temperature, we can write an energy balance for each element as

$$C_p \rho \Delta_x \Delta_y \Delta_z \frac{dT_n}{dt} = \Delta_x \Delta_y (Q_{n-1} + Q_n) \quad (20.1)$$

ignoring the elements corresponding to  $T_1$  and  $T_N$  for now, and where  $Q_{n-1}$  and  $Q_n$  correspond to the net thermal energy flux to element  $n$  from its neighbors to the left and right, respectively.

If  $k$  is the thermal conductivity of the material (with units  $J/m/s/K$ ), we can approximate Fourier's law of heat conduction by

$$Q_{n-1} = k \frac{T_{n-1} - T_n}{\Delta_z} \quad Q_n = k \frac{T_{n+1} - T_n}{\Delta_z} \quad (20.2)$$

## 20.1 Numerical solution

1. Start by taking the limit of  $\Delta_z \rightarrow 0$  (and recalling your very first calculus class) of (20.1), resulting in the energy flux at each boundary of the element  $n$ :

$$Q_{n-1} = -k \left. \frac{dT}{dz} \right|_{z+(n-1)\Delta_z} \quad Q_n = k \left. \frac{dT}{dz} \right|_{z+n\Delta_z} \quad (20.3)$$

Substituting (20.3) into (20.1) and again taking the limit of  $\Delta_z \rightarrow 0$  results in the partial differential equation and corresponding boundary and initial conditions:

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial z^2} \quad \text{subject to} \quad T(0) = T_A, T(L) = T_B, T(z, 0) = T_A, 0 < z < L \quad (20.4)$$

2. Given the boundary condition values  $T_A = 10^\circ C$  and  $T_B = 90^\circ C$ , compute the exact steady-state solution to the PDE derived in the previous question.

We integrate the steady state version of (20.4) twice to find

$$T(z) = az + b$$

where the constants of integration are determined from the boundary conditions resulting in

$$T(z) = 80z/L + 10.$$

3. Our system is such that the values of  $C_p$ ,  $\rho$ , and  $k$  result in

$$\frac{k}{\rho C_p} = 1 \frac{m^2}{s} \quad (20.5)$$

(nothing is lost in this assumption concerning the validity of our solution). With  $N = 3$ , write out the equations

$$\frac{dT_n}{dt} = \dots \quad n = 1, 2, 3$$

in their simplest possible form.

4. Write the three equations above in the following form

$$\frac{d}{dt} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

5. Create a MATLAB function of the form

```
[A,b] = makecoeff(N,Tleft,Tright)
```

That creates the **A** and **b** arrays given the number of elements  $N$  and the boundary temperature values. Report the results for  $N = 3$ .

6. Using the **A** and **b** arrays generated by the function above, numerically compute the steady state solutions for  $N = 3$  and  $N = 20$ ; compare your results to the exact solution by plotting the three on a single graph.
7. Write a MATLAB script that does the following

```
saveig = zeros(30);

for N = 1:30
    A = makecoeff(N,TA,TB);
    % now compute the eigenvalues of A
    % now sort them from smallest to largest
    % save the sorted eigenvalues in the saveig array
end
```

Now, plot the saved eigenvalues as a function of  $N$ ; what values do the eigenvalues converge to? Does the system appear to be stable or unstable?

8. For  $N = 30$ , plot the eigenvectors (versus  $z$ ) associated with the three smallest magnitude eigenvalues. What functions do the results look like?
9. Using `lodesolver.m`, compute a solution for  $N = 20$  and plot the temperature profiles as a function of  $z$  for  $t = 0, 0.1, 0.5, 1, 10$ s. What is your impression of the time scale(s) of this system?

## 20.2 Transport in more complicated geometries

Now consider the system shown in Fig. 20.2.

If (20.5) holds and  $\Delta_x = \Delta_y = \Delta_z = 1m$

1. Write out the equations

$$\frac{dT_n}{dt} = \dots \quad n = 1, 2, \dots, 10$$

in their simplest possible form.

2. Given the initial condition

$$\begin{aligned} T_n(t=0) &= 0 & n = 1, 3, 4, \dots, 10 \\ T_2(t=0) &= 1 \end{aligned}$$

compute solutions for  $0 \leq t \leq 5$ s; plot and describe the results.

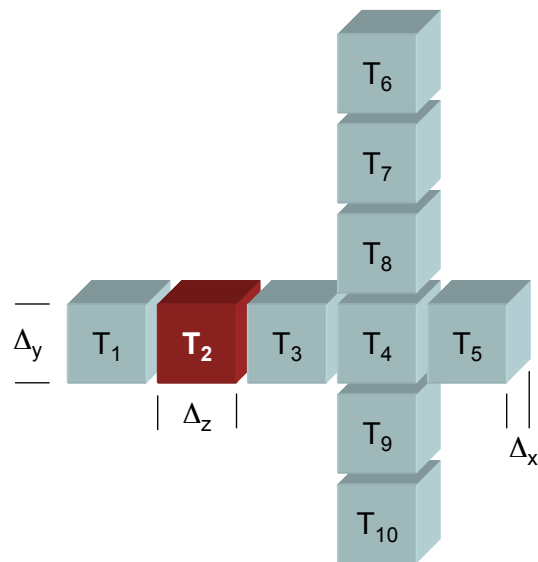


Figure 20.2: *Finite volume discretization of a geometrically more complicated system.*

# Bibliography

- [1] Adomaitis, R. A. Spectral filtering for improved performance of collocation discretization methods. *Comp. & Chem. Eng.*, **25** 1621-1632 (2001).
- [2] Adomaitis, R. A. and Y.-h. Lin, A technique for accurate collocation residual calculations, *Chem. Engng J.*, **71** 127-134 (1998).
- [3] Biegler, L. T., Solution of dynamic optimization problems by successive quadratic programming and orthogonal collocation, *Comput. & Chem. Engng* **8** 243-248 (1984).
- [4] Birnbaum, I. and L. Lapidus, Studies in approximation methods IV: Double orthogonal collocation with an infinite domain, *Chem. Engng Sci.* **33** 455-462 (1978).
- [5] Chang, H. -Y., R. A. Adomaitis, J. N. Kidder, Jr., and G. W. Rubloff, 'Influence of gas composition on wafer temperature in a tungsten chemical vapor deposition reactor: Experimental measurements, model development, and parameter estimation,' *J. Vac. Sci. Tech., B*, **19** 230-238 (2001).
- [6] The on-line complexity digest <http://www.comdig.com>
- [7] Cornish-Bowden, A. *Analysis of Enzyme Kinetic Data*, Oxford University Press (1995).
- [8] Cuthrell, J. E. and L. T. Biegler, On the optimization of differential-algebraic process systems, *AIChE J.* **33** 1257-1270 (1987).
- [9] Cuthrell, J. E. and L. T. Biegler, Simultaneous optimization and solution methods for batch reactor control profiles, *Comput. & Chem. Engng* **13** 49-62 (1988).
- [10] Doedel, E., AUTO: A program for the automatic bifurcation analysis of autonomous systems, *Congressus Numerantium* **30** 265-284 (1981).
- [11] Doherty, M. F. and M. F. Malone *Conceptual Design of Distillation Systems* McGraw-Hill Chemical Engineering Series (2001).
- [12] Engels, H., 1980, *Numerical Quadrature and Cubature* Academic Press, London.
- [13] Gottlieb, D. and S. A. Orszag, *Numerical Analysis of Spectral Methods* SIAM CBMS-NSF Regional Conf. Series in Appl. Math. Vol. 26 (1977).
- [14] Gottlieb, D. and C. -W. Shu, On the Gibbs phenomenon and its resolution, *SIAM Rev.* **39** 644-668 (1997).
- [15] Guckenheimer, J. and P. Holmes, 1983, *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, Applied Mathematical Sciences, vol. 42, Springer-Verlag, New York.

- [16] Hildebrand, F. B. 1965, *Methods of Applied Mathematics* Prentice-Hall, New Jersey.
- [17] Jahne, B., *Digital Image Processing. Concepts, Algorithms, and Scientific Applications, 2nd Ed.* Springer-Verlag, Berlin (1993).
- [18] Kelly, C. T. *Solving Nonlinear Equations with Newton's Method* SIAM Press (2003).
- [19] Kim, I.-W, M. J. Liebman, and T. F. Edgar, A sequential error-in-variables method for nonlinear dynamic systems, *Comput. & Chem. Engng* **15** 663-670 (1991).
- [20] Logsdon, J. S. and L. T. Biegler, A relaxed reduced space SQP strategy for dynamic optimization problems, *Comput. & Chem. Engng* **17** 367-373 (1993).
- [21] Middleman, S. and A. K. Hochberg, *Process Engineering Analysis in Semiconductor Device Fabrication* McGraw-Hill, Inc (1993).
- [22] Myers, R. H. and D. C. Montgomery, *Response Surface Methodology; Process and Product Optimization Using Designed Experiments* Wiley Series in Probability and Statistics, J. Wiley & Sons, Inc. (2002).
- [23] Ogunnaike, B. A. and W. H. Ray, 1994, *Process Dynamics, Modeling, and Control* Oxford University Press, New York.
- [24] Smith, J. M. 1981, *Chemical Engineering Kinetics* McGraw-Hill, Inc., pp. 249-250.
- [25] Smith, J. M., H. C. Van Ness, M. M. Abbott 2005, *Introduction to Chemical Engineering Thermodynamics, 7th edition*, McGraw-Hill's Chemical Engineering Series, New York.
- [26] Strang, G. *Introduction to Applied Mathematics* Wellesley-Cambridge Press (1986).
- [27] Stroud, A. H. 1974, *Numerical Quadrature and Solution of Ordinary Differential Equations* Springer-Verlag, New York.
- [28] Vandevein, H., Family of spectral filters for discontinuous problems, *J. Sci. Comput.* **6** 159-192 (1991).
- [29] Villadsen, J. and J. P. Sørensen, Solution of partial differential equations by a double collocation method, *Chem. Engng Sci.* **24** 1337-1349 (1969).