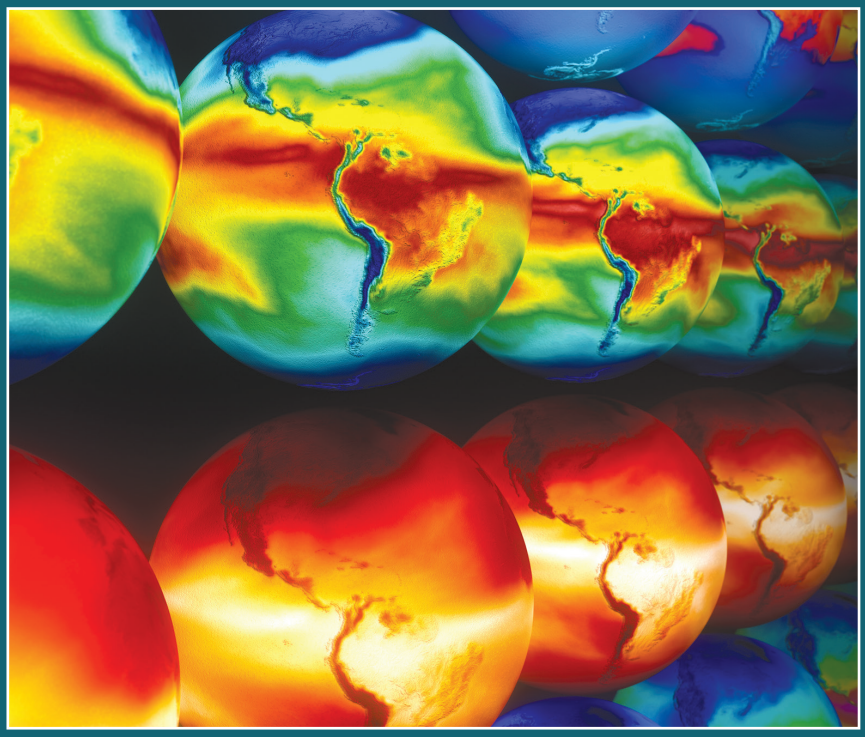
Climate Modeling for Scientists and Engineers

MATLAB Exercises



John B. Drake

siam
Mathematical Modeling and Computation

10 9 8 7 6 5 4 3 2 1

Cover art: Visualization of time dependent fields of the community climate system model. Courtesy of Jamison Daniel. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC05-00OR22725.

Figure 2.4 reprinted with permission from IPCC.

Library of Congress Cataloging-in-Publication Data

Drake, John B. (John Bryant), author.
Climate modeling for scientists and engineers / John B. Drake, University of Tennessee, Knoxville, Tennessee.
pages cm
Includes bibliographical references and index.
ISBN 978-1-611973-53-2
1. Climatology–Data processing. 2. Climatology–Mathematical models. I. Title.
QC874.D73 2014
551.501'1–dc23

2014016721

MATLAB Exercises
Contents

Chapter 1

MATLAB Programming

Computer programming may appear to be a mysterious, esoteric ritual of communicating with the giant electronic brains of science fiction. At least in the literary imagination, programmers seldom emerge into the light of day, keeping odd hours, subsisting on pizza and caffeine, and exhibiting peculiar social ties. Perhaps this image accounts for the dearth of female undergraduates that choose Computer Science as a college major and the lack of graduates with programming credentials.

From my perspective, as a computational scientist,[1] the computer itself is secondary; it is just a tool that we use in doing science. What matters is solving problems. How to do this, the techniques, methods, and insights, constitute the toolbox of the computational scientist. The research effort, in collaboration with other scientists, is often the assemblage of a computer program, or a code. Code is inherently communal, and, thus, it should be readable and read by other people. Code is not just for the computer to ingest but, more fundamentally, a community science document for other researchers to digest. The digital age and science in the digital age have not yet come to grips with the implications of computational science. Considerable misunderstanding and resistance to recognizing the role of modeling and simulation continues in many scientific disciplines. Much of the practice of software engineering, digital object tagging, and open information is still evolving, with programmers creating much of what we experience today.

I was often asked, when I was directing research programs, why we couldn't just hire a bunch of programmers to do climate modeling. Typically, the qualifications for a computational climate scientist involve an advanced degree in atmospheric science, physics, chemistry, or mathematics in addition to computer expertise. My experience has been that it is easier to teach someone how to program than it is for them to absorb a new area of science. Some of the data abstractions, as well as language abstractions, that computer science teaches are important for advanced programming practice, but the elegance and simplicity of a program is first and foremost an expression of the programmer's ability to break the tasks into steps and to organize these in a logical sequence. The teaching of programming should not be complicated with too much abstraction or syntax, and I maintain that the best way to learn the tool is to start to use it.

[1] Computer scientists are not necessarily computational scientists. That is, the science of computing is different than doing science with computers.

I had the good fortune in my own training to work and study with some of the pioneers in computing.[2] Programming is something you learn by doing, and in the spirit of these computational pioneers who tried to solve problems, I will present examples quickly leading to substantial and useful problems.

1.1 ▪ What is Programming?

Computer circuits are limited in the operations they can perform. These operations have remained nearly constant even as the number of transistors and switchs on a chip has increased exponentially over the past several decades.[3] What computers do at the basic machine instruction level is move the contents of a register to memory (store), retrieve contents from memory, and add or multiply two binary numbers that are in different registers. By grouping various collections of these *commands* or *instructions* together, more sophisticated tasks are performed, such as division, square roots, printing, graphics, and solving linear equations. The sequence of these instructions also requires some way to make decisions—to branch in the sequence of instructions based on some condition. The simplest condition is a test of whether a number is positive or negative.

Here, in Figure 1.1, is a simple program that prints the phrase "Hello World" depending on the value of the sign of the variable *awake*. The first three lines start with a % sign

```
% Comment: First program that prints Hello World
% Written by me
% Date:  today
   awake = 1;
   string = 'Hello World';
   if awake > 0
     display(string);
   end
```

Figure 1.1. *This first program simply prints the phrase "Hello World." By entering it and running it you will have figured out the basics of editing a program, some fundamental programming syntax, and how to execute or run a program. Though the program is trivial, these other things are not.*

indicating that what follows is a comment. The computer skips over comments, which are strictly for human consumption. Comments can appear anywhere in the program and include all the characters on the line after the %.

Let's explain the details of the rest of the program. Each line (or group of lines) is an instruction for the computer to do something. An instruction in MATLAB typically

[2]Michael Kennedy wrote a popular text introducing programming with ten simple statements. He was my supervisor in one of the first computer graphics labs on an academic campus. The University of Kentucky College of Architecture had serial number 35 of the Digital Equipment Corporation GT-40 computer in 1972. The graphics workstation was a PDP-11 hooked to a cathode ray tube as the display device. We programmed assembly language routines by toggling in octal codes. Programming became much easier as compilers and peripheral interfaces were developed for the machine. As a student I got to meet Rear Admiral Grace Hopper, who was the first to conceive of machine independent languages and develop a compiler. Henry Thatcher was the first to teach me numerical analysis, the art of performing mathematical calculations in finite arithmetic. Others to whom I owe a debt for my training in computational science include John Rice and Walter Gautchi at Purdue as well as Rolf Jeltch from his early days of teaching at the University of Kentucky.

[3]Gordon Moore conjectured in 1965 that the number of transistors on a printed circuit board would double about every two years. This became known as Moore's Law. As the size of the circuits shrank and clock speeds got faster, the speed of computation also increased proportionally. With all the technological improvements, a doubling of computing capacity has occured every 18 months for the last half century.

ends with a semicolon. The first executable instruction puts the numeric value 1 into a memory location with the variable name *awake*. The second line assigns the variable named *string* with the character string value 'Hello World'. A single quote is used to indicate a character string. The equal sign is really an assignment operator, an instruction that assigns a value to the variable on the left side of the equal sign. (Note that "1 = awake;" does not work. It is a syntax error.)

The first two lines illustrate two different *data types* in MATLAB. The first is a number and the second a character string. The variables on the left-hand side of these take on the data types of the constants that are assigned to them. MATLAB uses *implicit typing* of variables in this way. Many languages, for example, FORTRAN or C, encourage explicit typing for clarity. MATLAB has a unique feature in that variables are treated as matrices. The MATLAB language was developed with linear algebra in mind. Even though it makes no sense in this program, *awake* is a 1×1 matrix.

The third instruction is an *if statement*, a conditional or control statement. If the expression following the *if* is true, then the next statement is executed. If the expression is not true, then the program control drops through without executing the *display*. This program control gives the sense of execution following the sequence of commands in the program with different sequences depending on the values of the variables in the program. Conditionals offer much of the power and flexibility for problem solving in programming, and in this way the *input* to the program may change the processing and *output* of the program. Clearly, the input to this program is minimal—the setting of *awake* to one. And the output is also minimal—the printing of the phrase "Hello World." A computer program is nothing more than a sequential list of instructions.

As a programmer, you may use many short cuts. For example, in MATLAB any type of variable will display itself simply by invoking it on a line. So we could have used *string* inside the *if*. We could also save space by putting more than one instruction on a line. So the first two instructions, each ending with a semicolon, could be combined on a single line.

Exercise 1.1.1. *As a first programming exercise,* Hello World *appeared in the classic book* The C Programming Language *by Kernigan and Ritchie. Enter the program into a file named "hello.m". Using the edit screen within MATLAB, run the program. Modify the code, for example, changing the value of* awake*, to become familiar with the way to edit and use MATLAB. Also note the "Help" menu within MATLAB that you may use as a reference on syntax, commands, and programming.*

1.2 ▪ What is MATLAB?

The MathWorks company[4] developed MATLAB for exploration of numerical methods and to make computers accessible to students for linear algebra and complex numerical calculations. It is an *interpretive computer language* and *program development environment*, meaning that each instruction in a program is first interpreted and then executed before the next instruction is processed. It is as if each line were entered in the command window. A *compiled language* translates all the instructions to machine code before starting the execution. The MATLAB system offers a rich set of commands and toolboxes for programming and scientific computation. The extent of what is available is quite impressive, especially as personal computers have become faster and more powerful, allowing the feasible number of calculations to expand.

[4] See www.mathworks.com.

The program development environment of MATLAB provides everything you need to create, debug, and use your program. In the command window, you will notice an ability to compute interactively. We will not use the command window much. Instead we will edit MATLAB program files (files that end in the suffix .m), creating programs that perform more complex calculations. MATLAB also has the ability to add new user defined data objects to the MATLAB structure, creating what appear to be extensions to the language. This is extremely useful and is supported by the *object oriented* programming language.

Several toolboxes offer added capabilities for climate modeling. In partiuclar, the mapping toolbox is important since we are interested in displaying information about the earth with coastlines clearly delineated. The various map projections are built into the functionality of the toolbox, relieving us of the need to develop all the formulas for display of geographic information.

The *Hello World* code introduces input and output steps with no computation between. For analysis of climate data the steps quickly generalize to reading data and displaying graphical results. As a more useful illustration of computation, the next program looks at the relationship between temperature and CO_2 data from the Vostok ice cores.

Exercise 1.2.1. *The Vostok ice core data covers what time period? How can you get the data? After obtaining the data plot CO_2 versus ΔT, and dust versus ΔT. According to the data what should the present ΔT be, based on a current CO_2 concentration of 400 ppm? What about for 450 ppm? How would you characterize the "error bound" on your projection of global warming?*

The temperature data for the Vostok ice cores is contained in a file called *vostok.temp* and appears as two columns of numbers, the time before present and the temperature difference from the current Vostok surface average of 55^oC. The second file with the same structure, though with different times than contained in the temperature data, contains the CO_2 data and is called *vostok.co2*. The program must first read these files and store the data in appropriate arrays for plotting. The program that reads the files and creates a plot of the data is shown in Figure 1.2

```
% Open the data files and read columns into arrays.
tempID = fopen('vostok.temp');
co2ID = fopen('vostok.co2');
Temp = fscanf(tempID,'%f %f',[2 inf]);
Temp= Temp';
plot(Temp(:,1),Temp(:,2))
pause
CO2  = fscanf(co2ID, '%g %g',[2 inf]);
CO2= CO2';
plot(CO2(:,1),CO2(:,2))
pause
fclose(tempID);
fclose(co2ID);
```

Figure 1.2. *Open two files and read two columns of numbers, time, and value. To see that the data have been captured, some simple plots of temperature and CO_2 levels are created.*

To create the temperature versus CO_2 plot requires that values be interpolated to the

same times. The interpolation is handled in the next fragment of code, Figure 1.3. Finally,

```
% Plot Temp vs CO2
maxtime1 = max(Temp(:,1));
maxtime2= max(CO2(:,1));
maxtime = min(maxtime1,maxtime2);
mintime1= Temp(1,1);
mintime2= CO2(1,1);
mintime = max(mintime1,mintime2);
% Interpolate linearly
ntimes = 100;
Tunif = zeros(ntimes);
Cunif = zeros(ntimes);
dt = (maxtime - mintime)/(ntimes-1)
time = mintime;
for n = 1:ntimes
   Tunif(n) = interp1q(Temp(:,1),Temp(:,2),time);
   Cunif(n)  = interp1q(CO2(:,1),CO2(:,2),time);
  time = time + dt;
end
```

Figure 1.3. *MATLAB creates the arrays* Tunif *and* Cunif *by first storing zeros in them. This saves time in the* for*-loop since the elements do not have to be appended one at a time. The MATLAB interpolation routine* interp1q *performs a linear interpolation between the data values.*

the two arrays of interpolated temperature and CO_2 data are used to create a final plot; see Figure 1.4. The plot produced by this program is shown in Figure 1.5.

```
% Plot the correlation TvsC
plot(Tunif,Cunif,'ok',...
    'LineWidth',2,...
    'MarkerFaceColor','k',...
    'MarkerSize',8)
axis([-10 5 100 400])
xlabel('Temperature Difference (C)',...
    'fontsize',12,'fontweight','n','color','k')
ylabel('Atmospheric CO2 (ppm)',...
    'fontsize',12,'fontweight','n','color','k')
```

Figure 1.4. *Specifying the fonts and axis labels is done by assigning various values to the plot properties. This is a black and white plot according to the color 'k'.*

Keeping with the theme of useful input and output, the following sample code, Figure 1.6, takes advantage of the NetCDF capabilities within MATLAB to read monthly averaged Reanalysis Data from NCEP[5] and display this data on a map. The program it-

[5]The National Center for Environmental Prediction (NCEP) is the weather service for the United States. NCEP provides globally gridded data sets of the monthly average values for weather variables such as pressure, temperature, and precipitation as well as three dimensional fields useful for modeling and other scientific studies. These are called *Reanalysis Data* because they have been interpolated from actual observations under the constraints of a physical model. The other comparable dataset of the past climate system is the ERA-40 Reanalysis

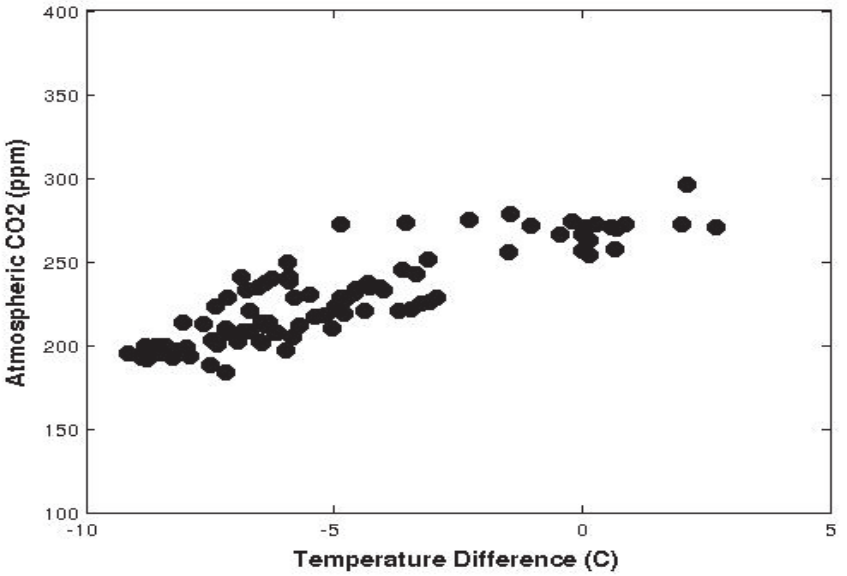


Figure 1.5. *Plot of interpolated Vostok temperature versus CO_2 data from the present time to 160,000 years ago.*

self is an example of simple input and output with no computation in between. But the ability to look at weather and climate data is a prerequisite to modeling. How else could we know what we are trying to model, or how good a job we are doing at reproducing the observed system?

A word about NetCDF, its origin, and role in climate research. The ability of a computer program to read data generated on a different computer using a different program is nothing short of remarkable. We have become used to interoperability as a standard on the Internet with formats like HTML for web pages and XML, forgetting the years of community effort to define the standards. The NetCDF acronym stands for "Network Common Data Form" and consists in a set of interfaces for array-oriented data access. It has been implemented in libraries for C, Fortran, C++, Java, and other languages.[6] This self-describing, machine independent format was developed to support the National Center for Atmospheric Research in Boulder and the broader goal of sharing scientific data across research programs sponsored by the National Science Foundation. Other institutions and agencies have developed similar formats such as GRIB and HDF.

Exercise 1.2.2. *Use the NCEP Reanalysis data for monthly mean temperatures (see Figure 1.8) to compute global surface air temperatures for different months and annually. The changes in the global temperature are of what magnitude? Compute and display the sea level pressure (slp) anomalies with the difference being computed from the average of the 1960–1980 period. What are the monthly standard deviations?*

Data, but we will not use that here.

[6] UNIDATA (http://www.unidata.ucar.edu) supports and makes these available.

```
function NCEPcdfPlay(fp, field)
%Usage: NCEPcdfPlay('filename', 'field name');
% Cracks open the NCEP Reanalysis file in NetCDF format
% MATLAB netcdf intrinsics are used to extract fields
% and the Mapping Toolbox is used to produce a visual of the field
%Inputs example fp='pres.mon.mean.nc'; field='pres';
%  ......... or fp='air.mon.mean.nc'; field='air';
%  ......... or fp='rhum.mon.mean.nc'; field='rhum';
%  ......... or fp='slp.mon.mean.nc'; field='slp';
%
ncid = netcdf.open(fp,'NC_NOWRITE');
%Get the info from the file
[numdims, numvars, numglobalatts, unlimdimID] = netcdf.inq(ncid)
%Get the names of the variables.
for i = 0:numvars-1
  [varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,i)
end
%  Get the lat and lon arrays
varid = netcdf.inqVarID(ncid,'lat');
lat = netcdf.getVar(ncid,varid);
varid = netcdf.inqVarID(ncid,'lon');
lon = netcdf.getVar(ncid,varid);
% Create big Xlon and YLAT arrays
[Xlat,Ylon] = meshgrid(lat,lon);
%
% Get variable ID of the field variable requested, given its name.
varid = netcdf.inqVarID(ncid,field);
% Get the dimensions of the field array
[dimname, nlon] = netcdf.inqDim(ncid,0)
[dimname, nlat] = netcdf.inqDim(ncid,1)
[dimname, ntimes] = netcdf.inqDim(ncid,2)
% Get the value of the field variable, given its ID.
F = netcdf.getVar(ncid,varid);
```

Figure 1.6. *The NCEP monthly data is contained in NetCDF files that MATLAB can read using built in functions. Input to the function is the file name and the variable name to display. Typically, more than one month is included in each file so a movie is created of the* ntimes *snapshots.*

1.3 ▪ Object Oriented Programming

Computer languages have become increasingly powerful as new abstractions and commands have been added. The ability of the user, that is you, to define new data types is perhaps the biggest advance. The computer then works on and with these new *objects* in ways the programmer may specify. In MATLAB the intrinsic object is the matrix, and there are many methods and operations to manipulate and calculate with matrices. But what objects are natural for you to work with? The MATLAB Object Oriented Programming allows you to define your own data types and methods to work with those types.

```
%
% Display using the Mapping toolbox
clf;
axesm ('miller','Frame','on','Grid', 'on');
coast=load('coast');
geoshow(coast);
for it = 1:ntimes
   Fit = double(F(:,:,it));
   geoshow(Xlat,Ylon,Fit,'DisplayType','texturemap');
   hcb = colorbar('horiz');
   set(get(hcb,'Xlabel'),'String','NCEP Reanalysis Data');
   pause
end
end
```

Figure 1.7. *Continuing with the NCEPcdfPlay function, the display portion uses the MATLAB Mapping toolbox. Coasts are added to the plot, and a Miller projection is used.*

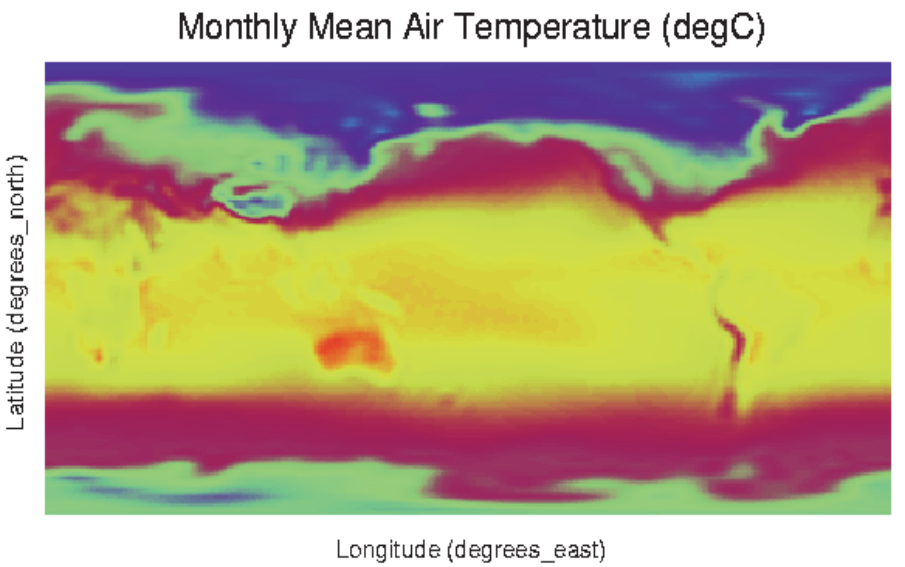


Figure 1.8. *NCEP Reanalysis data for monthly mean surface air temperature (C).*

In later chapters we will define our own data types. MATLAB looks for these user defined types in a directory called *@my_classes*. As you might imagine, the syntax and rules for specifying a new type are somewhat involved, and each object oriented language[7] has its own style. Until we need more complex data types, a new programmer should be

[7]The C++ language is the first and most advanced object oriented language. Others such as Java try to simplify the syntax by restricting some of the abstraction. Languages such as MATLAB and FORTRAN have added object oriented features but do not support many of the abstractions of C++.

aware of the following rules of thumb:

- "If you can think of 'it' as a separate idea, then make it a class."
- "If you can think of 'it' as a separate entity, then make it an object of a class" [23, p.10].

As new abstractions are introduced we will discuss why they make good sense. You can slice a cake many ways, and it is a matter of discretion which new concepts and syntax should be introduced. These may seem arbitrary at first.

1.4 ▪ A Cultural Note

The important quality of an abstraction is that it can be easily understood by the programming community that will use it. Since it is difficult to guess the most useful abstractions, the design of code architecture is iterative and fluid. Code is almost always developed by a team, and, for this reason, I encourage students to engage in collective efforts from the beginning of their projects. Planning the code architecture and developing suitable abstractions are best done with multiple inputs and perspectives. Working alone, there is no incentive to make your code readable, and, in consequence, no one, including yourself, will be able to read it at the conclusion of the project.

Large scientific code projects often involve tens to hundreds of team members, and maintaining, updating, and developing the capabilities on multiple computing platforms require software engineering discipline. An introduction to this discipline may be gained by using a version control system such as *subversion* to guarantee that team members' work is not lost or accidentally overwritten in coordinating code modifications. The simple version of *subversion* is the act of checking out code and on checking it back in to resolve conflicts. Testing procedures, though not required by the version control, are a next step in the development of a productive culture of programming. Though seemingly cumbersome at first, programmers learn to trust one another and gain confidence in their abilities based on the agreed upon procedures and coding conventions.

Chapter 2

What Is the Global Average Temperature of the Earth?

Some controversy persists over what a global average temperature is and how it can be meaningfully estimated. Since this is the key quantity used in discussing global warming, it is a topic that we should try to be clear about. The mathematics of this average should be the easy part, but this is complicated by the scattered collection of temperature observations over the surface of the globe.

The thermodynamics of the global temperature and warming is quite complicated, but, using a few simplifying assumptions, we can start to make sense of it.

2.1 ▪ Mathematics of a Global Average

The global average temperature, T_S, is an integral over the surface of the earth. Thinking of the surface of the earth as a sphere, S^2, the integral can be written

$$T_S = \frac{1}{|S^2|} \int_{S^2} T \, dA. \tag{2.1}$$

This notation needs to be explained, and the way in which global integrals could be computed needs to be described.

Suppose that temperature is given as a function of longitude (λ) and latitude (ϕ). The global integral of temperature is

$$\int_0^{2\pi} \int_{-\pi/2}^{\pi/2} T(\lambda, \phi) a^2 \cos\phi \, d\phi \, d\lambda. \tag{2.2}$$

The computation of integrals is accomplished through numerical quadrature rules. A quadrature rule specifies a set of points at which the function is evaluated and a set of weights associated with each of these points. The integral of the continuous function is then approximated by a finite sum of the discrete values,

$$\int_{S^2} f \, dA \approx \sum_{i=1}^{I} w_i f(x_i). \tag{2.3}$$

How to choose the location of the points and the values of the weights is an interesting numerical analysis study. Simpson's rule chooses the endpoints and midpoints of an

interval weighted with $\frac{1}{3}, \frac{4}{3}, \frac{1}{3}$. The way in which the weights are derived involves interpolating the point values with a quadratic polynomial and then explicitly integrating this approximation.

The integral may be viewed as a linear transformation (linear functional) that maps a discrete representation of the function into the reals. As such we look for a basis in which to express the matrix of the linear transformation. Let

$$L(f) = \int_{-1}^{1} f(x)dx, \tag{2.4}$$

where f is a polynomial of degree 2. The polynomials span a vector space with the usual definitions of polynomial addition and multiplication by a scalar. A basis for these polynomials is given by $1, x, x^2$. For these basis vectors, $L(1) = 2$, $L(x) = 0$, $L(x^2) = \frac{2}{3}$. In Simpson's rule the x_i are the left, middle, and right points of the interval, so $x_1 = -1, x_2 = 0, x_3 = 1$, and the set of equations for the weights is

$$w_1 \cdot 1 + w_2 \cdot 1 + w_3 \cdot 1 = 2 = L(1), \tag{2.5}$$

$$w_1 \cdot (-1) + w_2 \cdot 0 + w_3 \cdot 1 = 0 = L(x), \tag{2.6}$$

$$w_1 \cdot 1 + w_2 \cdot 0 + w_3 \cdot 1 = \frac{2}{3} = L(x^2). \tag{2.7}$$

The solution for the three unknown weights from the three simultaneous equations is the Simpson's rule weights. The matrix representing the linear transformation $L(f) : \Re^3 \to \Re$ is then

$$\left(\frac{1}{3}, \frac{4}{3}, \frac{1}{3}\right) \begin{pmatrix} f(-1) \\ f(0) \\ f(1) \end{pmatrix} = \frac{1}{3}f(-1) + \frac{4}{3}f(0) + \frac{1}{3}f(1). \tag{2.8}$$

There is a magical piece of mathematics that answers the following question: Can the accuracy of the formula be improved if the weights and the evaluation points are chosen together? The optimal answer is provided by the Gauss quadrature rules. As an example, consider the weights $\frac{5}{9}, \frac{8}{9}, \frac{5}{9}$ together with the Gauss points $-\sqrt{\frac{3}{5}}, 0, +\sqrt{\frac{3}{5}}$. The accuracy should be good for polynomials up to degree $2 \cdot 2 + 1$.

To compute an integral on the sphere, an analogous approach to quadrature rules may be followed. The longitudinal integration may be approximated with equally spaced points, and the latitudinal integration may use a different rule. Gauss quadrature rules are the norm for spectral models as they give the highest accuracy numerical approximation for functions defined on a sphere. But for equally spaced data on a lat-lon grid the trapezoidal rule is more appropriate. You have to remember about the area weighting, however, as points near the pole would incorrectly contribute more heavily to the sum than points on the equator unless weighted by area.

Exercise 2.1.1. *Write a MATLAB program to evaluate spherical integrals using these ideas. Check your answer by computing the surface area of the sphere and some other function that can be integrated analytically. How accurate is your method?*

The time variation enters a calculation of the earth's average temperature because we must ask over what period the average is taken. An easy way out is to consider time averaging as well as spatial averaging. In this way we can talk about the yearly average or monthly averages or even seasonal averages. For climate change discussions, we usually

average over a 20 or 30 year period as shorter periods do not sample the natural interannual variations with enough points. The average over a time period $[t^0, t^N]$ is an integral

$$T_{ave} = \frac{1}{(t^N - t^0)} \int_{t^0}^{t^N} T(t)dt \neq \sum_{n=0}^{N-1} \frac{1}{(t^{n+1} - t^n)} \int_{t^n}^{t^{n+1}} T(t)dt. \tag{2.9}$$

But is this the same, or not, as the average of monthly averages? How can you account for the fact that the months have different numbers of days? You have to think before you take an average of averages. What does the Central Limit Theorem say about this?

2.2 ▪ Analysis of Global Warming

The recent temperature of the earth shows a warming with a regional signature. The high latitude regions in the Northern Hemisphere are warming more rapidly than the ocean moderated Southern Hemisphere or the lower latitudes. But why take my word for it? You can analyze the monthly fields for yourself.

Exercise 2.2.1. *Use the NCEP Reanalysis data for monthly mean temperatures (see Figure 1.8) to compute average surface air temperatures for different months and annually. The changes in the global temperature are of what magnitude? Use an approximation for the spatial integral over the sphere to calculate the global average temperature.*

2.3 ▪ Numerical Solution of Ordinary Differential Equations

Before we can describe a dynamical model for the global average temperature, we need to know something about solving ordinary differential equations (ODEs). A first order ODE is an equation that involves the first derivative of a function. So if the unknown function is $u(t)$, we can write a general form as

$$u' = f(u, t), \tag{2.10}$$

where $f(u, t)$ is a given function of both the unknown u and t. (We use the prime to denote the derivative $\frac{du}{dt}$.) ODEs always come with another condition. For first order equations, the condition is typically an *initial condition* that specifies the value of the unknown function at some beginning time, e.g., $u(0) = u_0$.

Numerical solution of this equation can be derived in at least two ways: by approximating the derivative or approximating an integral. Let $u^n = u(t^n)$ be a shorthand notation for the value of the unknown function u at the time t^n. The numerical solution will find the values of the unknown function at the discrete set of time points, $t = t^1, t^2, \ldots, t^n, \ldots$. To approximate the derivative of u at time t^n, expand u in a Taylor series about this point.

$$u(t^{n+1}) = u(t^n) + u'(t^n)(t^{n+1} - t^n) + u''(t^n)\frac{(t^{n+1} - t^n)^2}{2} + \ldots. \tag{2.11}$$

Solving for the first derivative gives

$$u'(t^n) = \frac{u(t^{n+1}) - u(t^n)}{t^{n+1} - t^n} - u''(t^n)\frac{(t^{n+1} - t^n)}{2} + \ldots. \tag{2.12}$$

The discrete approximation drops the higher order terms

$$\frac{u^{n+1} - u^n}{\Delta t} \approx u'(t^n) = f(u^n, t^n) \tag{2.13}$$

and commits an error proportional to the stepsize $\Delta t = t^{n-1} - t^n$. Or, expressed as an advance over a time step,

$$u^{n+1} \approx u^n + \Delta t f(u^n, t^n). \tag{2.14}$$

Using the fundamental theorem of calculus, the ODE can be integrated over the time interval $[t^n, t^{n+1}]$ as

$$u^{n+1} - u^n = \int_{t^n}^{t^{n+1}} u'(t)dt = \int_{t^n}^{t^{n+1}} f(u(t), t)dt. \tag{2.15}$$

A numerical approximation of the integral also gives a numerical method for time stepping (or integrating) the ODE. The method derived from the Taylor expansion is called Euler's method and corresponds to approximating the integral as

$$\int_{t^n}^{t^{n+1}} f(u(t), t)dt \approx \Delta t f(u(t^n), t^n), \tag{2.16}$$

a left side rule. This method is an example of an *explicit method*, since the new time values are computed explicitly using old time values. By introducing an integral approximation that includes f evaluated at the new time level,

$$\int_{t^n}^{t^{n+1}} f(u(t), t)dt \approx \Delta t f(u(t^{n+1}), t^{n+1}), \tag{2.17}$$

an *implicit method* results. Since the value of $u(t^{n+1})$ is unknown, a nonlinear equation involving f must be solved for implicit methods. Using a trapezoidal or Simpson's rule gives a more accurate approximation, resulting in other implicit methods.

Exercise 2.3.1. *Write a MATLAB code to solve the ODE $\dot{u} = u$ with initial condition $u(0) = 1.0$. Use both the explicit and implicit Euler methods before experimenting with the built-in MATLAB ODE solvers.*

The structure of an ODE solver must address the question of how the right-hand side function is communicated to the solver. The calling program passes a *function handle* in this example of the implicit Euler method with a Picard, fixed point iteration used for solving the nonlinear system of equations.

The function is included in the same file and for the test problem is specified as

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}' = \begin{pmatrix} \sqrt{u_1} \\ \cos(u_2) \\ u_3 \end{pmatrix}. \tag{2.18}$$

The implicit Euler code requires that some information be held in an accessible place in order to call the Picard iteration code with the same function handle method.

2.4 ▪ A Simple Dynamic Model

We can develop a simple, box model [5] of the earth system by subdividing into atmosphere and ocean. Since we are ignoring the land, we call this an aqua-planet, which is not too bad an assumption given that 70% of the earth's surface is covered by water. Let T_a

```
function ImpEulerTest
% Implicit Euler ODE method for u' = f(u,t), u(0)= u0
% calls the Picard iteration with iteration formula in function F
% State variables, u(:,2) are held at two time levels, n and np1
%
% Initialize
  tend = 1.0;
  nsteps= 100;
  dt = tend/double(nsteps);
  time = 0.0;
  n=1;   np1=2; m = 3;  % m the dimension of the solution vector
 % Set up initial conditions
  u = ones(m,2);
  uout = zeros(nsteps,m);
  tout = zeros(nsteps);
% Run
  for nstep = 1:nsteps
      [u(:,np1)]=ImpEulerStep(@f, u(:,n), dt, time);
      time = time + dt;
      uout(nstep,:) = u(:,np1);
      tout(nstep) = time;
      ntmp = n;  n =np1; np1 = ntmp;   % Swap time level pointer
  end
% Finalize
  plot(uout);
  err = uout(nsteps,3)- exp(1.0)   % simple error measure
end
```

Figure 2.1. *For each of* nsteps *integrating from the initial time to the end time, using the time step dt, the solution is advanced using an implicit Euler method. To avoid copies of the solution between the n and $n+1$ time levels an index pointer is used. These kinds of tricks are common practice in coding for efficient and fast execution, but they make the code harder to read.*

```
   function uprime =f(u,t)
%  Function for ODE right hand side
   uprime = transpose([ sqrt(u(1)), cos(u(2)), u(3)]) ;
   end
```

Figure 2.2. *The solution is an m= three vector, and the forcing right-hand side consists of three familiar functions.*

and T_o be the time dependent temperatures of the atmosphere and the ocean. The fluctuation of the global average temperature is modeled using an ODE. The equation is solved numerically using a time integration method such as the explicit Euler method. Values of the parameters can be modified to study the sensitivity of the earth's average temperature.

The model equations are

$$C_a \frac{dT_a}{dt} + f(T_a) = Q + c_{ao}(T_o - T_a), \tag{2.19}$$

where

- T_a is the global average atmospheric temperature change from an reference value

```
function [unp1] = ImpEulerStep(f, un, dt, time)
%  Implicit Euler Step of ODE method
%  update follows u^{n+1} = u^n + dt*f( u^{n+1},t^{n+1} )
% Solve the nonlinear (implicit) equation using a Picard iteration
%
   global dtEuler;    % the internal step used by the method
   global tEuler;     % time for funtion evaluation
   global uold;       % old value for Euler method function
% Initial guess
   dtEuler = dt;
   tEuler  = time;
   uold = un;
   uk= un;
   tol =1.d-3;
   maxits = 100;
   [unp1,err, fpnorm,nits] = PicardIter(@F,uk,tol,maxits);
% Display solution and errors
%   unp1
%   err
%   fpnorm
%   nits
end
function [ukp1] = F(uk)
%  Iteration forumula for Backward Euler
%    for the ODE u' = f(u,t)
%  The variable dt is global and set in the calling program
      global dtEuler;
      global tEuler;
      global uold;
      ukp1 = uold + dtEuler* f(uk,tEuler) ;
end
```

Figure 2.3. *The ImplEulerStep needs to communicate the old solution value, time, and the time step to the Picard iteration function '@F' which specifies the nonlinear, fixed point equation to be solved or iterated. This communication has been accomplished by declaring several variables global in scope so that other functions can access their values. A variables scope in MATLAB is usually contained only within the function.*

(Kelvin);

- C_a is the atmospheric heat capacity $(J/m^3.K)$*depth(m) (0.75×10^3)*depth;
- $f(T)$ is the net radiation change at the tropopause due to internal dynamics of the climate system;
- Q is the change in net radiation crossing the tropopause from the reference value;
- $c_{ao} = 14.2(W/m^2.K)$,

$$C_o \frac{dT_o}{dt} = c_{ao}(T_a - T_o); \tag{2.20}$$

- T_o is the global average (mixed) ocean temperature change from an reference value (Kelvin);

- C_o is the total system heat capacity of deep ocean.

If the sun is assumed to give a constant input, then Q=0. The effect of extra absorption of long wave radiation by elevated CO_2 might increase the atmospheric heat source Q. The function f(T) can take various forms. We will use the form $f(T) = (f_b + f_w + f_i + f_d)T$. The term represents the feedbacks in the climate system. A positive coefficient gives a negative feedback and tends to stabilize the temperature. A negative coefficient gives a positive feedback and will cause the temperature to grow. The coefficient f_b represents the black body radiation from the earth to space and can be computed using the Stefan–Boltzmann law. The coefficient f_w represents the effect of water vapor on radiation. Water vapor (a greenhouse gas) absorbs long-wave radiation. The coefficient f_i represents the effect of ice reflecting incoming light, and f_d is the effect of dust.

Values given in various references are as follows:

- $f\,b = 3.75(W/m^2K)$;
- $f\,w = -1.7(W/m^2K)$;
- $f\,i = -0.8(W/m^2K)$ [Budyko [1] and Sellers [21], 1969] , = -0.3 (W/m^2K) [Lian and Cess [17], 1977], = −0.66 (W/m^2K) [Robok [20], 1983], = −0.44 (W/m^2K) [Chou et. al. [3], 1982];
- $f\,d = 0.75(W/m^2K)$ dust in the atmosphere.

Other values can be set from

- $C = depth(3.35x10^6(J/m^3K))$ [Sellers [21], 1969] or $C = 1.06(Wyr./m^2K)$ with depth= 10m;
- $Q = 4.0(W/m^2)$ [Ramanathan [19], 1989] due to greenhouse warming. Q can also vary according to variations in the solar constant. This is most affected by the orbital precession.

Much of climate science can be thought of as uncertainty about the processes lumped into f(T). The Fourth Assessment Report (AR4) of the IPCC presents the diagram in Figure 2.4 for the different factors in the radiation balance.

Exercise 2.4.1. *Write a MATLAB program to compute the solution to the coupled set of differential equations. Use this program to address various questions:*

1. *The depth of the ocean that interacts with the surface temperature is also a puzzle that directly affects the value of C. What happens if the depth is 100m or 1000m instead of 10m?*
2. *What is the climate sensitivity of this model under a doubling of CO_2? What is the timescale of the doubling response?*
3. *Modify the model to simulate a variable solar input with a periodic function in time modeling the earth's precession. Can you compute Milankovich cycles?*

We will now discuss a MATLAB code simulating the global average temperature (GAT.m) of the earth system by implementing this simple energy balance and feedback model. The code introduces MATLAB class structures defining a *linear dynamical system*

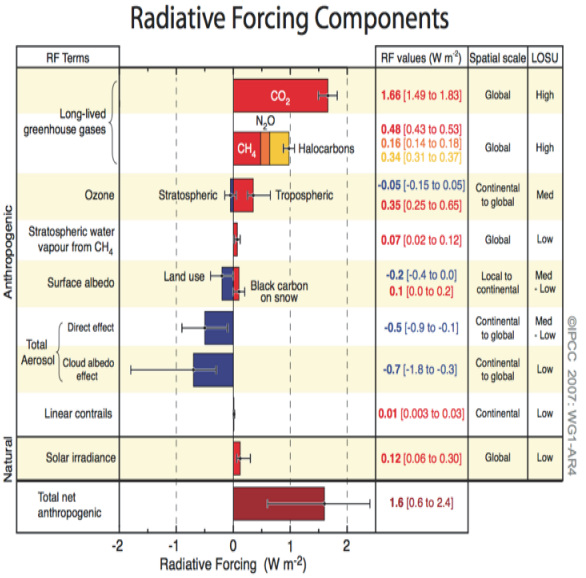


Figure 2.4. *Global Radiative Forcing. Source: IPCC - AR4 Working Group 1 [22].*

class (*@lds*) with the state variable as the atmosphere and ocean temeperature. The program will be divided into three phases: *begin*, *run*, and *finalize*. It is helpful for longer programs with many components to organize the computation into these phases just for clarity of programming. The initialization of the data structures and the problem are set up in the *begin* phase.

The linear dynamical system class is one of our user defined types. The dynamics are represented as a linear dynamical system with u as the state of the component, x as the input to the component, and y as the output of the component. This class defines the dynamics in terms of matrices so that

$$\dot{u} = Au + Bx(\text{dynamics}), \tag{2.21}$$

$$y = Cu + Dx(\text{observables}). \tag{2.22}$$

For nonlinear systems the right-hand sides are replaced,

$$\dot{u} = f(u,x,t) \text{ and } y = g(u,x,t). \tag{2.23}$$

The abstraction of a dynamical system is very powerful in classifying variables as input (forcing) variables and outputs (observables), while the internal state of the system is perhaps unobservable and hidden. Of course it is usually the state variables that appear in the equations of physics and that we are most interested in.

The syntax for specifying the user defined lds type is intuitive but complicated. We encourage the student to look at the code contained in the folder *my_classes/@lds* and read the MATLAB help about *Object Oriented Programming*.

The *run* phase of the program advances the dynamics through a series of time steps by calling an integration method such as explicit or implicit Euler in *step*.

The *finalize* phase of the code wraps things up and displays results. Figure 2.8 shows the response to CO_2 forcing as a temperature difference. The plotting of the dynamics

```
function [c, time, dt, nsteps] = LGAT_begin(c)
% Begin/Initialization method for LGAT
    Tatm = 0.0;   Tocn = 0.0 ;% initial temperature anomaly
    dt = 0.01;                % year (suggest dt =0.01,
    tend = 10;                % year to end (suggest tend = 10)
    time = 0.0;
    nsteps = tend/dt;         % number of steps to take
    c = lds('earth');         % linear dynamical system class
    c  = setstate(c,[Tatm;Tocn]);
    u = getstate(c);
    Tatm(1)= u(1); Tocn(1) = u(2);
%   Initialize coupled dynamics
    depthatm = 10000.0;       % depth of atmosphere (meters)
    depthocn = 100.0;         % mixed layer depth ocean (meters)
    Ca =750.0 *depthatm ; Co = 3.35e6*depthocn;  % (J/m3K)*meters
    fb = -3.75; fw = 1.7; fi = 0.8; fd = -0.75;  % W/m2K
    ftot = fb + fw + fi + fd;
    cao = 14.2;  % W/m2K
% Load the dynamics matrices
    AA = [ (ftot - cao)/Ca, cao/Ca ; cao/Co, -cao/Co] ;
    BB = [1.0/Ca, 0.0 ; 0.0, 0.0];
    CC = [1.0, 0.0 ; 0.0, 1.0];
    DD = [0.0, 0.0 ; 0.0, 0.0];
    c = setdynamics(c,AA,BB,CC,DD);
end
```

Figure 2.5. *The* begin *phase of the linear global average temperature model where initialization of the dynamical system takes place. The method* setstate *assigns values to the hidden state variables, while* getstate *retrieves them for public display. The action of the linear dynamical system is defined through the matrices* A, B, C, *and* D *that are denoted by* AA, BB, CC, DD *in the code.*

phase space plots the temperature of the atmosphere against the temperature of the ocean as a curve.

```
function [Tatm, Tocn ] = LGAT_run(c, time, dt, nsteps)
% GAT_run: run method for  Global Average Temperature (anomoly)
       Tatm = zeros(nsteps);
       Tocn = zeros(nsteps);
%  Start the timestepping loop
    for nstep = 2:nsteps
%  Set the input (forcing)
       Q = 4.0;  % anomolous energy balance (W/m2)
       x=[Q ;0.0] ;
       dts = dt*(60*60*24*365);  % Convert years to seconds
                                 % timestep units match Q
       c = step(c,time,dts,x);
       time = time + dt    % Update time and print
       y = getobservable(c);
%       u = getstate(c);   % Since observable is state no need
                           % to retrieve the state
       Tatm(nstep) = y(1);
       Tocn(nstep) = y(2);
    end
end
```

Figure 2.6. *The* run *phase manages the timestepping and advancement of the dynamical system through the* step *method. Observables are retrieved using the* getobservable *method, and the time history is saved in the Tatm and Tocn arrays.*

```
function LGAT_final(Tatm, Tocn, dt, nsteps)
% Finalize method of LGAT
% Plot the time series
    taxis = 0: dt : (nsteps-1)*dt ;
    plot(taxis, Tatm, taxis, Tocn)
    xlabel('Years')
    ylabel('\Delta Temperature')
    title('Earth Atm and Ocn Temperature Change')
    pause
% Plot the state space
    plot(Tatm, Tocn)
    xlabel('Tatm')
    ylabel('Tocn')
    title('Earth Phase Space')
end
```

Figure 2.7. *The* final *method plots the solution produced by integrating the dynamical system.*

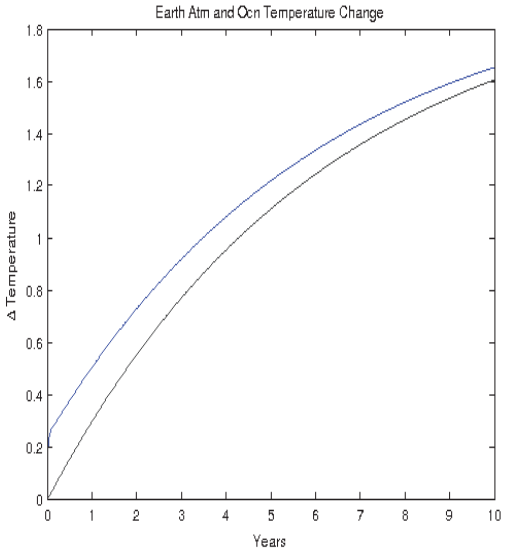


Figure 2.8. *Atmosphere and ocean warming in response to increased greenhouse gasses according to the simple Linear Global Average Temperature (LGAT) model. The ocean temperature lags behind the atmospheric temperature.*

Chapter 3

Approximation of the Advection-Diffusion Equation

Exercise 3.0.2. *Use the control volume discretizations to numerically solve the advection diffusion equation*

$$\frac{\partial u}{\partial t} + \mathbf{v}\cdot\nabla u = \alpha\nabla^2 u + f \ \textit{in}\ [0,2\pi]\times[0,2\pi]. \tag{3.1}$$

Let $\mathbf{v}$ be a given velocity vector field, $\mathbf{v} = (\sin x, \cos y)$, with doubly periodic boundary conditions $u(0,y,t) = u(2\pi,y,t)$ and $u(x,0,t) = u(x,2\pi,t)$. Choose the initial conditions as $u(x,y,0) = 0$ and integrate forward in time to $t = 10$ days with a diffusion coefficient $\alpha = 1$. Suppose that f is a point source (δ-function) at the center point and $f(\pi,\pi,t) = e^{-(t-\tau)^2}$, $\tau = 1.0$.

3.1 ▪ Divergence and Gradient Compatibility

To discretize the advection-diffusion equation (3.1) with the control volume method we integrate the equation over a small volume $\mathcal{V}$. The divergence theorem is applied to the Laplacian diffusion term in a straightforward manner, but we are left with the troublesome advection term. One way of treating this term is to use the calculus identity from the product rule,

$$\nabla\cdot(\phi\mathbf{v}) = \mathbf{v}\cdot\nabla\phi + \phi\nabla\cdot\mathbf{v}. \tag{3.2}$$

Integrating over the control volume,

$$\int_{\mathcal{V}} \mathbf{v}\cdot\nabla\phi = \int_{\mathcal{V}} \nabla\cdot(\phi\mathbf{v}) - \int_{\mathcal{V}} \phi\nabla\cdot\mathbf{v}. \tag{3.3}$$

Applying the divergence theorem to the first term on the right, and approximating the ϕ as a constant, $\bar{\phi}$, on the control volume gives an approximation for the advection term as

$$\int_{\mathcal{V}} \mathbf{v}\cdot\nabla\phi \approx \int_{\partial\mathcal{V}} \phi\mathbf{v}\cdot\mathbf{n}da - \bar{\phi}\int_{\partial\mathcal{V}} \mathbf{v}\cdot\mathbf{n}da. \tag{3.4}$$

The location of velocities on the control volume faces and ϕ at cell centers makes this a discretization compatible with a numerical version of the product rule. In particular, if the velocity field is divergence free, $\nabla\cdot\mathbf{v} = 0$, then the numerical approximation will be compatible with conservation of ϕ in the numerics.

3.2 ▪ A Word of Warning

The leapfrog explicit scheme can be used for the advection terms, but care must be taken with the diffusion terms to avoid the Richardson's method when $\mathbf{v} = 0$. If you choose an explicit method, then evaluating the diffusion term at t^{n-1} will result in the Dufort–Frankel scheme. If an implicit method is chosen, then a linear algebra ($Ax = b$) solver is required.

Approximate the source as deposited in a single control volume. Note that the total amount deposited should not be grid dependent; it should be equal no matter how small the control volumes are. That is, $\int_0^{10}\int_0^{2\pi}\int_0^{2\pi} f(x,y,t)dxdydt = C$ is problem dependent.

A numerical solution based on MATLAB code is shown in Figure 3.1.

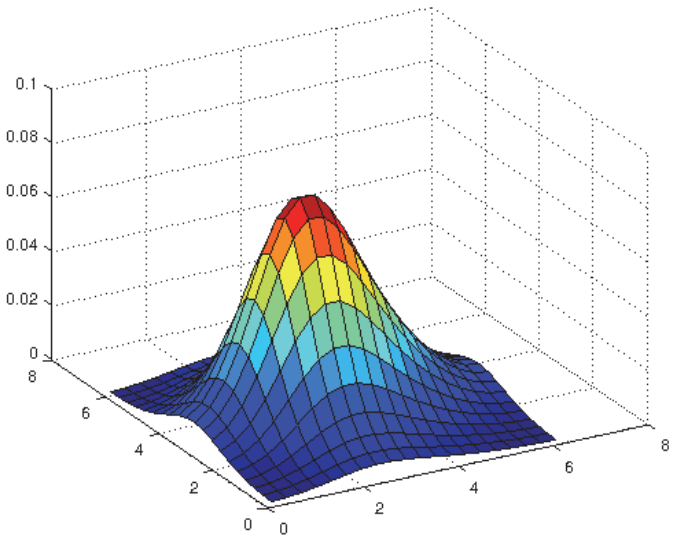


Figure 3.1. *Solution of the advection-diffusion problem on a* 20×20 *periodic control volume grid at time* $t = 10$ *days. Diffusion dominates advection with* $\alpha = 1.0$ *and the point source is seen to have spread out while shifting its center only a small amount.*

3.3 ▪ Objects Revisited

The advection-diffusion equation is representative of the kind of equations that must be solved numerically to simulate climate and weather. There is an advection term, $\mathbf{v} \cdot \nabla u$, and a diffusion term, $\nabla^2 u$, as well as the time dependant term, $\frac{\partial u}{\partial t}$. The first two are spatial terms, while the last is only in the time dimension. What does this mean in terms of the abstractions and organization of a program to solve the equation?

Underlying the spatial terms is a discretization on a grid. So we might propose that the grid is an object with methods from its class defined to compute the discretization. The

time dimension is usually not considered part of the grid,[8] and we have already introduced methods for integrating and discretizing the time derivative. So let's focus on a grid object for the control volume method; call it *cvgrid2d*.

Formulating a class abstraction requires that you identify the fundamental concepts belonging to the idea. In the case of grids, there are sets of points, with coordinates and a special number, or numbers, specifying the resolution of the grid. For ease, let's consider the data structure of the grid to be an $N \times M$ array of coordinates. But what does each coordinate represent? For the control volume method, each point (x_i, y_j) of the grid represents the center of a control volume.

But that is not all. The edges of the control volume must be specified somehow. We might do this by introducing another array of half or edge values. But we could just as well make the assumption and agree on the convention that the edges lie halfway between the centers. Some things we want to do will also require that we extend the grid to include boundary values. But there is no reason that the outside world needs to know about this. We could compute the extension internally and keep it as a private part of the object. There are always multiple ways to form the abstractions, and it can be confusing if the assumptions are not clearly stated or clearly part of the programming team's folklore. So let's agree for the purposes of this exercise to keep it simple and only implement uniform grids in each direction, with cell boundaries at the half points. The essentials in this case are, on input, the numbers N and M and the bounds of the region $[x_{left}, x_{right}]$ and $[y_{bottom}, y_{top}]$. The rest can be figured out internally.

The code in Figures 3.2 and 3.3 illustrates an implementation in MATLAB of a two-dimensional control volume grid object. We divide the constructor code for the object into two cases; first Figure 3.2, when no arguments are used, and second Figure 3.3, when properties are set.

When the grid object constructor is called with seven arguements, it has all the information requried to set the internal properties of the grid. A fragment of the code to set up the control volume centers and edges is shown in Figure 3.3.

The data storage associated with the object is really a *structure*. It holds the particulars associated with a given instantiation (realization) of the class. This storage is set aside in the similarly named routine of the class, called the *constructor*. What goes with the object and its storage are the methods that work with this data. In the case of a grid, we need an operator library—calculations such as $\nabla^2 u$ and $\mathbf{v} \cdot \nabla u$. If you think about it, these discrete numerical operators are implemented differently for every grid type, so they belong to the grid object and need to use its private data. In Figure 3.4, the Laplacian is computed for a field F defined on a control volume grid.

With these objects and classes as building blocks, you can build a code to solve the advection-diffusion equation without getting lost in subscripts and implementation details. For example, see Figure 3.5. The class may need to be extended to do what you want, but then others can easily use your code. Whenever possible, data should be hidden to simplify the user interface. For example, in the *cvgrid2d* example, your colleague need never know that to compute a Laplacian or perform interpolation on the grid requires that a halo region, a grid extension, must be filled. On the other hand, this makes it hard to change the type of boundary conditions on the region. Can you think of ways to generalize the specification of the boundary conditions without complicating the user interface?

[8] The semi-Lagrangian transport method is an exception where the advection term and time derivative are considered together.

```
function G = cvgrid2d(varargin)
% cvgrid2d creates an extended, 2d, control volume grid
%  Usage:  Adv = cvgrid2d(N,M,xleft,xright,ybot,ytop)
%   Once the extended grid is established, with index mappings for
%   halo updates, then interpolation is supported as a method
%  along with discrete operators Laplacian and gradient.
% Input:  nx  - number of control volume cells in the x-direction
%         ny  - number of control volume cells in the y-direction
%         xleft, xright  - edges of region in x
%         ybot, ytop     - edges of region in y
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  The grid data structure contains
%    nx, ny           the number cell centers of internal cells
%    xc,yc   -  cell centers of the uniform grid nx x ny
%               oriented left to right, bottom to top.
%  The edge grid for velocity and advection contains
%    nxe, nye  - cells in each direction
%    xe,ye   - the cell edge coordinates
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
switch nargin
case 0
% no input arguments so create a default component
 G.xc      = [];
 G.yc      = [];
 G.xe     = [];
 G.ye     = [];
 G.nxe    = 0;
 G.nye   = 0;
 G.nx     = 0;
 G.ny   = 0;
 G.nxpt   = 0;
 G.nypt   = 0;
 G.bctype ='none';
 G = class(G,'cvgrid2d');
case 1
% one input argument so check if it is a component already
 if (isa(varargin{1},'cvgrid2d'))
   G= varargin{1};
 end
```

Figure 3.2. *A two-dimensional control volume grid constructor when not much information is supplied. The internal pointers to storage for the object are set up but not allocated. This is standard procedure for class definitions.*

```
case 7
 nx   = varargin{1};
 ny   = varargin{2};
 xleft   = varargin{3};
 xright  = varargin{4};
 ybot    = varargin{5};
 ytop    = varargin{6};
 bctype  = varargin{7};
%for control volume grid, centers and edges
 xc  = zeros(nx,1);
 xe  = zeros(nx+1,1);
 yc  = zeros(ny,1);
 ye  = zeros(ny+1,1);
%Fill the coordinate arrays
 nxpt = 1;
 dx = (xright - xleft)/nx;
 dy = (ytop - ybot)/ny;
 xe(1) = xleft;
 for i = 2:(nx+1)
   xe(i) = xe(i-1) + dx;
 end
 ye(1) = ybot;
 for j = 2:(ny+1)
   ye(j) = ye(j-1) + dx;
 end
% Put the cell centers at cv centroid, halfway between edges
 for i = 1:nx
  xc(i) = 0.5* ( xe(i) + xe(i+1) );
 end
 for j = 1:ny
  yc(j) = 0.5* ( ye(j) + ye(j+1) );
 end
 G.xc = xc;
 G.yc = yc;
 G.xe = xe;
 G.ye = ye;
 G.nxe       = length(G.xe);
 G.nye       = length(G.ye);
 G.nx        = length(xc);
 G.ny        = length(yc);
 G.nxpt      = nxpt;
 G.bctype    = bctype;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% end cvgrid2d fill
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 G = class(G,'cvgrid2d');
otherwise
 error('Wrong number input arguments: cvgrid2d class constructor')
end
```

Figure 3.3. *A two-dimensional control volume grid constructor building the edge and center coordinates. With this information the grid object G is ready to use. The methods of the class have access to G and items can be retrieved using the* get *method.*

```
function LaplF = cvLapl(G,F)
% Calculate the Laplacian of F at the cv centers.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Input
% A is the extended control volume (cvgrid) on which fields are
%  defined, in particular [A.X,A.Y] and [A.Xe A.Ye] are the
%  meshgrid produced cell center points and extended cell edge
%  coordinate arrays.
% F is nxm field of values.
% Output:
% LaplF is the resulting Laplacian at cell center points
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nx   = G.nx;
ny   = G.ny;
nxpt = G.nxpt;
nypt = G.nxpt;
xe = G.xe;
ye = G.ye;
% Extend the grid using the private method cvextend
[Xex,Yex,Fex]= cvextend(G,F);
for j = (1+nypt):(ny+nypt)
  for i = (1+nxpt):(nx+nxpt)
   ic = i-nxpt;    % Center point indices
   jc = j-nypt;
   aj = ye(jc+1) - ye(jc);  % Cell face areas, da
   ai = xe(ic+1) - xe(ic);
   LaplF(ic,jc) = ...
        (Fex(i+1,j) -Fex(i,j))/(Xex(i+1) -Xex(i))   * aj ...
      - (Fex(i,j) -Fex(i-1,j))/(Xex(i)   -Xex(i-1)) * aj ...
      + (Fex(i,j+1) -Fex(i,j))/(Yex(j+1) -Yex(j))   * ai ...
      - (Fex(i,j) -Fex(i,j-1))/(Yex(j)   -Yex(j-1)) * ai ;
end
end
```

Figure 3.4. *Control volume discretization of the Laplacian assuming quantities at cell centers on an extended grid with halo regions setting the boundary conditions.*

```
function ADeq
%
%  Advection-diffusion program for the cvgrid2d class.
%  Field values are at the cell center points
%  Velocities are at cell edges
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;
%Uniform grid initialization
nx = 20;
ny = 20;
xleft = 0.0; xright = 2.0*pi;
ybot = 0.0;  ytop = 2.0*pi;
bctype = 'periodic';
%bctype = 'zero';
%  Set the control volume grid
G = cvgrid2d(nx,ny,xleft,xright,ybot,ytop,bctype);
%  Define timestepping and initial conditions
tstart = 0.0;
tend   = 10.0;
dt     = 0.1;    % time step (also in days)
[U,V,F] = init_case(G);
xe = get(G,'xe');
ye = get(G,'ye');
dx = xe(2)-xe(1);
dy = ye(2)-ye(1);
vol = dx*dy;
[X,Y]= meshgrid(xe,ye);
quiver(X,Y,U,V)
pause
xc = get(G,'xc');
yc = get(G,'yc');
surf(xc,yc,F);
pause
% Solve the advection-diffusion equation
alpha = 1.e0  % diffusion coeffecient
time = tstart;
surf(F)
while (time<tend)
 Fadv  = cvAdvect(G,U,V,F);
 Flapl = cvLapl(G,F);
 S = source(F,time,vol);
 Fnew = F + dt*(alpha*Flapl -Fadv) + dt*S;  % explicit Euler
 time = time + dt
  surf(xc,yc,Fnew)
  pause
 F = Fnew;
end
```

Figure 3.5. *After initializing the control volume grid G, initial conditions must be set for velocity U, V, and F. A time stepping loop advances the solution using an explicit Euler method with a timestep of $dt = 0.1$. The solution surface is plotted at every timestep.*

Chapter 4

Barotropic Modes of the Atmosphere

The shallow water equations on the sphere in advective form can be written

$$\frac{dq}{dt} = 0, \tag{4.1}$$

$$\frac{dh}{dt} = -\delta h, \tag{4.2}$$

$$\mathbf{v} = \mathbf{k} \times \nabla\psi + \nabla\chi, \tag{4.3}$$

$$\xi = \nabla^2\psi, \tag{4.4}$$

$$\delta = \nabla^2\chi. \tag{4.5}$$

The BV.m code solves the barotropic vorticity equation using the semi-Lagrangian transport method for updating the vorticity assuming that $\delta = 0$. The spectral method is used to solve for the stream function and to compute the velocity from the stream function. Since h does not change when $\delta = 0$, the definition of potential vorticity is simply $q = \xi + f$, where $f = 2\Omega \sin\phi$ is the Coriolis term. When δ is not zero then h is changing, and we must use the definition of potential vorticity as $q = \frac{(\xi+f)}{h}$. The set of equations above then models the shallow water solution.

What is interesting is how the barotropic structures of the atmosphere are affected by the divergence. To explore this we will take a typical barotropic solution, a Rossby wave, and perturb it by introducing a divergence.

The specified divergence is in lieu of another prognostic equation that would predict the δ and χ and form a complete nonlinear shallow water system. The δ will act as a perturbation, forcing the system much like the thermal heating of the earth's surface or the effect of surface topography on the momentum. In fact, the δ time variation is proportional to the $\nabla^2 h_s$, where h_s is the surface height.

The surface perturbation that we will use is from the mountain test case, case 5. The initial conditions will be case 6, i.e., a "stable" mode with Rossby wave number four. We will assume that all the other terms of the δ prognostic equation are in equilibrium and that the tendency $\frac{\partial\delta}{\partial t}$ is only from the topography. Since this change to the stable flow puts the system out of geostrophic equilibrium, we will study the response as the system adjusts. We will simulate that adjustment by forcing the δ tendency to zero through time using a multiplier as

$$\delta = \delta_0 e^{-\frac{t}{\tau}}, \tag{4.6}$$

where δ_0 is the initial delta tendency and $\tau = 1$ day is a relaxation time constant.

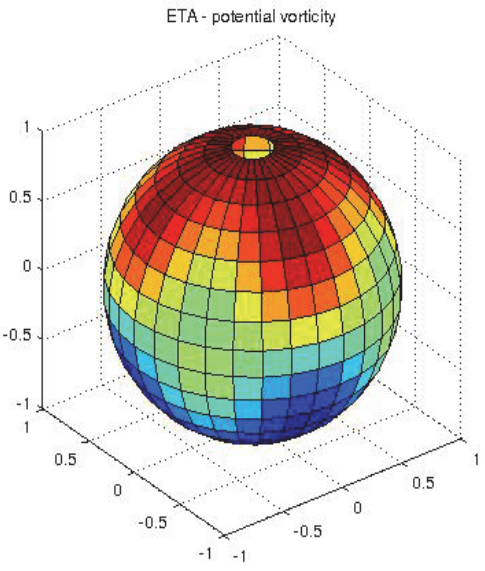


Figure 4.1. *Potential vorticity solution of the barotropic equation using BV.m. A stable wave five propagates zonally.*

The project involves four tasks:

1. Simulate stable Rossby waves with BV.m.
2. Add the h equation in advective form to BV.m.
3. Perturb the h equation's right-hand side with a specified δ.
4. Simulate 1, 5, 10, and 15 days from Rossby wave initial conditions, and examine the behavior of the solutions compared to the stable Rossby waves.

4.1 ▪ Algorithm for Modified BV

The spectral method will be used to solve diagnostic relations, and the semi-Lagrangian transport will be used to update the prognostic equations.

Algorithm.

- *Begin: Initialize the* h, q *fields as* slt_fields. *Initialize* $\xi, \psi, \delta, \chi, h_s$ *as* spectral_fields.
- *Run a time step loop* 1: nsteps.
    1. *adjust* δ *tendency for the h right-hand side*
    2. *use SLT to find particle paths and departure points*
    3. *interpolate RHS to mid points*
    4. *use SLT to advance time levels of q and h*

5. *find* $\xi = qh - f$
6. *solve using spectral for* ψ *and* χ *from* ξ *and* δ
7. *compute* $U = u\cos\phi$ *and* $V = v\cos\phi$
8. *update time and iterate loop*

- *Finalize the output plots*

4.2 ▪ The Spectral and Semi-Lagrangian Classes Used in BV.m

The abstraction of the *spectral_field class* specifies a variable to be a spectral field, i.e., one that has values on a Gaussian grid and also in terms of spectral coefficients. In other words, a field that is amenable to spectral analysis and synthesis. Before the field can be defined a *gauss_grid* of a given resolution must be defined. The syntax for the instantiation of these two objects is

```
G = gauss_grid('T10',nj);
PSI= spectral_field('Streamfunction',G);
```

where nj is the number of latitude lines used in the Gaussian grid. The methods of this class perform various computations with the spectral representation of the field. In particular, analysis, synthesis, divergence, curl, and Laplacian operators can be evaluated. The Helmholtz and Laplace equation can also be solved calling methods of this class. Besides the description of the spectral method in the text [6], the MATLAB codes are described in [8]. As per MATLAB convention, the code for this class lives in a folder called *@spectral_field* inside the *my_classes* folder on the MATLAB path.[9]

The other user defined class in the program BV.m is the *slt_grid* class. Functions of the spectral field class can be transferred to the slt_grid which extends their periodic values at longitude zero and 2π as well as extensions for interpolation at the poles. Semi-Lagrangian interpolation as well as particle tracking are methods of this class. These methods are used to find SLT departure points and interpolate the advected vorticity in BV.m with the syntax

```
A = slt_grid(xg,yg);
[XD,YD,XM,YM] = slt_particle(A,Uhalf,Vhalf,dt);
EtaD   = slt_interp(A,Etan,XD,YD);
```

The transfer of data into and out of these objects is accomplished using the *get* and *put* methods of each of the classes. This programming requirement of MATLAB objects makes the code a little bit of a cludge; a more complete implementation of the user defined data types might provide for type conversion and operator overloading, but for now these *gets* and *puts* can be thought of as simple copies to the appropriate data structures. Thus to retrieve the gridpoint values of a spectral field the syntax is

```
Un = get(U,'gp');
```

4.3 ▪ Analysis of the Solution Using SWAN

A Shallow Water Analysis (SWAN.m) program that computes the characteristics of the solution including the power spectra is illustrated in Figure 4.3. This is part of a MATLAB code that organizes the simulation into phases similar to the component models and stages

[9] MATLAB code for BV.m and the classes described is available online.

of the Earth System Modeling Framework (ESMF). A model object is specified, and begin, run, and finalize methods are called. The main program is shown in Figure 4.2. The SWAN computation and display is part of the *SWE_Analyze* method.

```
% Set up the model buffer
model.name = 'SWE spectral, semi-implicit';
model.hgrid = [];       % horizontal grid
model.fields = [];      % fields, u,v,phi, etc
model.control = [];     % control parameters
model.constants = [];   % physical constants, etc...
%
model = SWE_Begin(model);
model = SWE_Run(model);
model = SWE_Finalize(model);
model = SWE_Analyze(model);
clear all;
```

Figure 4.2. *Main calling program for a shallow water equation simulation with an* Analyze *phase.*

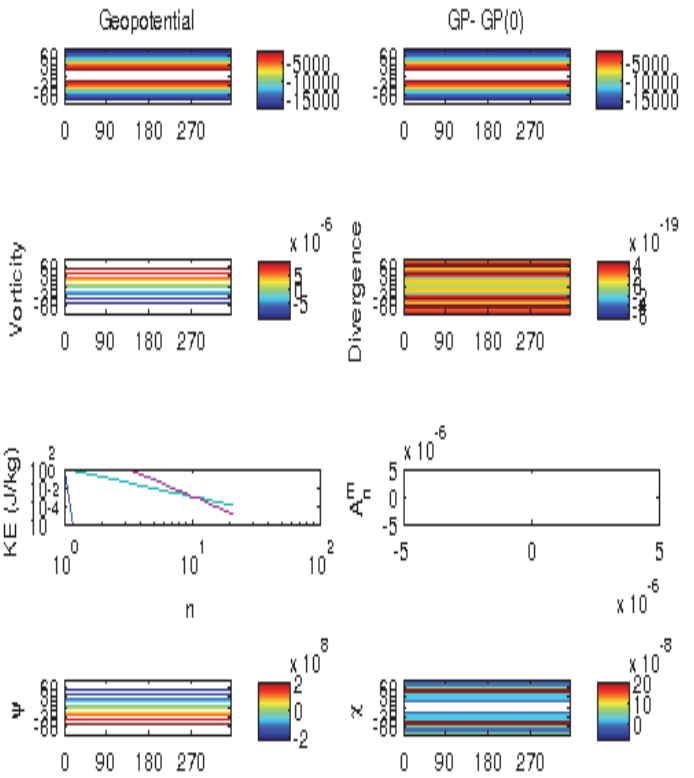


Figure 4.3. *Shallow Water Analysis (SWAN) dashboard. Along with plots of the solution fields, analysis of the kinetic energy spectrum and the divergence spectrum are included.*

Chapter 5

Vertical Structure of the Atmosphere

5.1 ▪ Vertical Structure Equation

The vertical structure equation (VSE) is [4]

$$(p^2 u_p)_p + q u = 0, \tag{5.1}$$

where the vertical coordinate for the atmosphere is pressure and $q = \frac{RT}{gh}$ are constants of an isothermal atmosphere. The boundary conditions are $u_p + \frac{\kappa}{p} u = 0$ at $p = 0, p_s$. The choice of the upper boundary at zero pressure effectively places the top of the atmosphere at an infinite height resulting in a singular Sturm–Liouville problem [7]. This is the problem that Charney coped with poorly, but in this exercise we may fair no better.

Here we use a compact method applied to the VSE, and the eigenvalues (modes) are computed. It is interesting that these modes also appear in a normal mode expansion of the hydrostatic (or nonhydrostatic [18]) equations of motion from a separation of variables. The VSE modes are indicative of the way the atmosphere responds dynamically in the vertical. Any discretization of the vertical can be viewed as giving an approximation of the VSE modes; see Figure 5.1. Depending on how that discretization is developed, it may or may not be true to the vertical operator. This has implications for the type of discretizations that are acceptable, adequate resolution, and correct boundary conditions that should be applied in the vertical.

5.2 ▪ A Lorentz Vertical Grid Class

Any discretization in the vertical must specify an underlying grid and location of the essential atmospheric state variables. So before introducing the discretization of the VSE, (5.1), we will introduce a user defined type for one particular choice of the vertical grid and atmospheric column representation. These are the *lvgrid* and *lvcolum* types. One of the functions of this class computes the standard atmosphere, for example, given values of pressure. The top of the model must be specified for the class, a value that essentially regularizes the singular Sturm–Liouville problem. The grid choice is the same as the Community Climate System Model CCSM4 and is referred to as a Lorentz grid.

The purpose of the Lorentz vertical grid class is to set up storage and initialize variables associated with the vertical. For example, the number of layers k as well as the full z_k and half layer $z_{k+1/2}$ coordinate values ranging between one and zero. At half values are the

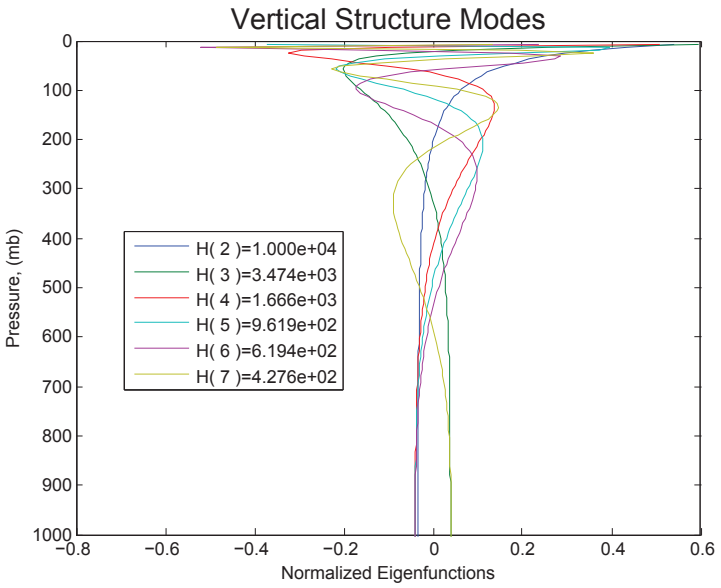


Figure 5.1. *The modes of the VSE as computed with a compact method. The oscillatory behavior of higher modes is evident as the upper ($p = 0$) boundary is approached.*

continuity and vertical momentum variables z, $\dot{z}$ and π, $\ln \pi$, and $\ln p$. At integer values are the horizontal momentum and thermodynamic variables U, V, T, q, and p. The methods of the class compute various terms, such as fluxes, derivatives, and integrals on the grid. To instantiate a Lorentz grid the initialization program can be constructed as illustrated in Figure 5.2.

```
%
%  First set up a lvgrid and lvcolumn
%
g = lvgrid(96);  % resolution of the column specified here
KK = get(g,'KK');
zk = get(g,'zk');
zh = get(g,'zh');
c = lvcolumn('vertical',g);
%
%  Other initializations ....
[Z,H]= vse(c)    % solve the vertical structure equation
```

Figure 5.2. *Set-up for a vertical column of the atmosphere using the Lorentz grid and column class. The object g and c are the structures holding all the dynamic and thermodynamic variables. After setting pressures for the grid points, the discretization of vertical structure equation can be numerically solved.*

5.3 ▪ A Compact Discretization

Another interesting discretization that yields spectral convergence is a compact method based on the Laguerre polynomials [14]. The approximation is based on discrete points of a grid $\{x_j\}$, and there are only a finite number of them since we cannot keep running out to infinity on a computer. A compact method for the derivative [15] means an approximation of the form

$$\sum_{j=L}^{M} A_{ij} u_x(x_j) = \sum_{j=J}^{K} B_{ij} u(x_j). \tag{5.2}$$

The matrix A and B will be determined to get an accurate approximation about the point x_i. In terms of a stencil, the formula will use several implicit derivative points and several function value points. These do not need to be on the same grid; for example, staggered grids can be used. The compact procedure works for more than a derivative. We could use it for interpolation, quadrature, or, actually, any linear operator. To make this generalization, write the unknown point values as $\mathbf{d}$ and the known point values as $\mathbf{p}$. The compact approximation seeks matrices $\mathbf{A}$ and $\mathbf{B}$ such that $\mathbf{Ad} - \mathbf{Bp} = 0$,

$$(\mathbf{D}, \mathbf{P}) \begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix} = 0, \tag{5.3}$$

where $\mathbf{D}$ and $\mathbf{P}$ are Vandermonde matrices for $\mathbf{d}$ and $\mathbf{p}$.

I'd like to propose that we choose the method to be exact on the first few Laguerre polynomials. That is, we will have an exact approximation for the unknown for functions in a subspace of the larger space when evaluated at the grid points. The equation for the method coefficients evaluating a derivative ($\mathbf{d} = u_x$, $\mathbf{p} = u$) is

$$\begin{pmatrix} 1 & \dots & 1 & 0 & \dots & 0 \\ L_0'(x_L) & \dots & L_0'(x_M) & -L_0(x_J) & \dots & -L_0(x_K) \\ L_n'(x_L) & \dots & L_n'(x_M) & -L_n(x_J) & \dots & -L_n(x_K) \end{pmatrix} \begin{pmatrix} A_i^T \\ B_i^T \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{5.4}$$

Suppose we let $\mathbf{d} = L(u)$ where L represents the self-adjoint linear operator of the VSE, (5.1). Then the equation for the method coefficients is

$$\begin{pmatrix} 1 & \dots & 1 & 0 & \dots & 0 \\ \lambda_0 L_0(x_L) & \dots & \lambda_0 L_0(x_M) & -L_0(x_J) & \dots & -L_0(x_K) \\ \lambda_n L_n(x_L) & \dots & \lambda_n L_n(x_M) & -L_n(x_J) & \dots & -L_n(x_K) \end{pmatrix} \begin{pmatrix} A_i^T \\ B_i^T \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{5.5}$$

To solve the equation $Lu = f$ using the compact method, note that $\mathbf{A}^{-1}\mathbf{B}u = f$ on a pointwise basis, so that $u = \mathbf{B}^{-1}\mathbf{A}f$. There is some assembly required to deal with the implicit points solving on the whole grid.

The VSE code implementing the compact method and solving the Sturm–Liouville problem is shown in Figures 5.3 and 5.4.

```
function [Z,H] = vse(c);
%usage [Z,H] =vse(c);
% Solve the vertical structure equation for the column c
%  Note:  this is an eigenvalue/eignfunction problem for H and Z.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Inputs:
%  c - initialized Lorentz vertical column.
%      This will include values for
%      Ts, \theta, T, and p at the grid layers and surfaces
%                  of the LV grid.
%      From these \gamma can be comuputed for each layer.
%      \gamma is the static stability parameter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Outputs:
%  Z - matrix of eignfunction columns
%  H - vector of eigenvalues  (dimension is KK)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% First get the LVgrid values associated with the column
g  = get(c,'LV');
KK = get(g,'KK');
zk = get(g,'zk');
zh = get(g,'zh');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize half and integer values
 gammak = zeros(size(zk));
 gammah = zeros(size(zh));
 Dpk    = zeros(size(zk));
 Pk     = zeros(size(zk));
 Dph    = zeros(size(zh));
% Extract some fields and thermodynamics constants from the column
P = get(c,'P');
T = get(c,'T');
theta = get(c,'theta');
Rgas = get(c,'Rgas');
Ts   = get(c,'Ts');
gravty = 9.806;  % m/s^2
% Set up other variables
for k = 1:KK
 Pk(k) = 0.5*(P(k)+P(k+1));  % Pressure at the integer levels
 Dpk(k) = P(k+1)-P(k) ;      % \Delta p_k
end
% Compute static stability parameter, gamma, on the zh grid
for k = 1:KK-1
 gammah(k) = - (T(k)+T(k+1))/(theta(k)+theta(k+1)) * ...
                ( theta(k+1) - theta(k))/(Pk(k+1)-Pk(k));
end
 gammah(KK) = -T(KK)/theta(KK) * ...
              ( theta(KK)-theta(KK-1))/(Pk(KK)-Pk(KK-1));
 gammah(KK+1) = gammak(KK);
for k = 1:KK-1
 Dph(k+1) = Pk(k+1)-Pk(k); % \Delta p_{k+1/2)
end
 Dph(1) = Dpk(1);
 Dph(KK+1) = Dpk(KK);
% fix the bottom and top
```

Figure 5.3. *Initialization and set-up for the VSE solution.*

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Set up system matrix:  AZ = lambda Z, where lambda = - 1/(gH)
% The discretization is based on control volumes
%  centered in the integer levels of a Lorentz vertical grid.
%  Half index values are obtained by averaging and boundary values
%  are extrapolated (linearly) from interior points, eg.
%    Zs = Z_KK + (Ps - P_KK)/(P_KK - P_KK-1) * (Z_KK - Z_KK-1 ).
A = zeros(KK);
for k = 2:KK-1
  A(k,k-1) = P(k)/(Rgas*gammah(k)*Dph(k)*Dpk(k));
  A(k,k) = - P(k+1)/(Rgas*gammah(k+1)*Dph(k+1)*Dpk(k))  ...
           - P(k)/(Rgas*gammah(k)*Dph(k)*Dpk(k));
  A(k,k+1) = P(k+1)/(Rgas*gammah(k+1)*Dph(k+1)*Dpk(k));
end
% Assign first, last equation coefficients for boundary conditions
   A(1,1) = - P(2)/(Rgas*gammah(2)*Dph(2)*Dpk(1));
   A(1,2) =   P(2)/(Rgas*gammah(2)*Dph(2)*Dpk(1));
   A(KK,KK) =  -P(KK)/(Rgas*gammah(KK)*Dph(KK))*Dpk(KK) ...
               -P(KK+1)/(Rgas*Ts*Dpk(KK)) ...
               *(1.0 + (P(KK+1)-Pk(KK))/Dph(KK));
   A(KK,KK-1) =  P(KK)/(Rgas*gammah(KK)*Dph(KK))*Dpk(KK) ...
               + P(KK+1)/(Rgas*Ts*Dpk(KK)) ...
               *((P(KK+1)-Pk(KK))/Dph(KK));
%
% Solve the eigenvalue problem using MATLAB dense
%    matrix routine, eig
%
[Z0,Lambda] = eig(A);
 Z   = zeros(size(Z0));
 H   = zeros(size(zk));
for k = 1:KK
   H(k) = -1.0/(gravty*Lambda(k,k));
end
% Sort, reorder and Pack the height eigenvalues
%  so that returned Z has eignvectors in descending order
[H0,indx] = sort(H);
% Convert to (m) and scaling so first mode = 10km
unitsconversion = H0(KK-1)/10.e3;
for k = 1:KK
   Z(:,KK-k+1) = Z0(:,indx(k));
   H(KK-k+1)   = H0(k)/unitsconversion;
end
```

Figure 5.4. *Define the discretized problem and solve for the eigenvalues using MATLAB. Return the modes, Z, and normalized eigenvalues H.*

Chapter 6

Rosenbrock Stiff ODE Methods for Chemical Reactions

The equation for chemical species conservation and reaction in the atmosphere takes the form

$$\frac{Dc}{Dt} = R(t, c). \tag{6.1}$$

The variable c expresses a concentration or mixing ratio. The material derivative expresses that the chemical reaction occurs in a parcel of air moving with the flow and that if no reaction occurred, the chemical species would simply be advected with no change of concentrations. Within this air parcel, a chemical reaction takes place that can be modeled with ordinary differential equations (ODEs). Choosing an individual parcel, let $y(t) = \int_{A(t)} c \, da$, and write the ODE system as

$$y' = F(t, y) \text{ with initial conditions } y(0) = y_0. \tag{6.2}$$

This is the common notation for writing an ODE, and we will use it in this section to avoid confusion and simplify notation for the numerical methods. The exercise will be to code the methods and perform some analysis of the stability and asymptotic properties of a simple chemical reaction test case.

6.1 ▪ The Runge–Kutta–Rosenbrock Method

Runge–Kutta methods fall into two categories—explicit and implicit. Chemical reactions are almost always stiff in the sense that they involve multiple and disparate time-scales. The use of an explicit method for this type of system would require restricting the time-step to the length of the smallest time-scale, a prospect that is not appealing for the long time integrations of climate problems. The Rosenbrock methods are an instance of the Runge–Kutta implicit methods.

Runge–Kutta methods can all be expressed using a method tableau (also called the Butcher tableau)

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array}$$

where $\mathbf{b}$ is the quadrature weights and $\mathbf{c}$ indicates intermediate positions within the time step. The matrix A is of size $s \times s$, where s is the number of stages in the Runge–Kutta method. A step of the method can be described from the tableau through the following

equations to advance from an initial condition at time t_0, to the time t_1 and time-step size $h = t_1 - t_0$.

$$y_1 = y_0 + h\mathbf{b}^T\mathbf{k}, \tag{6.3}$$
$$k_i = F(t_0 + hc_i, y_0 + hA\mathbf{k}). \tag{6.4}$$

The stability function for the Runge–Kutta method (with $z = h\alpha$ analyzing $y' = \alpha y$) is

$$R(z) = 1 + z\mathbf{b}^T(I - zA)^{-1}. \tag{6.5}$$

The two stage, second order Rosenbrock scheme (ROS2) for an autonomous system[10] can be written [24]

$$\big(I - h\gamma F'(y_0)\big)k_1 = F(y_0), \tag{6.6}$$
$$\big(I - h\gamma F'(y_0)\big)k_2 = F(y_0 + hk_1) - 2k_1, \tag{6.7}$$
$$y_1 = y_o + \frac{3}{2}hk_1 + \frac{1}{2}hk_2, \tag{6.8}$$

where $\gamma = 1 + \frac{1}{\sqrt{2}}$ is chosen so that the method is L-stable. The stability function with $z = h\alpha$ for the problem $y' = \alpha y$ is

$$R(z) = \frac{1 + (2 - 2\gamma)z + (\frac{1}{2} - 2\gamma + \gamma^2)z^2}{(1 - \gamma z)^2}. \tag{6.9}$$

Exercise 6.1.1. *Using the ROS2 method, solve the test problem*

$$y' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} y. \tag{6.10}$$

With initial conditions $y_1(0) = 0$ and $y_2(0) = 1.0$, the solution is $y_1(t) = \sin t$ and $y_2(t) = \cos t$. Integrate to $t = 20$ with $\Delta t = 0.4, 0.2, 0.1$, and 0.05 to consider the accuracy of the method. (Figure 6.1 plots y_1 versus y_2 for each of these values.) Plot the stability region of the complex plane for which $|R(z)| \leq 1$. Relate this to A-stability and L-stability.

Butcher [2] derives the condition for a Runge–Kutta method to be symplectic: $M = diag(\mathbf{b})A + A^T diag(\mathbf{b}) - \mathbf{b}\mathbf{b}^T = 0$. Is ROS2 symplectic?

6.2 ▪ Chemical Reaction Problem

The appearance of the Jacobian of F in the implicit system shows the major drawback for implicit systems: how do we calculate or approximate them? And then the question becomes how we can most efficiently and accurately solve the implicit system. The Rosenbrock scheme has the advantage of requiring only one system to be formed at each timestep, though it will be solved twice for different k's. We also note that k_1 is a first order approximation to the solution and can be used to estimate the error over the step as $\epsilon = y_1 - k_1$.

An example of an atmospheric reaction is the photochemistry of chlorine in the stratosphere. Photolysis breaks apart Cl_2 molecules high in the atmosphere and is a relatively

[10] In an autonomous system the right-hand side function depends only on the solution, i.e., $y' = f(y)$.

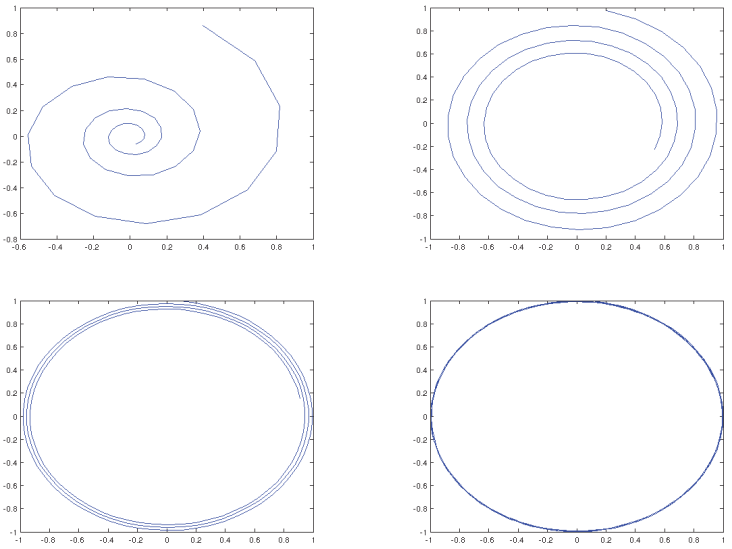


Figure 6.1. *Rosenbrock method ROS2 integrating a test problem with progressively finer step size,* $\Delta t = 0.4, 0.2, 0.1,$ *and* 0.05. *The exact solution should track the unit circle.*

slow reaction, but then these individual atoms recombine very quickly. To see how the Rosenbrock method applies, consider the toy chemical reaction with a fast, a very fast, and a slow process involving three constituents, A, B, and C. (This example was given by Willoughby in 1974 [10].)

The reactions are diagrammed by the mechanism

$$A \to B \text{ (slow)}, \tag{6.11}$$

$$B + B \to B + C \text{ (very fast)}, \tag{6.12}$$

$$B + C \to A + C \text{ (fast)}. \tag{6.13}$$

The nonlinear, stiff ODE system is defined by

$$y_1' = -0.04 y_1 + 10^4 y_2 y_3, \tag{6.14}$$

$$y_2' = 0.04 y_1 - 10^4 y_2 y_3 - (3 \times 10^7) y_2^2, \tag{6.15}$$

$$y_3' = +(3 \times 10^7) y_2^2. \tag{6.16}$$

The initial conditions at $t = 0$ are $y_1 = 1$, $y_2 = y_3 = 0$.

Exercise 6.2.1. *Use MATLAB's ROS2 integrator to solve this system and examine the conservation properties and accuracy.*

6.3 ▪ Strang Splitting in ROS2

An ODE scheme can be applied in more complex situations where the right-hand side source term can be split into different processes. It is often advantageous (if not accurate)

to treat each process separately. For example, atmospheric chemistry must model transport, reaction, and diffusion. The ROS2 scheme can be modified to split the source term as [24]

$$F(y) = F_A(y) + F_D(y) + F_R(y), \tag{6.17}$$

representing advection, diffusion, and reaction, respectively. The Strang splitting divides the step into three substeps, and updates of the transport, $F_T = F_A + F_D$, are treated with the explicit trapezoidal rule while the reaction term is treated with ROS2. The scheme is given by

$$\big(I - h\gamma F_R'(Y_1)\big)k_1 = F_R(Y_1), \tag{6.18}$$
$$\big(I - h\gamma F_R'(Y_1)\big)k_2 = F_R(Y_1 + hk_1) - 2k_1, \tag{6.19}$$
$$\tag{6.20}$$

together with the other substeps

$$Y_0 = y_0, \tag{6.21}$$
$$Y_1 = Y_0 + \frac{1}{4}hF_T(Y_0) + \frac{1}{4}hF_T\left(Y_0 + \frac{1}{2}hF_T(Y_0)\right), \tag{6.22}$$
$$Y_2 = Y_1 + \frac{3}{2}hk_1 + \frac{1}{2}hk_2, \tag{6.23}$$
$$Y_3 = Y_2 + \frac{1}{4}hF_T(Y_2) + \frac{1}{4}hF_T\left(Y_2 + \frac{1}{2}hF_T(Y_2)\right), \tag{6.24}$$
$$y_1 = Y_3. \tag{6.25}$$

This splitting is second order accurate in time. Note that the trapezoidal rule is A-stable but not L-stable. Many of the other source splittings used in climate modeling are only first order.[11]

6.4 ▪ Transport-Reaction Problem

In this section we explore coupling the stratospheric chlorine photochemistry problem with two dimensional horizontal transport and vertical transport and diffusion. The flow itself will be prescribed as in [12]. Using a zonal, meridional, and hybrid vertical coordinate η, the velocities are time and space varying according to

$$u(\lambda,\theta,t) = \varkappa \sin^2(\lambda)\sin(2\theta)\cos\left(\pi\frac{t}{T}\right) + 2\pi\frac{\cos(\theta)}{T}, \tag{6.26}$$
$$v(\lambda,\theta,t) = \varkappa \sin(2\lambda)\cos(\theta)\cos\left(\pi\frac{t}{T}\right), \tag{6.27}$$
$$\dot{\eta}(\lambda,\theta,t) = \frac{\omega_0}{p_0}\cos\left(2\pi\frac{t}{\tau}\right)\sin\left(s(\eta)\frac{\pi}{2}\right), \tag{6.28}$$

[11] The Strang splitting is based on a factorization of the matrix exponential as a solution to the linear, split problem $y' = L_1 y + L_2 y$ which has solution $y(t) = e^{t(L_1+L_2)}$. Strang noticed that $e^{tL_1}e^{tL_2} = e^{t(L_1+L_2)} + \frac{1}{2}t^2(L_1L_2 - L_2L_1) + O(t^3)$, so the error is second order unless the operators commute, i.e., $L_1L_2 = L_2L_1$. However, there is another factorization that avoids this problem: $e^{\frac{t}{2}L_1}e^{tL_2}e^{\frac{t}{2}L_1} = e^{t(L_1+L_2)} + O(t^3)$. See [16] for more details.

where τ is four days, the reference pressure $p_0 = 1000\text{hPa}$, the maximum pressure velocity is $\omega_0 = \frac{4\pi \times 10^4 \text{Pa}}{\tau}$, and the vertical shape function is given by

$$s(\eta) = \min\left[1, 2\sqrt{\sin\left(\frac{\eta - \eta_{top}}{1 - \eta_{top}}\pi\right)}\right]. \tag{6.29}$$

Exercise 6.4.1. *To advect the 3D chemical species use a semi-Lagrangian horizontal transport scheme together with a hybrid vertical coordinate based on the vertical module. Use the ROS2 scheme with Strang splitting to update the reaction, vertical transport, and diffusion. Integrate for $T = 14$ days using the prescribed velocities and chemical tracer initial conditions. Explore the diffusion and mixing properties of the scheme using diagnostics from [13].*

Chapter 7

Bayesian Uncertainty Quantification

This chapter is contributed by Evan Kodra and Mellisa Allen,[12] who reported on using Bayesian statistical methods as a particular approach to uncertainty quantification in numerical modeling. The problem was to estimate the uncertainty in the source of a species subjected to advection and diffusion in a simple one dimensional periodic domain. The write-up and their contributed MATLAB code implementing the methods and used for their project are largely unaltered. It is an interesting project with larger implications that shows how MATLAB programming can be used to approach advanced topics. Some background on uncertainty quantification is given in Section 5.8 of *Climate Modeling for Scientists and Engineers*.

7.1 ▪ Characterizing Uncertainty in a Source

We use Bayesian methods to build a distribution characterizing uncertainty in the source emission. In this study, we have the advantage of knowing the "true" emission, and thus we can test performance of multiple variations of the Bayesian approach, illustrating various issues in its effective implementation.

The target distribution will be the last 30 time steps out of 110. That is, we are comparing an "observed" climate of 30 time periods to a "simulated" climate of 30 time periods. Each of the 30 grids is a 6-by-6 matrix; these are converted to 36-length vectors for a total of a 1080-length vector, denoted by $\mathbf{Y}_{True}$. $\mathbf{Y}_{True}$ is obtained from running the control volume model forward with the true source emission.

In each simulation, starting with some initial guess for the emission, E_g, we use the Metropolis (M) (or Metropolis–Hastings (MH) in one case [11]) algorithm to simulate the posterior distribution. We use the multivariate normal distribution to characterize $\mathbf{Y}$, with covariance matrix $\sigma 2\mathbf{I}$, proportional to the following:

$$P(\mathbf{Y}|E) \propto \exp\left(-\frac{1}{2\sigma^2}(\mathbf{Y}_{True}-\mathbf{Y}(E))^T(\mathbf{Y}_{True}-\mathbf{Y}(E))\right), \tag{7.1}$$

where in practice we allow $\sigma^2 = 1$, assuming that all errors $\mathbf{Y}_{i,True}-\mathbf{Y}_{i,E}$ are independent and follow a normal distribution with mean 0 and variance 1, as in [9]. We assume a uniform prior distribution over the real line for E, which is effectively 1 and will not

[12] The team of Kodra and Allen developed this project as part of an Advanced Methods for Climate Modeling course offered in the spring semester of 2011 at the University of Tennessee.

appear in the ratio r. The ratio r (31, 32) is then computed as

$$r = \frac{\exp\left(-\frac{1}{2\sigma^2}(\mathbf{Y}_{True} - \mathbf{Y}(E^*))^T(\mathbf{Y}_{True} - \mathbf{Y}(E^*))\right)}{\exp\left(-\frac{1}{2\sigma^2}(\mathbf{Y}_{True} - \mathbf{Y}(E^s))^T(\mathbf{Y}_{True} - \mathbf{Y}(E^s))\right)}. \tag{7.2}$$

The posterior is estimated as follows:

- E_g is chosen as the best guess at the true E. We assume in some cases better prior knowledge of E than in others to demonstrate the value of informed prior knowledge.
- A proposal distribution for E is chosen: either uniform, normal, or Gamma together with a spread parameter.
- The M algorithm runs for 10,000 iterations, as described earlier.
- Density plots are constructed, and 95% quantile-based intervals, as well as medians, are obtained from the 10,000 iterations. We report medians as point estimates of E.

7.2 ▪ UQ Results

Table 7.1 summarizes the various settings and outcomes for each simulation. In each simulation, *the true emission is arbitrarily set to 38.71.* Although the chosen settings (columns 2–4) may seem a bit arbitrary, they are chosen to illustrate some of the issues related to Bayesian simulation, in the context of the inverse problem, discussed earlier.

Table 7.1.

Sim	Initial guess	Proposal dist.	Spread	Median	0.025th percentile	0.975th percentile	Acceptance ratio
1	25	uniform	(E(s)-2,E(s)+2)	38.72	37.76	39.62	0.3578
2	55	uniform	(E(s)-2,E(s)+2)	38.72	37.84	39.63	0.3605
3	25	uniform	(E(s)-0.1,E(s)+0.1)	38.59	31.08	39.57	0.9353
4	55	uniform	(E(s)-0.1,E(s)+0.1)	38.89	37.97	49.17	0.9293
5	25	normal	variance=1	38.70	37.82	39.60	0.4686
6	55	normal	variance=1	38.73	37.81	39.63	0.4713
7	25	normal	variance=25	38.71	37.79	39.66	0.1126
8	55	normal	variance=25	38.70	37.83	39.60	0.1152

Eight simulations, each producing 10,000 values, were run: four using a uniform distribution for generation of proposal values and four using a normal distribution. The first simulation shows the results of a uniform distribution for the proposed value with an initial guess of 25 and a spread of "previous proposed value" ±2. The resulting median simulated value of 38.72 comes very close to the true value of 38.71, while the values in the confidence range are within 0.1 and 0.4, respectively, of the true value. For the second simulation, the initial proposed value was slightly farther from the true value at 55, although the spread values remained the same as those in the first simulation. The resulting median of the simulation matches that of the first simulation, but the values within the confidence interval are closer to the true value.

Does this result indicate that a simulation using any initial proposed value will converge? In theory, it should; however, a simulation with 100 as the initial proposed value was attempted. The result shows the solution flat-lining at the value of the initial guess.

This outcome is due to machine (and/or MATLAB) rounding error. Since the values in both the numerator and the denominator of r in this case are so small, they are rounded to zero, creating a value of "infinity" that the program cannot resolve. Therefore, if the initial proposed value produces this type of result, the initial proposed value should be adjusted. Practically speaking, if the simulation appears to "flat-line" at any point, then this may indicate a poor choice of an initial guess value. In some cases, this may serve as a nice tool for deducing the neighborhood of the true initial condition.

7.3 ▪ Discussion of the Approach

In the context of a simple control volume model, we have demonstrated several simple aspects of Bayesian statistics and Markov chain Monte Carlo (MCMC) simulation, notably including the M algorithm and several of its considerations. We have shown that these methods can indeed quantify uncertainty in an initial condition like that chosen in this work.

Although all of the simulations attempted produced reasonable outputs, application of this process is at present limited to single source emission in a relatively simple system. Estimation of a vector of initial emissions, of continuous emissions, or of multiple initial condition variables is beyond the scope of this experiment. However, these more complex issues may be of interest to both a statistician and a climate modeler. Evaluations of such multivariate parameters will require adjustment to both the formulation of the advection-diffusion model and to its statistical evaluation.

Beyond simply estimating uncertainty in more complex, multivariate initial conditions, these statistical methods will ideally be employed in some form on a much larger scale and more complex simulations, such as global or regional climate models. Keys to such implementations may include model reduction schemes and/or computational parallelism of analysis. Parallel computing will undoubtedly be a useful tool, as the M and MH algorithms are naturally built and would actually benefit from such computing.

In addition, the choice of evaluation statistics (whether the model is replicating observed) may be much more complex in such an experiment. For example, when estimating uncertainty in an initial condition vector of a global climate model, what set of variables should be compared to observations (as in (7.1) and (7.2))? Furthermore, at what spatial and temporal resolutions should they be compared? If we arrive at a distribution of initial conditions using observed versus modeled temperatures in an M algorithm, for example, how does that ensure that using precipitation instead would not yield a completely different initial condition distribution? Must we consider, then, joint performance on both temperature and precipitation? These are some questions, undoubtedly among many others, that one might encounter when adapting methods proposed here to real-world climate models.

In this work, we actually used the control volume model to compute a synthetic observed time series of grids. This actually assumes that if we choose the correct initial condition, we could replicate observations exactly. Of course, in real applications, observations will not come from such a model, and hence this will not be possible; thus, in estimating uncertainty in initial conditions, there will be additional uncertainty resulting from any model failure of parametrizing or discretizing some physical process. Conceptually, the Bayesian framework should capture this uncertainty, as it lets the data guide the construction of the uncertainty in initial conditions.

Further opportunities exist in this statistical process for the optimization and uncertainty quantification of boundary conditions from global models used in downscaling to regional models. This is something that could initially be explored in a manner similar to this work.

Bibliography

[1] M.I. Budyko. The effect of solar radiation variations on the climate of the Earth. *Tellus*, 21(5):611–619, 1969. (Cited on p. 16)

[2] J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*. 2nd ed. John Wiley and Sons, New York, 2008. (Cited on p. 39)

[3] M.-D. Chou, L. Peng, and A. Arking. Climate studies with a multi-layer energy balance model. Part II. The role of feedback mechanisms in the CO2 problem. *J. Atmos. Sci.*, 39:2657–2666, 1982. (Cited on p. 16)

[4] R Daley. *Atmospheric Data Analysis*. Cambridge University Press, Cambridge, UK, 1991. (Cited on p. 33)

[5] R.E. Dickinson. Climate sensitivity. In *Issues in Atmospheric and Oceanic Modeling, Part A*, S. Manabe, ed. Advances in Geophysics 28. Academic Press, New York, 1985. (Cited on p. 13)

[6] J. B. Drake. *Climate Modeling for Scientists and Engineers*. SIAM, Philadelphia, 2014. (Cited on p. 30)

[7] J. B. Drake. *Supplemental Lectures* on Climate Modeling for Scientists and Engineers. SIAM, Philadelphia, http://www.siam.org/books/MM19, 2014. (Cited on p. 33)

[8] J.B. Drake, P. Worley, and E. DÁzevedo. Algorithm 888: Spherical harmonic transform algorithms. *ACM Trans. Math. Software*, 35(3):23, 2008. (Cited on p. 30)

[9] D. Galbally, K. Fidkowski, K. Willcox, and O. Ghattas. Non-linear model reduction for uncertainty quantification in large-scale inverse problems. *Int. J. Numer. Meth. Engrg.*, 81:1581–1608, 2010. (Cited on p. 43)

[10] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 2nd ed. Springer, New York, 2002. (Cited on p. 40)

[11] W.K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 2005. (Cited on p. 43)

[12] C. Jablonowski, P.H. Lauritzen, M.A. Taylor and R.D. Nair. Idealized test cases for the dynamical cores of Atmospheric General Circulation Models: A proposal for the NCAR ASP 2008 summer colloquium. (http://www.cgd.ucar.edu/cms/pel/asp2008/idealized_testcases.pdf), 2008. (Cited on p. 41)

[13] P.H. Lauritzen and J. Thuburn. Evaluating advection interrelated tracers, transport schemes using scatter plots and numerical mixing diagnostics. *Quart. J. Roy. Meteor. Soc.*, 138:906–918, 2012. (Cited on p. 42)

[14] S.K. Lele. Compact finite difference schemes with spectral-like resolution. *J. Comp. Phys.*, 103:16–42, 1992. (Cited on p. 35)

[15] L.M. Leslie and R.J. Purser. A comparative study of the performance of various vertical discretization schemes. *Meteorol. Atmos. Phys.*, 50:61–73, 1992. (Cited on p. 35)

[16] R.J. LeVeque. *Numerical Methods for Conservation Laws*. Birkhäuser, Basel, 1990. (Cited on p. 41)

[17] M.S. Lian and R.D. Cess. Energy balance climate models: A reappraisal of ice-albedo feedback, *J. Atmos. Sci.*, 34:1058–1062, 1977. (Cited on p. 16)

[18] K. Puri and A. Kasahara. Spectral representation of three-dimensional global data by expansion in normal mode functions. *Mon. Wea. Rev.*, 109, 1981. (Cited on p. 33)

[19] V. Ramanathan et al. Cloud-radiative forcing and climate: Results from the earth radiation budget experiment. *Science*, 243(4887):57–63, 1989. (Cited on p. 16)

[20] A. Robok. The dust cloud of the century, *Nature*, 301:373–374, 1983. (Cited on p. 16)

[21] W.D. Sellers. A global climatic model based on the energy balance of the Earth-Atmosphere system. *J. Appl. Meteor.*, 8:392–400, 1969. (Cited on p. 16)

[22] S. Solomon, D. Qin, M. Manning, Z. Chen, M. Marquis, K. B. Averyt, M. Tignor, and H. L. Miller, eds. *IPPC, 2007: Climate Change 2007: The Physical Science Basis. Contributions of the Working Group I to the Fourth Assessment Report of the Intergovernment Panel on Climate Change.* Cambridge University Press, Cambridge, UK, 2007. (Cited on p. 17)

[23] B. Stroustrup. *The C++ Programming Language*. 2nd ed. Addison–Wesley, Reading, MA, 1991. (Cited on p. 9)

[24] J.G. Verwer, E.J. Spee, J.G. Blom, and W. Hunsdorfer. A second-order Rosenbrock method applied to photochemical dispersion problems. *SIAM J. Sci. Comput.*, 20(4):1456–1480, 1999. (Cited on pp. 39, 41)