CHAPTER 3

DIGITAL SIMULATION

3.1. INTRODUCTION

Simulation of continuous systems started with the use of analog computers. Analog computers had been used widely, but they had several disadvantages. Time and magnitude scaling were cumbersome. Inaccuracies were caused by analog systems, which cannot separate signal change from noise in principle. There were also frequent breakdowns in hardware components. With the development of digital computers in the late 1960's and the growth in capacity of mainframes, digital simulation became the predominant technique for continuous simulation. In the 1980's, digital simulation spread to mini- and microcomputers. The high speed and large memory size of microcomputers have enabled us to use almost the same simulation languages that once were only available on mainframes.

The concept of digital simulation is the same with that used in analog computers. It is suitable for student lab work since PCs are now available in microcomputer labs in many schools.

3.2. SYSTEM DYNAMICS

The approach to simulating continuous systems, which is called System Dynamics has been well-known since it was used to simulate what would happen on the earth. The results of the simulation were published as "World Dynamics" by J. W. Forrester (1971) and later as "The Limits to Growth" by his successors such as D. H. Meadows (1972). The approach itself was developed originally by Forrester, and the simulation language **DYNAMO** was invented by him. It can be said that **DYNAMO** opened the era of digital simulation. The “World Dynamics” model of population expansion, energy resource expenditure, and pollution increase on the earth was so popular that its rather pessimistic predictions astonished public administrators in many countries in the 1970's. This kind of model cannot predict an event such as an energy crisis, which is caused politically and is discontinuous; when a crisis occurred, the rate of consumption of energy resources decreased, and the whole situation was then reconsidered and modified.

The language **DYNAMO** is more or less in the form of difference equations and has not been well used because more friendly languages such as **CSMP** were developed soon after. System dynamics is defined as a methodology to analyze system behaviour including feedback loops. The main applications are for analyses of social, biological, and ecological processes with many nested feedback loops and of non-linear systems.

3.3. SIMULATION LANGUAGES

Languages are classified into two groups: those for continuous systems and those for discrete systems. Languages such as **GPSS** and **SIMAN** are mainly for discrete systems; recent versions can handle continuous simulation too (Pegden, 1986), although all programs for continuous systems are more or less of the nature of **FORTRAN** subroutines. On the other hand, a continuous system simulation language such as **ACSL** can handle discrete simulation. Therefore, the two groups are getting closer, although there are still large discrepancies between the two. We are mostly interested in simulation of continuous systems, and it will be emphasized here.

Digital simulation languages have been developed to capitalize on the recent developments in simulation of continuous systems and the advantages of digital computers. They free us from the disadvantages of both time and magnitude scaling which were needed on analog computers. They take advantage of the higher accuracy of each component more than analog computers, and place no actual limitation on the number of functions used. Popular languages were **DYNAMO**, **MIMIC**, **DSL/90**, **CSMP** and **CSSL**.

The languages were then separated into two groups after transient languages such as **MIMIC** were discounted. One was the group represented by **CSMP** (Continuous System Modeling Program), developed by IBM, and the other was **CSSL**, which was mostly popular in Europe. The basic types of languages have similar capabilities, but with some differences in expressions. In biological research, it can be said that **CSMP** was still more popular than the other languages available on mainframes. PC versions of this kind of simulation language have been developed: for example, **micro-CSMP** (**CSMP** version for PCs and compatibles), **PCSMP** (similar to **micro-CSMP** with some hardware restriction), **ACSL** (Advanced Continuous System Language, an advanced version of **CSSL**) has much more flexibility but needs more typing in debugging processes and has hardware protection), and **SYSL** (80 - 90% compatibility with **CSMP**). **Micro-CSMP** was chosen as the language in the first edition of this book.

Since then, several new languages have been developed, such as **SIMNON**, **DYMOLA** and **Stella**. **SIMNON** is very similar to **ACSL** and **CSMP**. On the other hand, **DYMOLA** and **Stella** are called modelling languages and are more or less object-oriented languages (see Cellier, 1991). They have functions to make a model and can cooperate with other simulation languages such as **ACSL**, **SIMNON**, **MATLAB** (**SIMULINK**) and **FORTRAN**. **Stella** and **DYMOLA** introduce a relatively large number of functions for particular purposes and their functions might not be needed for the models in this book. Inputs and outputs are connected with special symbols. Flow diagrams are the main part of the program instead of description by equations. An ecological modelling group in the Netherlands that first developed **PCSMP** also developed **FST**(Fortran Simulation Translator). **FST** is more or less the same as **PCSMP** but is much more restricted by the tight grammar of **FORTRAN,** such as no mixture of integer and real numbers, than **PCSMP**. For example, **TO** can not be a variable since it is a part of the **GO TO** statement of **FORTRAN**.

A group of mathematical software has been developed including **Mathematica**, **MATLAB**, **Mathcad**, and **Maple**. These types of the software were originally developed to find formulas and perform mathematical calculations such as on matrices. **MATLAB** is linked with **SIMULINK,** which is a symbolic solver similar to analog computer techniques and **Stella** (see Bennett, 1995).

Since the operating system of PCs has changed from DOS to Windows and a Windows version of **micro-CSMP** or **PCSMP** does not exist, a new Windows-based simulation language is required to accompany the models developed in this book. **MATLAB** (**SIMULINK)**, product of MathWorks Inc., USA, is one of the most successful software packages currently available and is widely used for mathematical calculations. It is suited for work in control and in simulation as well. It is a powerful, comprehensive and user-friendly software package for performing mathematical computations. Equally as impressive are its plotting capabilities for displaying information. In addition to the core package, referred to as **MATLAB**, there are 'toolboxes' for several application areas. However, only the core package is required to solve the models of this book with one example in **SIMULINK**, one of the toolboxes.

The concept of any of these languages is based on the analog computer; that is the same. Once you become familiar with one simulation language, it is much easier to understand one of the others than to go from **FORTRAN** to **BASIC**.

It is clear that language is a tool, and you can make a model in a common language such as **FORTRAN** or **BASIC**. However, it is much easier to make a model in a simulation language and to understand a model someone else has developed than in a programming language, because a model in a simulation language is more like a set of mathematical equations than a list of programs.

3.4. DIGITAL SIMULATION BY CSMP AND MATLAB

3.4.1. Concept of the analog computer

The concept of analog computers is alive in digital simulation. One of the most important functions on analog computers is integration. Its symbol, shown in Fig. 3.1, is well-known to all engineering students. This figure exactly corresponds to the hardware configuration on analog computers. If $d\mathbf{Y}/d\mathbf{t}$ (derivative of an arbitrary function **y** due to time **t** is supplied into the operational amplifier whose function is integration, the original function **Y** can be obtained through an integrator which is shown by the integral sign. The minus sign is due to the hardware configuration. If input and output wires of the amplifier are connected together after the sign is converted, **exp**(**t**) is obtained as the output **y**. On analog computers you need not solve the equation. It can be said that the computer solves the differential equation you supply. In **SIMULINK**, the integrator is simply replaced by the function named 1/S, and without an inverter for sign change the same output is obtained.

Using digital simulation languages such as **CSMP**, the process is similar; that is, for example, in **CSMP** a function to integrate according to time (**INTGRL**) is available with other functions.

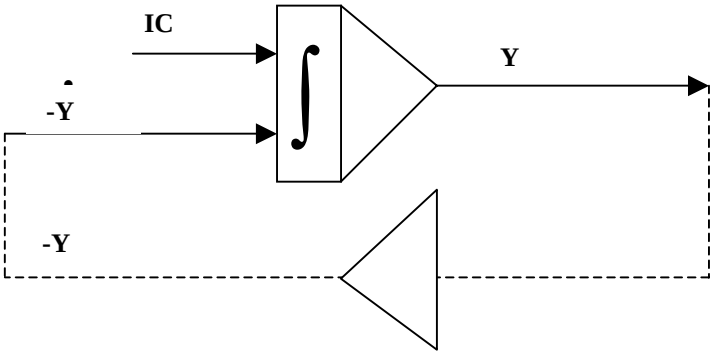


Figure 3.1. Diagram of integration on analog computers.

3.4.2. Comparison of CSMP and MATLAB programs with mathematical equations

Simulation of either a linear or a non-linear system, which is time-dependent -- in other words, dynamic and continuous -- can be done using analog computers. Simulation using analog computers has the advantage of being equation-oriented. Analog computers have many functions, such as integration, arbitrary function generation, and implicit expression.

Plant growth is a good example of non-linear and non-steady-state systems. In general, plant growth that is continuously changing can be expressed as a system of differential equations,

$$d\mathbf{Y1}(\mathbf{t})/d\mathbf{t} = \mathbf{f1}\ (\ \mathbf{Y1}(\mathbf{t}), \mathbf{Y2}(\mathbf{t}), \ldots, \mathbf{Yn}(\mathbf{t}), \mathbf{E1}(\mathbf{t}), \mathbf{E2}(\mathbf{t}), \ldots, \mathbf{Em}(\mathbf{t})\ )$$
$$d\mathbf{Y2}(\mathbf{t})/d\mathbf{t} = \mathbf{f2}\ (\ \mathbf{Y1}(\mathbf{t}), \mathbf{Y2}(\mathbf{t}), \ldots, \mathbf{Yn}(\mathbf{t}), \mathbf{E1}(\mathbf{t}), \mathbf{E2}(\mathbf{t}), \ldots, \mathbf{Em}(\mathbf{t})\ )$$
$$\vdots$$
$$d\mathbf{Yn}(\mathbf{t})/d\mathbf{t} = \mathbf{fn}\ (\ \mathbf{Y1}(\mathbf{t}), \mathbf{Y2}(\mathbf{t}), \ldots, \mathbf{Yn}(\mathbf{t}), \mathbf{E1}(\mathbf{t}), \mathbf{E2}(\mathbf{t}), \ldots, \mathbf{Em}(\mathbf{t})\ )$$

where **Y1** to **Yn** are **n** state variables such as photosynthesis, and leaf and root weights, **E1** to **Em** are **m** environmental or boundary conditions such as solar radiation, temperature and CO_2 concentration, and **f1** to **fn** are functional relations which describe the rate of change of each state variable. With **n** unknown variables **Y1** to **Yn**, and **n** differential equations, unique solutions can be derived numerically.

These equations are expressed in an integral form in **CSMP** as

$$\mathbf{Y1}(\mathbf{T})=\mathbf{INTGRL}(\mathbf{IY1}, \mathbf{f1}(\mathbf{Y1}(\mathbf{T}),\mathbf{Y2}(\mathbf{T}),..,\mathbf{Yn}(\mathbf{T}),\mathbf{E1}(\mathbf{T}),\mathbf{E2}(\mathbf{T}),\ldots,\mathbf{Em}(\mathbf{T})))$$
$$\mathbf{Y2}(\mathbf{T})=\mathbf{INTGRL}(\mathbf{IY2}, \mathbf{f2}(\mathbf{Y1}(\mathbf{T}),\mathbf{Y2}(\mathbf{T}),..,\mathbf{Yn}(\mathbf{T}),\mathbf{E1}(\mathbf{T}),\mathbf{E2}(\mathbf{T}),\ldots,\mathbf{Em}(\mathbf{T})))$$
$$\vdots$$
$$\mathbf{Yn}(\mathbf{T})=\mathbf{INTGRL}(\mathbf{IYn},\mathbf{fn}(\mathbf{Y1}(\mathbf{T}),\mathbf{Y2}(\mathbf{T}),..,\mathbf{Yn}(\mathbf{T}),\mathbf{E1}(\mathbf{T}),\mathbf{E2}(\mathbf{T}), \ldots ,\mathbf{Em}(\mathbf{T})))$$

where **IY1**, **IY2**, . . . , **IYn** are initial conditions for **Y1**, **Y2**, . . . , **Yn**, respectively, and we still have **n** equations for **n** unknown variables.

Let's consider more details using a concrete expression. Suppose we have a mathematical expression as follows:

$$d^2\mathbf{x}/d\mathbf{t}^2 = \mathbf{F} - \mathbf{A} * d\mathbf{x}/d\mathbf{t} - \mathbf{B} * \mathbf{x}$$

where initial conditions are $\mathbf{x}(0) = \mathbf{X0}$, $d\mathbf{x}(0)/d\mathbf{t} = \mathbf{DX0}$. Then a complete CSMP program for this is,

```
X = INTGRL (X0, DX)
DX = INTGRL (DX0, F - A * DX - B * X)
TIMER  FINTIM = 10.0, OUTDEL = 0.5, DELT = 0.1
PRTPLOT X
END
STOP
```

The main point is that the mathematical expression for plant growth is in the derivative form and the corresponding expression in **CSMP** is in integral form. The first two equations in the **CSMP** program are essential ones. It is not difficult to understand the correspondence between these two equations and the differential equation in the mathematical expression, if you notice that the **CSMP** program has the same variable **DX** ($d\mathbf{x}/d\mathbf{t}$ in the mathematical expression) in the first two equations and that the first terms in the parentheses of the function **INTGRL**(INTeGRaL) are initial conditions.

The mathematical expression can be easily divided into two differential equations:

$$\mathbf{x} = \int d\mathbf{x}/d\mathbf{t}\ (dt)$$

$$d\mathbf{x}/d\mathbf{t} = \int (\mathbf{F} - \mathbf{A} * d\mathbf{x}/d\mathbf{t} - \mathbf{B} * \mathbf{x})\ (d\mathbf{t})$$

TIMER in the **CSMP** program means time sets for numerical integration. **FINTIM** is the finish time, **OUTDEL** is the time step for the output and **DELT** is the time increment for integration. **PRTPLOT** (PrinT & PLOT) is a kind of output control, which gives a printout of numerical values as well as a printer plot. **TIME** is reserved as a system variable and is incremented by **DELT**. In **CSMP**, capital letters can only be used for expression, except in comments that start with an asterisk.

In **MATLAB**, two program files need to be generated to solve the above problem: A main program and an ODE function subprogram. A semicolon is required at the end of each command except for comments that start with a percent sign. As shown in Fig. 3.2, the dimensions of both **y0** and **dy** are 2 rows by 1 column. The dimensions of **y0** and **dy** need to be consistent.

```
% main.m
    T0=0; Tfinal=10;
    y0=[X0; DX0];
    [t,y]=ode23('subprg',[T0 Tfinal],y0);
    plot(t,y);
% subprg.m
    function dy=subprg(t,y)
    X=y(1);                          % define y1
    DX=y(2);                         % define y2
    DIFF_X = DX;                     % expression 1:  dy1/dt
    DIFF_DX=F - A * DX - B * X;      % expression 2:  dy2/dt
    dy=[ DIFF _X; DIFF _DX];         % dy=[expression 1; expression 2]
```

Figure 3.2. Structure of MATLAB programs for solving the differential equations.

The biggest advantage in **CSMP** or **MATLAB** programs is that the original equation need not be split into the form the computer can understand, as is always required in other common languages. Therefore, the program itself becomes very easily understandable for not only the programmer but also other people.

3.5. MODEL STRUCTURE AND REPRESENTATION

Models are classified into several categories. As we are concentrated into biological and environmental models, classification based on model structures should be noted. In order to understand how to make a model, the physical structure is the most important aspect to be considered. Thus, three main categories, lumped or distributed models, steady state or dynamic models and linear or non-linear models, are of primary concern. The **CSMP** and **MATLAB** languages are particularly powerful for solving dynamic non-linear system behaviour. To simplify the model, lumped models in which one variable is assigned to each object to express an average are very often used. For example, normally one variable is assigned to the inside air temperature of a greenhouse. However, if there is a large temperature gradient in one object, such as the soil layer, more than two variables can be assigned to the object even if the model is one-dimensional. The soil layer is divided into several layers, and in each soil layer temperatures are defined separately. This kind of model is called a distributed model. Air temperature in the greenhouse can also be divided into separate regions (see section 6.5).

3.5.1. Differential equation

Exponential growth can be expected if the relative growth rate of a living creature is constant. Let us consider population increase for humans. Assuming constant birth rate and no mortality, population increase rate is expressed as the product of the present population and the relative growth rate, that is,

$$d\mathbf{P}/d\mathbf{t} = \mathbf{BR} * \mathbf{P} \qquad (3.1)$$

where **P** is the present population and **BR** is the birth rate (births/unit time).

If **BR** is constant, eq. 3.1 is linear and is easily solved analytically. The solution is **P** = **A*exp** (**BR * t**), assuming the initial condition of **P** is **A**. This is programmed in the following manner by **CSMP**:

$$\mathbf{P} = \mathbf{INTGRL}(\mathbf{A}, \mathbf{BR} * \mathbf{P}) \qquad (3.2)$$

where **INTGRL** is one of the powerful functions of **CSMP** used to integrate the second argument in the parentheses in terms of time. The initial condition is placed as the first argument in the same parentheses. The expression can be left as implicit. The solution of eq. 3.1 is programmed in the following manner in **MATLAB**:

```
% main.m
[t,y]=ode23('F', TSPAN, A)
% F.m
function  dy=F(t, y)
DIFF_P=BR * y;              % define y1=y , expression 1: dy1/dt
dy=[DIFF_P];
```

where **TSPAN** = [**T0 TFINAL**] integrates the system of differential equations **y'** = **F**(**t**,**y**) from time **T0** to **TFINAL** with initial condition **A**. '**F**' is the name of an **ODE** file.

Numerical integration is carried out in the background of this expression, and several integration methods are available. Therefore, this one expression is equivalent to approximately 20 to 50 **FORTRAN** statements for numerical integration. Since all calculations are conducted numerically, the system to be simulated is not necessarily linear.

If we consider death by disease or other reasons, eq. 3.1 can be changed to

$$d\mathbf{P}/d\mathbf{t} = \mathbf{P} * (\mathbf{BR} - \mathbf{DR}) \qquad (3.3)$$

where **DR** is mortality. This equation means that a small population increase (*d***P**) in a small time increment (*d***t**) is equal to net increase, the difference between population inflow (**BR * P**) and outflow (**DR * P**).

Sometimes flow charts or diagrams which were originally used for **DYNAMO** are used for **CSMP** and **MATLAB**, as shown in Fig. 3.3. This figure is the diagram of eq. 3.3. The population level is express by a rectangular shape. **BR** and **DR** are flow rates, and straight lines show the flow of people. In this figure, it is assumed that people flow from a kind of source to a kind of sink, both of which are indicated with cloud-like symbols. Valves can change flow rates in an actual flow, such as water in pipelines.

The shape of the valve is used to calculate flow rates, **BR** and **DR** in the present case. This kind of flow diagram can be an aid to understanding the program, but sometimes it is cumbersome, while direct translation from mathematical equations is

straight- forward. In the present textbook, therefore, no flow charts or diagrams are used.

This concept of the relationship between flow and level can be applied to similar systems such as heat flow and mass flow in the air and in soil.

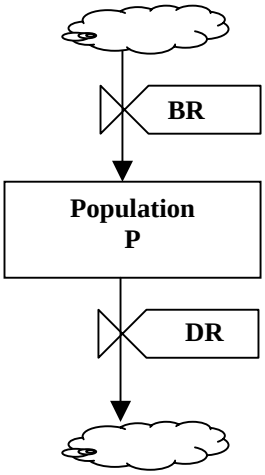


Figure 3.3. Flow diagram of eq. 3.3.

3.5.2. Description of systems

The important point in the description of systems is that the basic concept of the model is a flow of something: for example, it can be heat, water vapor or carbon dioxide in the air. Therefore, the governing rule is conservation of these components.

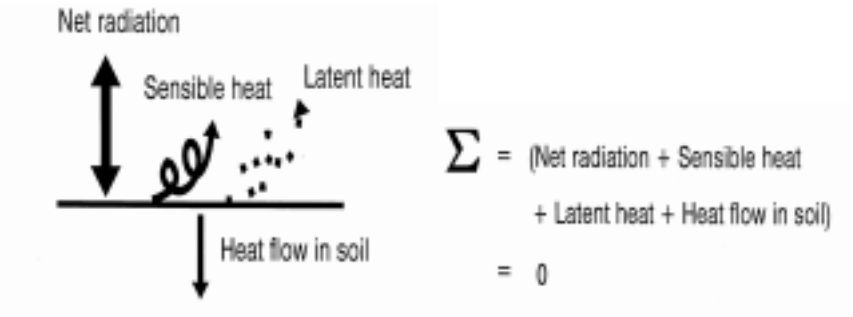
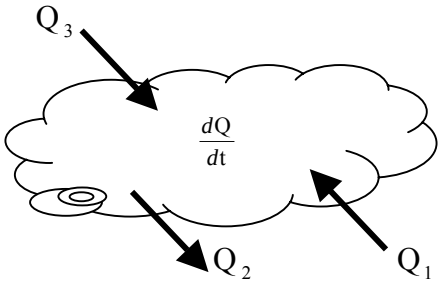


Figure 3.4. Energy balance equation.

In Fig. 3.4, energy balance of the soil surface is considered. We are assuming a hypothetical very thin film soil layer at the surface, which does not have appreciable volume to store energy in it. Since energy conservation holds here, the summation of all energy inflow and outflow is zero. In other words, net incoming energy is equal to net outgoing energy. This is the basic concept of how to build up an equation to describe a system. Fig. 3.5 shows how to build a differential equation. Let's assume there is a mass of air or water of which the volume is **V**, density is r, and volumetric heat capacity is $\mathbf{C_p}$. Energy $\mathbf{Q}_1$ and $\mathbf{Q}_3$ are coming in and $\mathbf{Q}_2$ is going out from the mass. Then, the energy change of this mass in *d***t** time is expressed as *d***Q**/*d***t**, and it is clearly equal to $\mathbf{Q}_1$ - $\mathbf{Q}_2$ + $\mathbf{Q}_3$ (the difference of inflow and outflow). This energy change can be easily converted to temperature change by introducing thermal properties of the mass as shown in the figure. This is a basic differential equation to show the temperature change of the mass.



$$\frac{dQ}{dt} = Q_1 - Q_2 + Q_3$$

$$= C_p \rho V \frac{d\theta}{dt} \ (= \text{inflow} - \text{outflow})$$

Figure 3.5. Differential equation to express an energy balance.

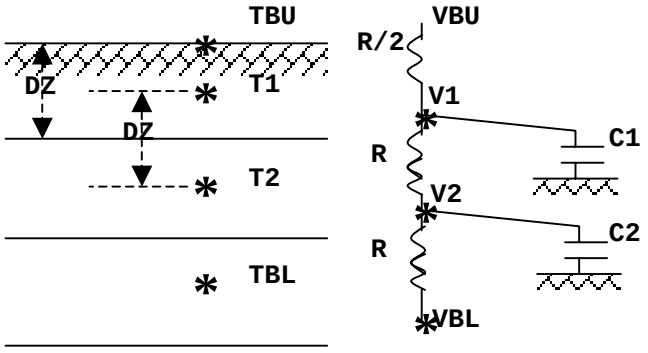


Figure 3.6. Heat flow and temperature regime in the soil layer: (a). Left: Heat flow, (b). Right: Electric network analogy.

3.5.3. Heat flow and temperature regime in the soil

Heat flow in the soil is complicated because heat flow is associated with water flow. However, in most cases, it is sufficient to consider heat flow using apparent thermal conductivity, which includes the effect of water flow. Then heat flow in the soil is that in a solid body.

Referring to Fig. 3.6a, let us suppose the flat ground is divided into three even layers in depth and the temperatures at the middle of each layer are **T1**, **T2** and **TBL**, respectively. The bottom temperature **TBL** is a boundary condition and the temperature at the surface of the ground is **TBU.**

Referring to this analogy and the scheme in Fig. 3.5, the following equations are derived, as the temperature increase in the layer in a small time increment is equal to the total heat flow in the layer considered from the surroundings, in this case the upper and lower layers:

It is clear from the same figure that an electric passive network system, which is called π (pi) network, can represent the heat flow in the soil layer (see Fig. 3.6b).

$$\mathbf{CS*VS*}(d\mathbf{T1}/d\mathbf{t}) = \mathbf{KS*AS*}((\mathbf{TBU-T1})/\mathbf{DZ}*2+(\mathbf{T2-T1})/\mathbf{DZ}) \quad (3.4)$$

$$\mathbf{CS*VS*}(d\mathbf{T2}/d\mathbf{t}) = \mathbf{KS*AS*}((\mathbf{T1-T2})/\mathbf{DZ}+(\mathbf{TBL-T2})/\mathbf{DZ}) \quad (3.5)$$

where **CS** is volumetric heat capacity of the soil, **VS** is the volume of the soil layer, **AS** is the surface area of the soil layer, **DZ** is the thickness of one layer, **KS** is the thermal conductivity of the soil, and **t** is time. We assumed that heat flows in from the upper and the lower layers. This is appropriate if we consider properly the signs involved. If the flow is opposite previously assumed direction, then the sign is inverted automatically.

These two equations are in the differential form. Again, they can be rearranged into the integral forms of **CSMP** (eqs. 3.6a and 3.7a) and the differential forms of **MATLAB** (eqs. 3.6b and 3.7b) as shown below:

T1 = INTGRL(IT1, KS*((TBU - T1)*2.0 + (T2 - T1))/ DZ/DZ/CS) (3.6a)

T2 = INTGRL(IT2, KS*((T1 - T2) + (TBL - T2))/ DZ/DZ/CS) (3.7a)

DIFF_T1 = KS*((TBU - T1)*2.0 + (T2 - T1))/ DZ/DZ/CS (3.6b)

DIFF_T2 = KS*((T1 - T2) + (TBL - T2))/ DZ/DZ/CS (3.7b)

With two unknowns, **T1** and **T2**, and two boundary conditions, **TBU** and **TBL,** the system can be solved.

Let's assume that heat flow in the soil layer shown in Fig. 3.7 is two-dimensional, with vertical and horizontal flows. Each heat flow is proportional to the temperature gradient and thermal conductivity and inversely proportional to the distance.

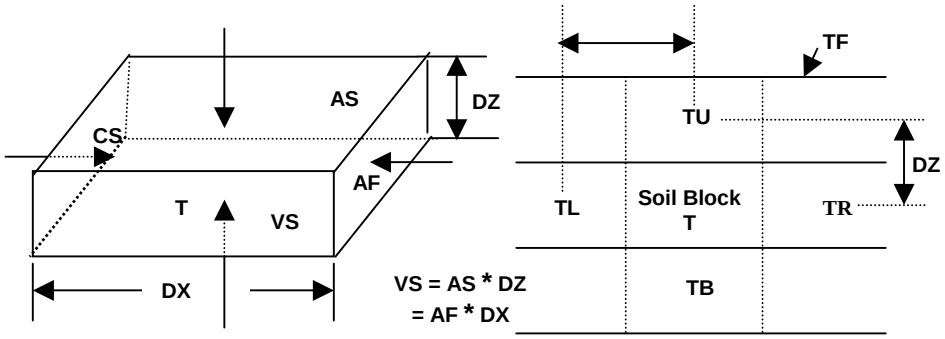


Figure 3.7. Basic diagram of heat flow in a system.

Suppose each temperature shown in the figure is at the center of a soil block. Then the differential equation for the system is:

$$\mathbf{CS} * \mathbf{VS} * (d\mathbf{T}/d\mathbf{t}) = \mathbf{KS} * (\mathbf{AS} * ((\mathbf{TU}-\mathbf{T}) / \mathbf{DZ} + (\mathbf{TB}-\mathbf{T}) / \mathbf{DZ}) + \mathbf{AF} * ((\mathbf{TR}-\mathbf{T}) / \mathbf{DX} + (\mathbf{TL}-\mathbf{T}) / \mathbf{DX})) \quad (3.8)$$

where **KS** is the thermal conductivity of the soil, **AS** is soil surface area for vertical heat flow, **AF** is that for horizontal heat flow, **DX** is the horizontal distance from center to center of soil blocks, **DZ** is the vertical distance from center to center, and **CS** is the heat capacity of the center soil block.

The differential form is more common in mathematical expressions, but the integral form is straightforward for programming in simulation languages. Therefore, the **CSMP** and **MATLAB** expressions for eq. 3.8 are as follows:

```
T = INTGRL(IT, KS * (((TU - T) + (TB - T)) / DZ / DZ ...
           + ((TR - T) + (TL - T)) / DX / DX) / CS)          (3.9a)

DIFF_T = KS * (((TU - T) + (TB - T)) / DZ / DZ  + ...
               ((TR - T) + (TL - T)) / DX / DX) / CS         (3.9b)
```

Model representation is not necessarily explicit. The form of integration always includes implicit expression, and the function **IMPL** in **CSMP** or a similar function **fzero** in **MATLAB** can be used to solve implicit functions except for integrals. Programming is straightforward and can be a kind of translation of mathematical equations into the expressions in each language. Therefore, the following several examples are the best way to understand programming and thus modelling. An ellipsis (...) indicates that the statement continues to the following line in the **CSMP** and **MATLAB** programs.

3.6. A MODEL FOR TEMPERATURE REGIMES IN THE SOIL (**CUC01**)

3.6.1. Model description

Fig. 3.8 shows the Command Window of **MATLAB** running the **cuc01** model. The user can enter '**cuc01**' or '**cuc01(n)**' to run the model, where n is 1 to 4. Entering '**cuc01**' will have the same result as with '**cuc01(1)**'. Fig. 3.9 shows the result of the CUC01 model in the Figure Window. Fig. 3.10 shows the scripts of the model, which consists of a main program (Fig. 3.10a) and a function subprogram (Fig. 3.10b). In the **MATLAB** program, **%** denotes that the line is a comment only. The command '**function cuc01(trial)**' is the first line of the script, in which '**cuc01**' is the name of the function and needs to be consistent with the file name. The variable 'trial' contains the number carried into the program. If there is no argument or the argument value is larger than 4 or smaller than 1, the value of 1 will be assigned to 'trial'. Based on the '**trial**' value, the '**switch…case…end**' provides branching with various pairs of **ks** and **cs** values. The thermal conductivity of the soil, **ks** indicates how fast the heat will transfer through the soil, and the heat capacity of the soil, **cs** indicates how much heat can be store by the soil. These values are declared as **global** variables in the second line of the scripts listed in Fig. 3.10a and Fig. 3.10b.

The matrix variable **y0** contains the initial temperatures of T_1 to T_5. As listed in Fig. 3.10a, **y0** is a 5 by 1 matrix (column vector). The **y0** matrix is then fed to the subprogram '**soil01.m**' using the **ode23()** function. The core of the main program of the CUC01 model lies in the following command: **[t, y] = ode23('soil01',[tstart tfinal],y0);**

Several functions can be used in solving simultaneous ordinary differential equations, including **ode23, ode45, ode113, ode15s, ode23s, ode23t** and **ode23tb**. The major difference among these functions is in the numerical methods used. For non-stiff differential equations, **ode23**, **ode45** and **ode113** can be used. For stiff differential equations, **ode15s** or **ode23s** can be used. More details can be found from online help of **MATLAB**; for example, type '**help ode23**' in the Command Window followed by the Enter key to learn more about **ode23**. Some helpful information is listed in the last section of this Chapter.

The last line of the '**soil01.m**' subprogram is the **dy** matrix. The dimensions of this matrix must be consistent with the **y0** matrix listed in '**cuc01.m**'.

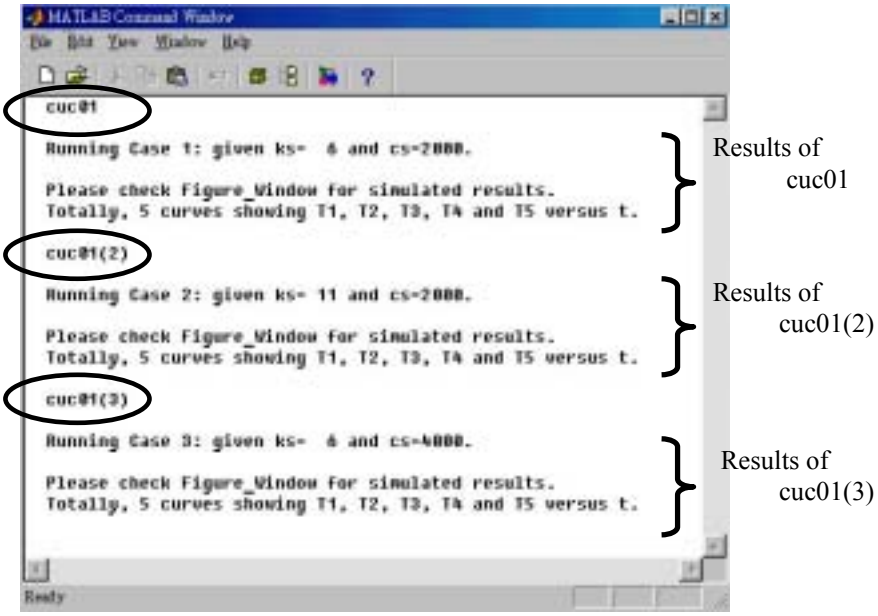


Figure 3.8. Command Window of MATLAB running cuc01 model.

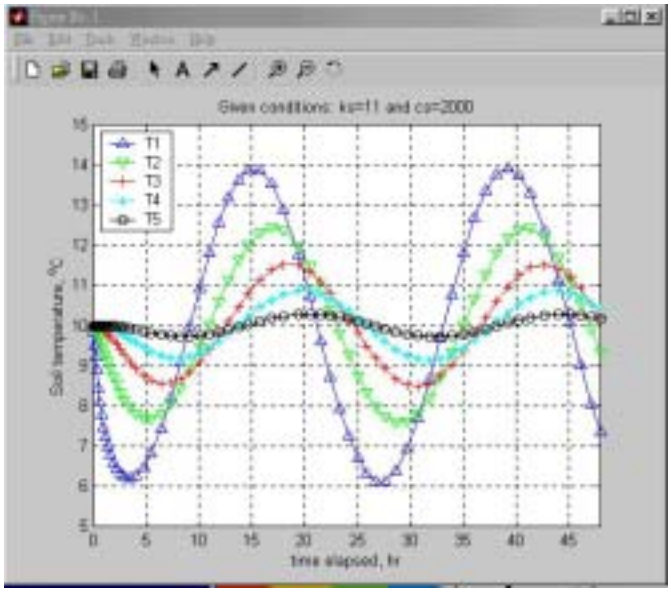


Figure 3.9. Figure Window of MATLAB running cuc01 model.

```
%    Temperature regime in the soil layer                    CUC01.m
%    Boundary condition is surface temperature.
%    Function required: soil01.m
%
function cuc01(trial)
global ks cs
if nargin==0                  % If no argument
   trial=1;                   % use 1 as the default.
end
if trial>4 | trial <1         % If argument_value >4 or <1
   trial =1                   % use 1 as the default
end
switch trial
case 1
       ks=5.5;cs=2000;        % default values
case 2
       ks=11;cs=2000;         % ks doubled
case 3
       ks=5.5;cs=4000;        % cs doubled
case 4
       ks=11;cs=4000;         % ks, cs both doubled
end
% ks: Soil thermal conductivity (kJ/m/C) and ks/3.6 (W/m/C)
% cs: Heat capacity of soil (kJ/m3/C)
%
tstart = 0;    tfinal = 48;
y0=[10;10;10;10;10];          % 5x1 matrix for initial conditions.
[t,y] = ode23t('soil01',[tstart tfinal],y0);
% calling function ode23t with constants and eqs. in 'soil01.m'
%                       with simulated time from tstart to tfinal
%                       with initial conditions in matrix y0 and
%                       with calculated answer in matrix y.
%
plot(t,y(:,1),'b^-',t,y(:,2),'gV-',t,y(:,3),'r+-',…
             t,y(:,4),'c*-',t,y(:,5),'ko-');
axis([-inf,inf,5,15]);
grid on;
xlabel('time elapsed, hr');
ylabel('Soil temperature, ^oC');
tit=['Given conditions: ks=',num2str(ks),' and cs=', num2str(cs)];
title(tit);
legend('T1','T2','T3','T4','T5',2);
fprintf('\n  Running Case %1.0f: given ks=%3.0f and cs=%4.0f. \n\n',…
        trial,ks, cs);
disp('  Please check Figure_Window for simulated results.');
disp('  Totally, 5 curves showing T1, T2, T3, T4 and T5 versus t.');
```

Figure 3.10a. Main program of model for temperature of soil layers **(CUC01.m)**.

The command '**plot(t, y(:,1),'b^-',....)**' draw five curves on the Figure Window as shown in Fig. 3.9. The first and second arguments are the data for the x and y axes. The length of the matrices **t** and **y(:,m)**, where **m** equals 1 to 5, need to be the same. The third argument is a character string that defines line types, plot symbols and color to be displayed on the screen. Entering '**help plot**' in the Command Window can reveal all the possible combinations as listed below.

y	Yellow	**.**	Point	**-**	Solid
m	Magenta	**O**	Circle	**:**	Dotted
c	Cyan	**X**	x-mark	**-.**	Dashdot
r	Red	**+**	Plus	**--**	Dashed
g	Green	*****	Star		
b	Blue	**s**	Square		
w	White	**d**	Diamond		
k	Black	**v**	Triangle (down)		
		^	Triangle (up)		
		<	Triangle (left)		
		>	Triangle (right)		
		p	Pentagram		
		h	Hexagram		

The command '**axis**' allows self-arrangement on both x and y axes. The '**-inf**' and '**inf**' stand for no preset lower bound (LB) and upper bound (UB) of this axis. The first two parameters are for the LB and UB of the x axis and the third and fourth parameters are for the LB and UB of the y axis, respectively.

There are four commands frequently used after the **plot** command. Commands '**xlabel('text')'** and '**ylabel('text')'** allow the user to assign text to the x and y axes; commands '**title('text')'** and '**legend('text1','text2',…,pos)'** allow the user to assign text to the title of the plot and to the legend, respectively. The last argument '**pos**' of '**legend()**' places the legend in the specified location:

0 = Automatic "best" placement (least conflict with data)
1 = Upper right-hand corner (default)
2 = Upper left-hand corner
3 = Lower left-hand corner
4 = Lower right-hand corner
-1 = To the right of the plot

The '**fprintf(format,a,…)**' command writes formatted data to the screen, and '**format**' is a string containing C language conversion specifications. Conversion specifications involve the character %, optional flags, optional width and precision fields, optional subtype specifier, and conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. For more details, see the '**fprintf'** function description in the online help or refer to a C language manual. The special formats \n,\r,\t,\b,\f can be used to produce linefeed, carriage return, tab, backspace, and form feed characters respectively. Use \\ to produce a backslash character and %% to produce the percent character. The command '**disp(x)**' can display an array on the screen. If **x** is a string of text, the text is displayed. The results of the last three commands of the script listed in Fig. 3.10a can be found in Fig. 3.8 as indicated by the large '**}**' sign.

Fig. 3.10b lists the script of '**soil01.m**'. Since the **ode** function must return a column vector, **dy** is the column vector to be returned as listed in the second line

from the bottom of Fig. 3.10b. In the script for calculating **TF**, '**pi**' is used, and is a reserve word of **MATLAB**, representing π. In the subprogram '**soil01.m**', there are 5 unknowns, **y(1)** to **y(5)** with 5 ordinary differential equations.

```
% Subprogram to be used with cuc01.m                          soil01.m
function dy = soil01(t,y)
global ks cs
z=0.1;  % Depth of each soil layer (m)
T0=10;  % Average outside temperature (C)
TU=5;   % Amplitude, temperature variation
TBL=10; % Boundary soil temperature (C)
TF=T0+TU*sin(2*pi/24.*(t-8));
% t is time (in hours)
% TF: Soil temperature of surface layer (C)
%     Maximum temperature occurs
%     at 2 o'clock in the afternoon
T1=y(1);T2=y(2);T3=y(3);
T4=y(4);T5=y(5);
DIFF_T1=2*(TF-T1)+(T2-T1);
DIFF_T2=(T1-T2)+(T3-T2);
DIFF_T3=(T2-T3)+(T4-T3);
DIFF_T4=(T3-T4)+(T5-T4);
DIFF_T5=(T4-T5)+(TBL-T5)*2;
val=ks/z/z/cs;
dy = [DIFF_T1; DIFF_T2; DIFF_T3; DIFF_T4; DIFF_T5]*val;
% Format of dy should be consistent with y0 in cuc01.m (5x1 matrix)
```

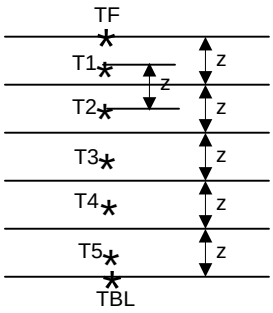


Figure 3.10b. Subprogram of cuc01 model **(soil01.m)** and soil diagram.

3.7. APPLICATION TO STEADY STATE MODELS

Once the dynamic model for non-steady-state conditions has been developed, it can be easily applied to steady-state conditions. For example, if the two boundary conditions in eqs. 3.4 and 3.5 are constant, the left-hand side of the equations will become zero after a certain period of time. Then, the temperature gradient would be linear, and each temperature would be found by interpolation using the physical properties of the soil layer.

3.8. MORE ON MATLAB

Fig. 3.11 shows '**cuc01a.m**', which is an expansion of '**cuc01.m**' with more **MATLAB** commands included. The command '**tic**' starts a stopwatch timer; '**toc**' reads the stopwatch timer. The execution time required between commands '**tic**' and '**toc**' will be displayed on the screen upon the execution of '**toc**'. Note that the **y0** matrix looks different from the one listed in '**cuc01.m**'; however, they are in fact the same. The **y0** listed in Fig. 3.10a is a 5 by 1 matrix (column vector) and the **y0** listed in Fig. 3.11 is the transpose matrix of a 1 by 5 matrix. The transpose matrix of a row vector is again, a column vector.

The command '**h1=findobj('tag','Temperature')**' will find the object using 'Temperature' as a tag name and assign the handle to the **h1** variable. Following by '**close(h1)**' will close the h1 object, that is the one with 'Temperature' as a tag name. Adding these two commands, prior to the '**figure()**' command can prevent opening too many Figure Windows with the 'Temperature' tag name if the program is executed several times.

The command '**figure()**', by itself, creates a new Figure Window. Many properties were involved in the Figure Window such as '**tag**', "**Resize**', '**MenuBar**', '**Name**', '**NumberTitle**', '**Position**', etc. Fig. 3.9 was created without using the '**figure()**' command and Fig. 3.12 was created with the '**figure()**' command listed in the '%--[Figure 1]--' section of the script in Fig. 3.11. Fig. 3.12 has a user-defined 'Name', that is the text written at the top of the Figure Window, and also is without the command menu and icons listed in the second and third rows from the top of Fig. 3.9.

Assigning **figure** to a handle using the command '**h=figure(…)**', followed by '**get(h)',** will generate a list of figure properties and their current values. More details can be found in online help.

The command '**h=plot(…..)**' assigns the plotting operation to a handle '**h**', allows future manipulation on this plot such as setting the line width, and returns its handle.

```
%    Temperature regime in the soil layer                    CUC01a.m
%    Boundary condition is surface temperature
%    Function required: soil01.m
%
function cuc01a(trial)
global ks cs
if nargin==0 | trial>4 | trial <1,  trial =1;       end
switch trial
       case 1, ks=5.5;       cs=2000;
       case 2, ks=11;        cs=2000;       % ks doubled
       case 3, ks=5.5;       cs=4000;       % cs doubled
       case 4, ks=11;        cs=4000;       % ks, cs both doubled
end
%---[Core]-----------------------------------------------------------
tic                                        % start recording time
tstart = 0;tfinal = 48;
IT1=10;IT2=10;IT3=10;IT4=10;IT5=10;
y0 = [10 10 10 10 10]' ;  % Transpose of row matrix is column matrix
[t,y] = ode23('soil01',[tstart tfinal],y0);
toc                       % show elapsed time from tic to toc
%---[Figure1]--------------------------------------------------------
h1=findobj('tag','Temperature'); close(h1);
% prevent from opening too many same figure windows
figure('tag','Temperature','Resize','on','MenuBar','none',...
   'Name','CUC01a.m (Figure 1: Temperatures in 5 soil layers)',...
   'NumberTitle','off','Position',[160,80,520,420]);
h=plot(t,y(:,1),'k-*',t,y(:,2),'b:o',t,y(:,3),'r:^',t,y(:,4), ,...
   'go-',t,y(:,5));
set (h,'linewidth',2); axis([-inf,inf,5,15]);grid on;
xlabel('time elapsed, hr'); ylabel('Soil temperature, ^oC');
legend('T1','T2','T3','T4','T5');
```

```
%---[Figure2]-----------------------------------------------------------
h2=findobj('tag','Temp5');close(h2);
h2=figure('tag','Temp5','Resize','on','MenuBar','none',...
   'Name','CUC01a.m (Figure 2: Temperature in each soil layer)',...
   'NumberTitle','off','Position',[200,40,520,420]);
figure(h2); subplot(5,1,1);   plot(t,y(:,1),'k-*');  ylabel('T1');
% draw the 1st plot out of 5 row x 1 col. plots per figure
axis([-inf,inf,5,15]); grid on;
subplot(5,1,2);       plot(t,y(:,2),'b:o');  ylabel('T2');
% draw the 2nd plot out of 5 row x 1 col. plots per figure
axis([-inf,inf,5,15]); grid on;
subplot(5,1,3);       plot(t,y(:,3),'r:^');  ylabel('T3');
axis([-inf,inf,5,15]); set(gca,'ytick',[8 10 12]);   grid on;
subplot(5,1,4);       plot(t,y(:,4),'go-');  ylabel('T4');
axis([-inf,inf,5,15]); set(gca,'ytick',[8 10 12]);   grid on;
subplot(5,1,5);       plot(t,y(:,5));        ylabel('T5');
xlabel('time elapsed, hr');
axis([-inf,inf,5,15]); set(gca,'ytick',[8 10 12]);   grid on;
clc;    % clear command window
disp('  Thank you for using CUC01a.');   disp(' ');
disp('  You can enter ''close all'' to close Figure_Windows.');
```

Figure 3.11. Main program of CUC01a model **(CUC01a.m)**.

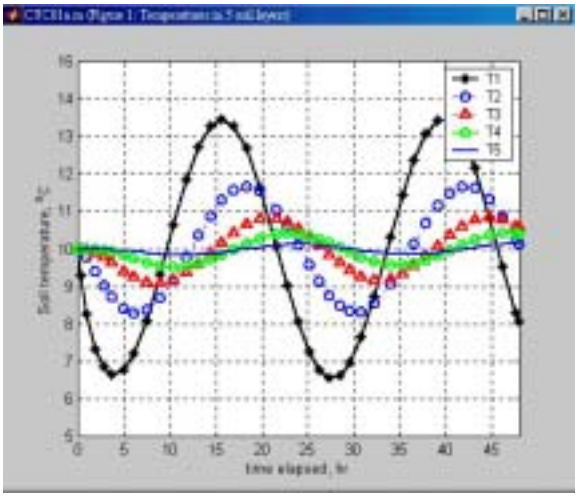


*Figure 3.12. Figure generated from '%--[Figure 1]--' section of cuc01a.m***.**

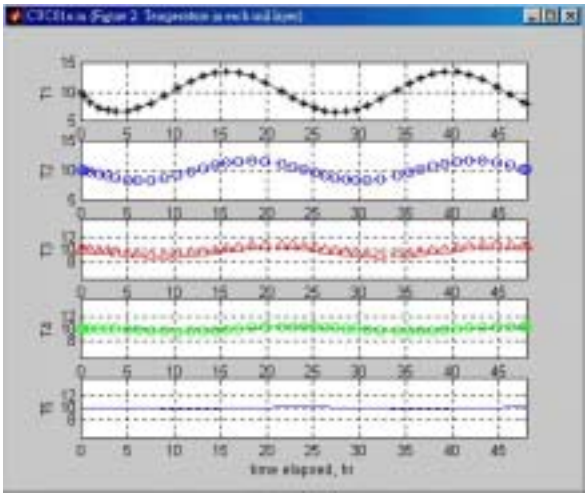


Figure 3.13. Figure generated from '%--[Figure 2]--' section of cuc01a.m.

The script listed in the '%--[Figure 2]--' section of Fig. 3.11 generates Fig. 3.13. The command '**figure(h)'** makes h the current figure, forces it to become visible, and brings it to the foreground, in front of all other figures on the screen. If Figure h does not exist and h is an integer, a new figure is created with handle **h**. The command '**subplot(m,n,p)**' breaks the Figure Window into an m-by-n matrix of small axes, and selects the p-th axes for the current plot. The command '**plot()**' following '**subplot(…,p)**' draws the plot in the p-th axes. As shown in Fig. 3.13, there are five subplots in one Figure Window. The last three subplots have different y ticks compared with the first two subplots. These y ticks of the last three subplots were generated using the '**set(gca,'ytick',[8 10 12])**' command. The term 'gca', representing 'get handle to current axis', is a reserve word in **MATLAB**. Both 'ytick' and 'xtick' can be assigned by the user with the '**set(gca,..)**' command. The command '**clc**' is used to clear the Command Window.

3.9. SIMULINK

SIMULINK is one of the toolboxes linked with **MATLAB** and is suitable for dynamic simulation. It is a kind of graphical approach and is based on the concept of analog computers as shown in Fig. 3.14. This figure shows the model CUC01 in **SIMULINK**. Fig. 3.1 can be a step to understanding this figure. In Fig. 3.14, the integrator is labeled 1/S, the summers are shown as squares with plus and minus signs, and the coefficients to be multiplied are in triangles. The construction of the model is straightforward. The flow is from left to right. There are two boundary conditions, the soil surface temperature change is given by a sine wave (all parameters are hidden under each symbol) plus a constant, and the bottom

temperature is given as a constant 10. The scope symbol is the output, and any of the outputs T1 through T5 can be seen through the scope. It is apparent that the first line components are all for the temperature of the first soil layer, T1. T1 is the output of the first integrator and is fed back as an input to the summers with a minus sign. Then, the first boundary condition, the soil surface temperature minus T1 is one of the two inputs to the next summer. Following this approach, the whole diagram can be understood.

In this book, models in **SIMULINK** are not included because of the space limitations.

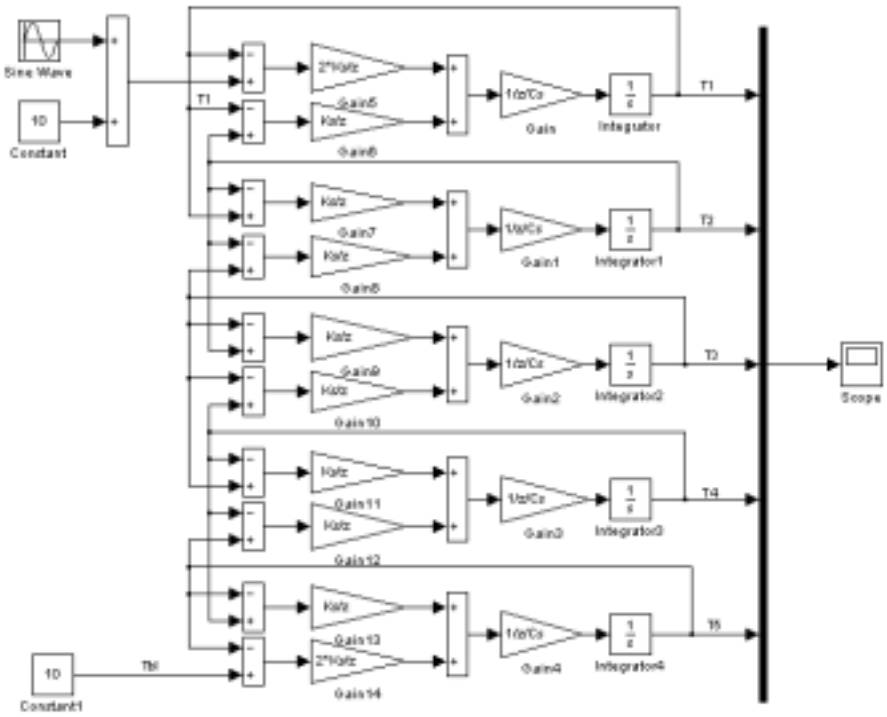


Figure 3.14. The model CUC01 in SIMULINK.

MATLAB FUNCTIONS USED

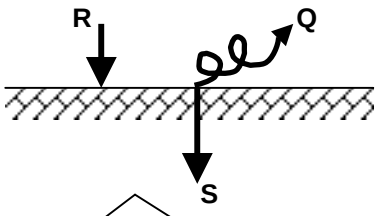
%	Comments.
;	Prohibit from display to the monitor.
:	Represent a complete row or column of a matrix.
…	Continue in next line.
axis	Control axis scaling and appearance. **Axis**([XMIN XMAX YMIN YMAX]) sets scaling for the x- and y-axes on the current plot.
clc	Clear command window.
disp	Display array. **Disp**(X) displays the array, without printing the array name. In all other ways, the same as leaving the semicolon off an expression except that empty arrays don't display. If X is a string, the text is displayed.
figure	Creates a new figure window, and returns its handle.
findobj	Find objects with specified property values.
fprintf	Write formatted data to screen.
global	Define global variable.
grid	Grid lines. **Grid on** adds grid lines to the current axis. **Grid off** takes them off. **Grid** , by itself, toggles the grid state of the current axis.
gca	Get handle with Current Axis.
legend	Graph legend. **Legend**(string1,string2,string3, ...) puts a legend on the current plot using the specified strings as labels. **Legend**(...,Pos) places the legend in the specified location: 0 = Automatic "best" placement (least conflict with data) 1 = Upper right-hand corner (default) 2 = Upper left-hand corner 3 = Lower left-hand corner 4 = Lower right-hand corner -1 = To the right of the plot
num2str	Convert number to string.
ode23	Solve non-stiff differential equations, low order method. (**MATLAB** 4 and higher versions) [T, Y] = **ode23**('F',TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates the system of differential equations y' = F(T,Y) from time T0 to TFINAL with initial conditions Y0. 'F' is a string containing the name of an ODE file. Function F(T, Y) must return a column vector. Each row in solution array Y corresponds to a time returned in column vector T.
ode45	Solve non-stiff differential equations, medium order method. (**MATLAB** 4 and higher versions)
ode113	Solve non-stiff differential equations, variable order method. (**MATLAB** 5 and higher versions)
ode15s	Solve stiff differential equations and DAEs, variable order method. (**MATLAB** 5.2 and higher versions)
ode23s	Solve stiff differential equations, low order method.

	(**MATLAB** 5.2 and higher versions)
ode23t	Solve moderately stiff ODEs and DAEs, trapezoidal rule. (**MATLAB** 5.2 and higher versions)
ode23tb	Solve stiff differential equations, low order method. (MATLAB 5.2 and higher versions)
Plot	Linear plot. **Plot**(X,Y) plots vector Y versus vector X.
Set	Set object properties. **Set**(H,'PropertyName',PropertyValue) sets the value of the specified property for the graphics object with handle H. H can be a vector of handles, in which case SET sets the properties' values for all the objects.
subplot	Create axes in tiled positions.
Tic	Start a stopwatch timer.
Title	Graph title. **Title**('text') adds text at the top of the current axis.
Toc	Read a stopwatch timer.
xlabel	X-axis label. **Xlabel(**'text') adds text beside the X-axis on the current axes.
Ylabel	Y-axis label. **Ylabel**('text') adds text beside the Y-axis on the current axes.

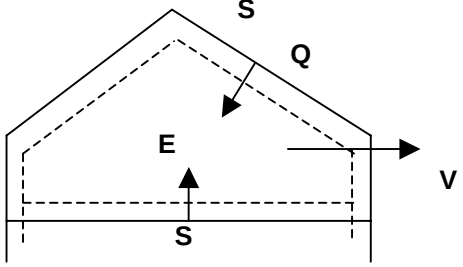
PROBLEMS

1. Develop the energy balance equations.

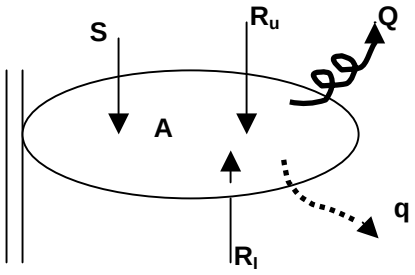
   a) On the bare ground surface, assume absorbed radiation is **R** (J/s/m^2), out-going heat flux mostly by convection is **Q** (J/s/m^2), and heat flux into the soil is **S** (J/s/m^2).



   b) Heat (**E**) is stored in the air mass in the greenhouse after obtaining heat (**Q**) from the covering surface, **S** from the ground surface, and **V** by ventilation. Units are all (J/s).

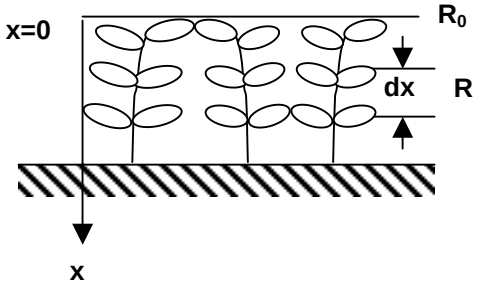


   c) For a single horizontal leaf of area **A** (cm^2), assume no heat capacity of the leaf and solar radiation absorbed is **S** (W/cm^2), net long wave radiation (effective radiation) on the upper side is $\mathbf{R_u}$ and that at the lower side is $\mathbf{R_l}$ (J/cm^2/s), heat convection from the leaf is **Q** (J/cm^2/s), transpiration is **q** (g/cm^2/s), and latent heat of vaporization is **L** (J/g).



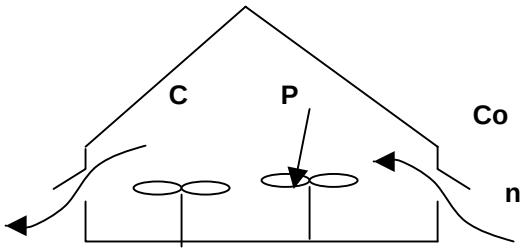
2. Develop the differential equations.

   a) Direct solar radiation **R** penetrated into a plant canopy follows Lambert-Beer's law. Assuming radiation just above the canopy is $\mathbf{R_0}$ and extinction coefficient in the canopy is **k**, describe the penetrated radiation rate in terms of the depth from the top of the canopy.



b) A coffee cup is filled with water is heated by **Q** (J/s), and the over-all heat loss from the cup is **L** (J/s). Describe temperature increase in the cup. Assume the amount of water in the cup is **W** (g), the heat capacity of water is $\mathbf{C_p}$ (J/^{o}C/cm^3) and its density is 1 (g/cm^3).

c) Describe the carbon dioxide concentration change in the greenhouse, assuming outside concentration is $\mathbf{C_o}$ (ppm) and constant, inside is **C** (ppm), no generation from soil, consumption by plant photosynthesis is **P** (mg/cm^2/min), total leaf area is **A** (cm^2), greenhouse volume is **V** (m^3), and ventilation rate is **n** (1/h). Note that 1 mole of carbon dioxide (44 g) is equivalent to 22400 cm^3 at 0 ^{o}C and 1 atm. More discussion on the concentration units, μ *l*/*l*, ppm, vpm, and μ mol/mol is given in section 10.5.



3. Write the following differential equations in **MATLAB**.

a) $d\mathbf{y}/d\mathbf{t} = -\mathbf{A} * \mathbf{y}$ and $\mathbf{y}_{t=0} = \mathbf{B}$
b) $d\mathbf{y}/d\mathbf{t} = \mathbf{cos(y)}$ and $\mathbf{y}_{t=0} = 1$
c) $d^2\mathbf{x}/d\mathbf{t}^2 = \mathbf{F} - \mathbf{A} * d\mathbf{x}/d\mathbf{t} - \mathbf{B} * \mathbf{x}$ and $\mathbf{x}_{t=0} = \mathbf{X0}$; $d\mathbf{x}/d\mathbf{t}_{t=0} = \mathbf{DX0}$
d) Host-parasite or predator-prey model
$d\mathbf{H}/d\mathbf{t} = (\mathbf{K}_1 - \mathbf{K}_2 * \mathbf{P}) * \mathbf{H}$ and $\mathbf{H}_{t=0} = \mathbf{H0}$
$d\mathbf{P}/d\mathbf{t} = (-\mathbf{K}_3 + \mathbf{K}_4 * \mathbf{H}) * \mathbf{P}$ and $\mathbf{P}_{t=0} = \mathbf{P0}$
e) Shells and limpets model
$d\mathbf{S}/d\mathbf{t} = \mathbf{K}_1 * \mathbf{S} - \mathbf{K}_2 * \mathbf{S}^2 - \mathbf{K}_3 * \mathbf{L}$ and $\mathbf{S}_{t=0} = \mathbf{S0}$
$d\mathbf{L}/d\mathbf{t} = \mathbf{B} * \mathbf{K}_3 * \mathbf{S} * \mathbf{L} - \mathbf{K}_4 * \mathbf{L} - \mathbf{K}_5 * \mathbf{L}$ and $\mathbf{L}_{t=0} = \mathbf{L0}$

4. Develop **MATLAB** programs to calculate the following equations and run from time t = 0 to 5.

a) Growth curve
$\mathbf{y} = \mathbf{exp}\ (\mathbf{t})$
$d\mathbf{y}/d\mathbf{t} = \mathbf{y}$ and $\mathbf{y}_{t=0} = 1$
b) Decay curve
$\mathbf{y} = 10 * \mathbf{exp}\ (-0.1 * \boldsymbol{t}\ )$
$d\mathbf{y}/d\mathbf{t} = -0.1 * \mathbf{y}$ and $\mathbf{y}_{t=0} = 10$

c) Periodic curve
   $\mathbf{y} = 3 * \mathbf{sin}\ (\ 0.6 * \mathbf{t}\ )$
   $d^2\mathbf{y}/d\mathbf{t}^2 = -\ 0.36 * \mathbf{y}$ and $d\mathbf{y}/d\mathbf{t}_{\ t=0} = 1.8$; $\mathbf{y}_{t=0} = 0$
d) Response curve
   $\mathbf{y} = 1 - \mathbf{exp}\ (-\ 3 * \mathbf{t}\ )$
   $d\mathbf{y}/d\mathbf{t} = 3 - 3 * \mathbf{y}$ and $\mathbf{y}_{t=0} = 0$
e) Rectangular hyperbola (Michaelis-Menten relation)
   $\mathbf{u} = \mathbf{k} * \mathbf{t}\ /\ (\ \mathbf{K} + \mathbf{t}\ )$
   $d\mathbf{u}/d\mathbf{t} = \mathbf{k}*\mathbf{K}/(\ \mathbf{K} + \mathbf{t}\ )^2$ and $\mathbf{u}_{t=0} = 0$
f) Logistic curve
   $\mathbf{W} = \mathbf{W}_i * \mathbf{W}_f * \mathbf{exp}\ (\mathbf{W}_f * \mathbf{k} * \mathbf{t}\ )\ /(\ \mathbf{W}_f - \mathbf{W}_i + \mathbf{W}_i * \mathbf{exp}\ (\mathbf{W}_f * \mathbf{k} * \mathbf{t}))$
   $d\mathbf{W}/d\mathbf{t} = \mathbf{k}*\ (\ \mathbf{W}_f - \mathbf{W}\ )*\ \mathbf{W}$ and $\mathbf{W}_{t=0} = \mathbf{W}_i$

5. Derive that $d\mathbf{X1}/d\mathbf{t} = \mathbf{A} * \mathbf{X1} - \mathbf{X2}$ and $d\mathbf{X2}/d\mathbf{t} = -\ \mathbf{X1}$ are equivalent to
   $d^2\mathbf{X1}/d\mathbf{t}^2 = \mathbf{A} * d\mathbf{X1}/d\mathbf{t} + \mathbf{X1}$.

6. Write a **MATLAB** statement equivalent to eq. 3.3, assuming the initial condition of **P** is **A**.

7. Modify the program **CUC01**, assuming the thermal conductivity of soil **KS** is a function of temperature. Use the expression **KS** = 5.5 + 0.1 * **TEMP**, where **TEMP** is soil temperature.

8. Derive the system of differential equations from Figure 3.14. Note: The input sine function is given in Figure 3.10b.