iSCRIPT User's Manual

TTC Technologies, Inc.
P.O. Box 1527
Stony Brook, New York 11790

June 2006, August 2006, December 2006, April 2007, July 2007, November 2007

Table of Contents

Table of Contents, cont'd.

Table of Contents, cont'd.

1. Introduction

1.1. Introduction to Industrial Design/Optimization

iSCRIPT was developed primarily for performance calculations and design/optimization of engineering systems, with a focus on large systems. As a result, iSCRIPT was designed to include tools for modeling and optimizing large-scale systems. It has provisions for modeling a system in a decomposed fashion, and automatically executes in parallel, taking advantage of multi-processor computational resources. iSCRIPT also contains novel optimization procedures, which allow for the optimization of decomposed systems in an efficient manner. To facilitate detailed modeling of engineering components and systems, iSCRIPT has the full features of a programming language, comparable to those available in traditional programming environments such as FORTRAN.

A common technique for carrying out a multi-level design/optimization can be divided into three tasks, as shown in Figure 1.1 below.

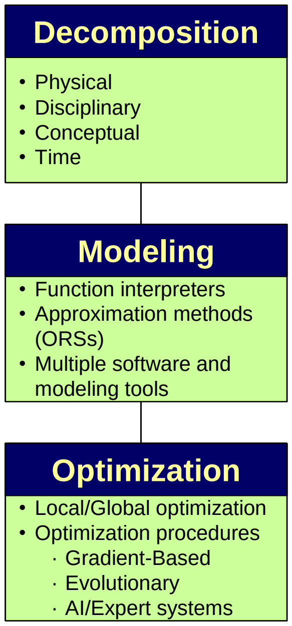


Figure 1.1. Tasks in a multi-level design/optimization analysis.

1.1.1. Decomposition

In physical decomposition, the system is divided up into physically-interacting subsystems, each possessing a certain degree of autonomy but depending on other subsystems via a number of coupling variables. Disciplinary decomposition divides the system along the lines of different disciplines, such as thermodynamics, economics, aerodynamics, etc. Conceptual decomposition breaks down the system according to the type of variables. For instance, the system can be broken down into operational variables that vary in time and those that do not vary in time. Time decomposition decomposes a dynamic problem into a series of quasi-stationary problems or a series of stationary time segments.

1.1.2. Modeling

Modeling of the various subsystems in a large system typically involves software products from different vendors. A great challenge in this step is the integration of the different software products. Several levels can be identified in the multi-level modeling and optimization process:

Low-Level Function Interpreters and Symbolic Language Programs

These are tools that allow an engineer to specify the equations and models comprising a component in a mathematical form, aggregate these low-level models into higher-level models through additional mathematical expressions and functions. In principle, a complete system can be built using low-level functions. However, the procedure is difficult and prone to error. Sample tools in this category include spreadsheets, such as Excel, and programming and scripting languages, such as MATLAB, Mathematica, or Maple.

Aggregated Component Tools

These are pre-packaged tools for specific models. Examples include engine simulator for computing the thrust and weight of the propulsion system (e.g., Weight Analysis of Turbine Engines (WATE)) or ADVISOR, a public domain drive-train analysis tool, or a heat exchanger program for the various heat exchangers in the sub-systems of aircraft. Component tools are typically treated as a “black box” in the integration of models into the complete system.

Approximation Tools

Response surfaces may be generated from measurements of the performance of a component as a function of selected decision variables and used as the model in the multi-level optimization phase.

1.1.3 Optimization

The use of a detailed representation of a component is a critical factor in terms of computational resources. The problems of interest are typically multi-objective, with objectives such as:

- drag minimization (or maximization of lift/drag ratio in a mission segment)
- gross take-off weight minimization
- fuel consumption minimization
- minimization of acoustic noise during take-off and landing
- cost minimization (capital, operating, and environmental)

Although the capability for multi-objective optimization is usually important, a recombination of the various objectives into a single metric, such as exergy, may sometimes alleviate the multi-objective requirement. iSCRIPT optionally supports exergy-based optimization of aircraft.

Optimization procedures include gradient formulation and genetic algorithms (GA). Gradient-based methods work well for subsystems with continuous variables, but are prone to local optima and divergence when the initial guess solutions are far from the true solutions. Procedures based on genetic algorithms and expert systems are more computationally intensive, but they are not as prone to local optima (are more tolerant to arbitrary initial guess), and can be used for mixed integer problems. In aerospace engineering, the variables include integers, Boolean, and continuous variables. GAs are utilized to isolate the optima while the gradient-based method can be combined with the GA to speed up the "climb to the peak," once all integer variables are set. In other words, a combination of several optimization procedures is typically used for a complex problem.

1.2. Why We Developed iSCRIPT

iSCRIPT was developed mainly because existing tools for engineering design were not explicitly developed for that purpose, and were therefore unable to take advantage of the inherent structures that are present in engineering problems. Engineering context includes (engineering) units, material or physical limits (constraints), or the natural association of variables with components or system, and the association of components with systems or subsystems. Engineers must fill this gap by writing codes to convert variables between different engineering units and procedures, and to coordinate the association of components within a system. This makes the development of detailed large systems very unwieldy and error-prone.

With improved computational power and little additional work, engineers can now afford to carry out performance analysis and design/optimization to increased levels of fidelity. iSCRIPT takes advantage of modern computer architecture by implementing procedures to automatically operate in a parallel environment without requiring the designer to explicitly parallelize models, as would be necessary in a traditional programming environment.

One problem facing designers of large-scale engineering systems is that the models for the design are not always available a single software or programming platform. For instance, engineers designing aircraft may sometimes call on the code – WATE – for the modeling of an engine separate from other models, which may be implemented in other software products. System modeling and design tools must allow systems to be built based on components, some of which are modeled in a different software or programming platform. iSCRIPT is being developed with this cross-platform operability in mind.

1.3. How We Want You to See ISCRIPT

Although iSCRIPT can be used in the same manner as a traditional programming environment, with all the features of a programming language, including decision structures, loop elements, array variables, and subprogram units, it was designed primarily for performance analysis and design optimization of engineering systems. Systems can be described in a building block manner as being composed of subsystems which, in turn, are composed of components (Figure 1.2).

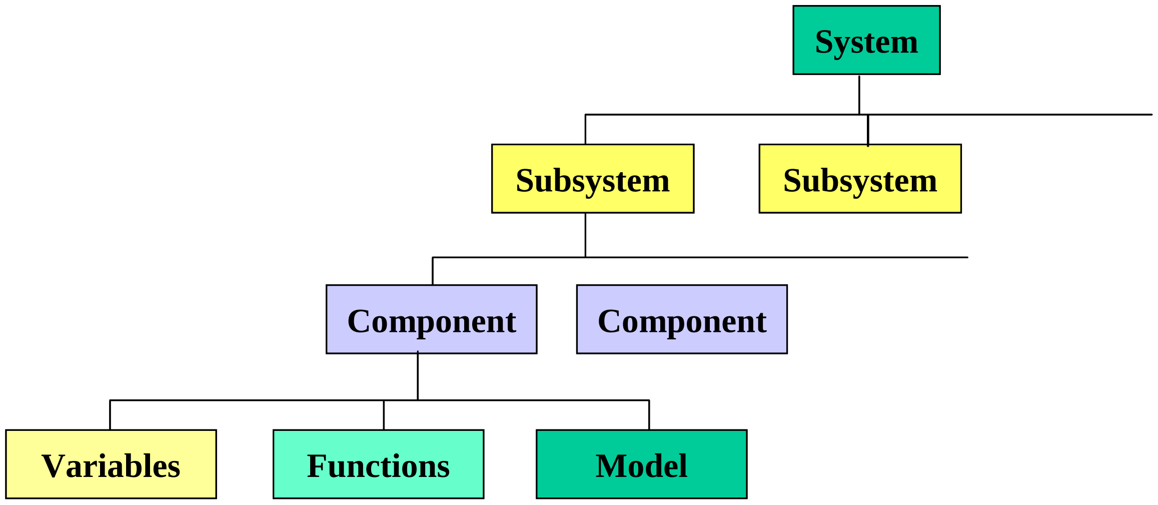


Figure 1.2. Elements of an iSCRIPT system project.

Consequently, iSCRIPT can accommodate virtually any decomposition procedure, as the components may be models of physical components, conceptual segments of a system, or a disciplinary subdivision of a system.

iSCRIPT also contains built-in procedures for optimizing a system made up of decomposed subsystems using the iterative local-global optimization (ILGO) procedure.

As previously mentioned, iSCRIPT was designed to automatically execute in a parallel environment, taking advantage of multi-processor facilities for faster turn-around on simulation tasks.

To provide system-of-systems features, iSCRIPT can execute third-party software, potentially providing access to models available on other software products. iSCRIPT is also compatible with some engineering modeling environments, such as FORTRAN and MATLAB. Programs developed in these environments can, with minimum modification, be run in iSCRIPT or included as part of an iSCRIPT system model.

1.4. Conventions Used in this Manual

The following conventions have been adopted in this manual:

- Arial font will be used for script segments, iSCRIPT keywords, and commands that should be typed on the keyboard as input. For example:

  ```
  if (Burner.T > 2700) then
  ```

CreateComponent (name [,description])

- Segments of a syntax enclosed within square brackets are optional. For instance, given the above syntax template for the CreateComponent command, the following two instances of using the command are acceptable:

  CreateComponent (Burner, ‘Engine Burner’)

  CreateComponent (Burner)

  Note that the comma “,” inside the square bracket is omitted when the choice is that of using the syntax without the square bracket.

- File names and paths are usually italicized *Times Roman*. For example in the instruction below, the italicized portion represents a file path and name:

  Open the file *\SampleScripts\HeatRejection\outputscript.txt* in the iSCRIPT installation directory.

2. Installation

2.1. System Requirements

ISCRIPT requires the following:

- Windows 2000/XP/2003, Mac, or Linux
- Pentium processor with at least 128MB

2.2. Required Software

The following software products may be required in order to use iSCRIPT:

- iSCRIPT installation file
- MPICH, for execution of iSCRIPT in parallel.

Note: MPICH is optional. It is only needed for parallel calculation. If sequential calculation is needed, you do not need to install MPICH.

Obtain these software products as shown below.

iscript_install.exe

(1) The installation file for ISCRIPT can be downloaded from http://www.ttctech.com/SiteFiles/Downloads/iscript_install.exe.

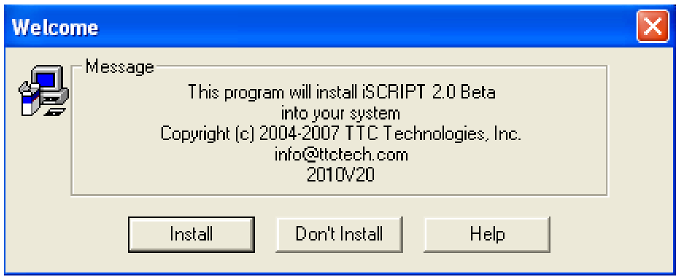
mpich.nt.1.2.5.exe

(2) MPICH, developed by Argonne National Laboratory, is a freely available, portable implementation of MPI, a standard for message-passing protocol for distributed-memory applications used in parallel computing. MPICH can be downloaded from http://www.mcs.anl.gov/mpi/mpich/

2.3. Installing the Required Software

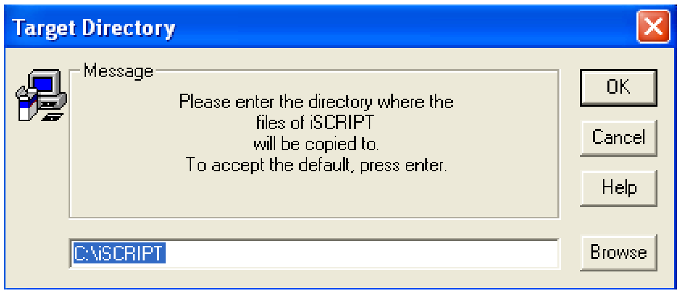
(1) To install iSCRIPT:

- Double-click the installation file.



This program will install iSCRIPT 2.0 Beta
into your system
Copyright (c) 2004-2007 TTC Technologies, Inc.
info@ttctech.com
2010V20

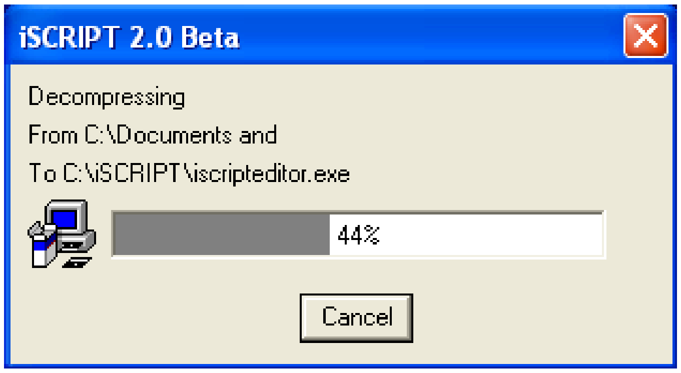
- Click "Install."



Please enter the directory where the
files of iSCRIPT
will be copied to.
To accept the default, press enter.

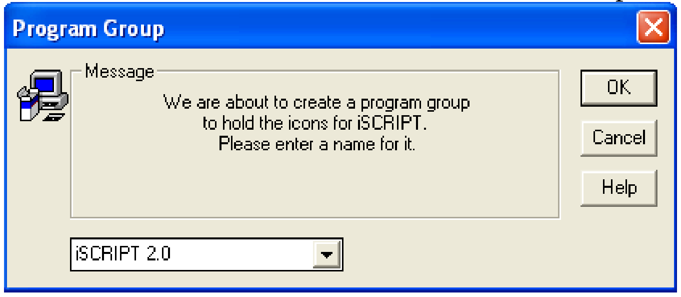
C:\iSCRIPT

- Select where to install iSCRIPT and click "OK."



Decompressing
From C:\Documents and
To C:\iSCRIPT\iscripteditor.exe
44%

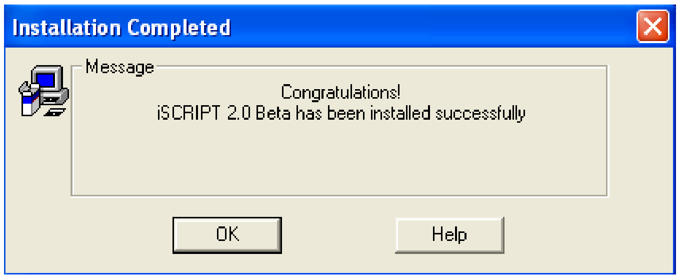
- It may take several minutes for the installation files to be copied.



We are about to create a program group
to hold the icons for iSCRIPT.
Please enter a name for it.

iSCRIPT 2.0

- Click "OK."


Installation Completed

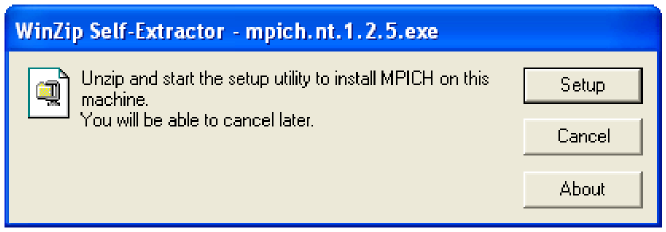
Message

Congratulations!
iSCRIPT 2.0 Beta has been installed successfully

- Click "OK."

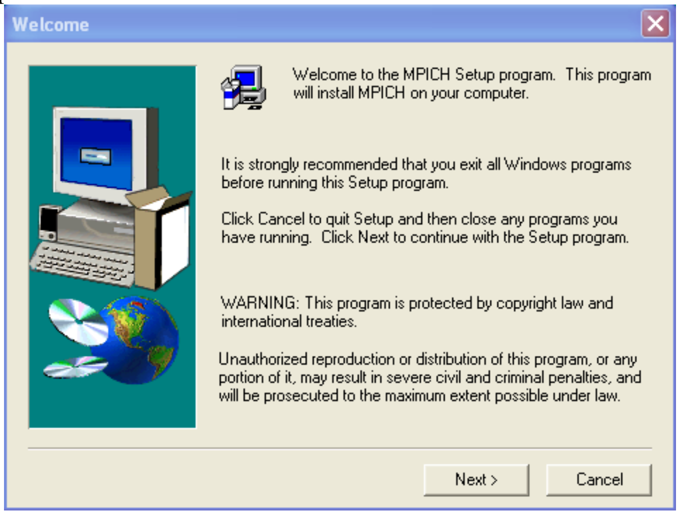
(2) Install MPICH (Optional; only needed for parallel calculation):

- Double click the installation file.


WinZip Self-Extractor - mpich.nt.1.2.5.exe

Unzip and start the setup utility to install MPICH on this machine.
You will be able to cancel later.

- Click "Setup."


Welcome

Welcome to the MPICH Setup program. This program will install MPICH on your computer.

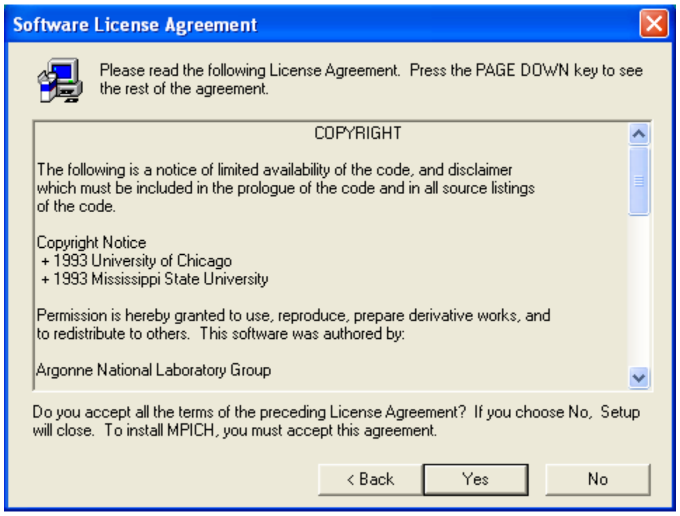
It is strongly recommended that you exit all Windows programs before running this Setup program.

Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

- Click "Next."


Software License Agreement

Please read the following License Agreement. Press the PAGE DOWN key to see the rest of the agreement.

COPYRIGHT

The following is a notice of limited availability of the code, and disclaimer which must be included in the prologue of the code and in all source listings of the code.

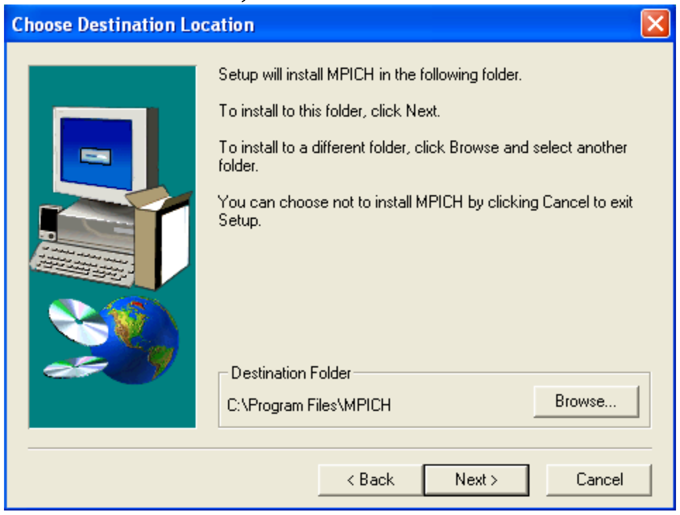
Copyright Notice
+ 1993 University of Chicago
+ 1993 Mississippi State University

Permission is hereby granted to use, reproduce, prepare derivative works, and to redistribute to others. This software was authored by:

Argonne National Laboratory Group

Do you accept all the terms of the preceding License Agreement? If you choose No, Setup will close. To install MPICH, you must accept this agreement.

- Select where to install MPICH, then click "Next."


Choose Destination Location

Setup will install MPICH in the following folder.

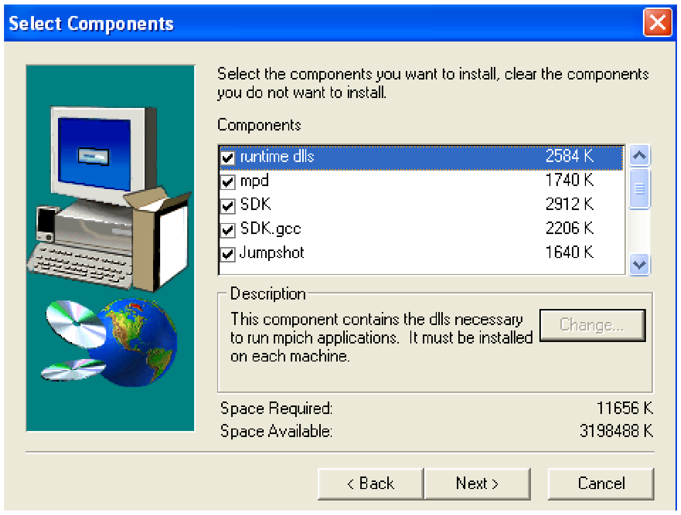
To install to this folder, click Next.

To install to a different folder, click Browse and select another folder.

You can choose not to install MPICH by clicking Cancel to exit Setup.

Destination Folder
C:\Program Files\MPICH

- Click "Next."


Select Components

Select the components you want to install, clear the components you do not want to install.

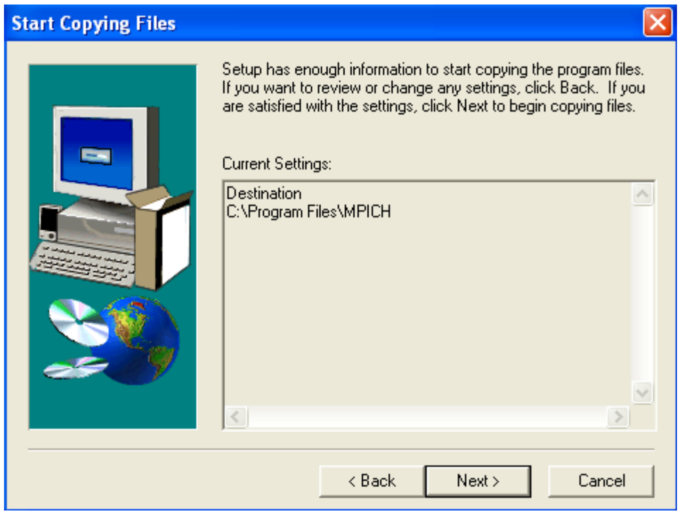
Components

✓ runtime dlls	2584 K
✓ mpd	1740 K
✓ SDK	2912 K
✓ SDK.gcc	2206 K
✓ Jumpshot	1640 K

Description
This component contains the dlls necessary to run mpich applications. It must be installed on each machine.

Space Required: 11656 K
Space Available: 3198488 K

- Click “Next.”

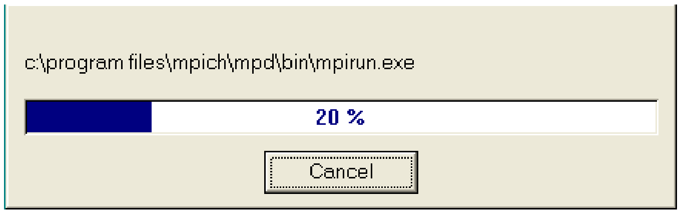


Setup has enough information to start copying the program files. If you want to review or change any settings, click Back. If you are satisfied with the settings, click Next to begin copying files.

Current Settings:

Destination
C:\Program Files\MPICH

- Click “Next.”



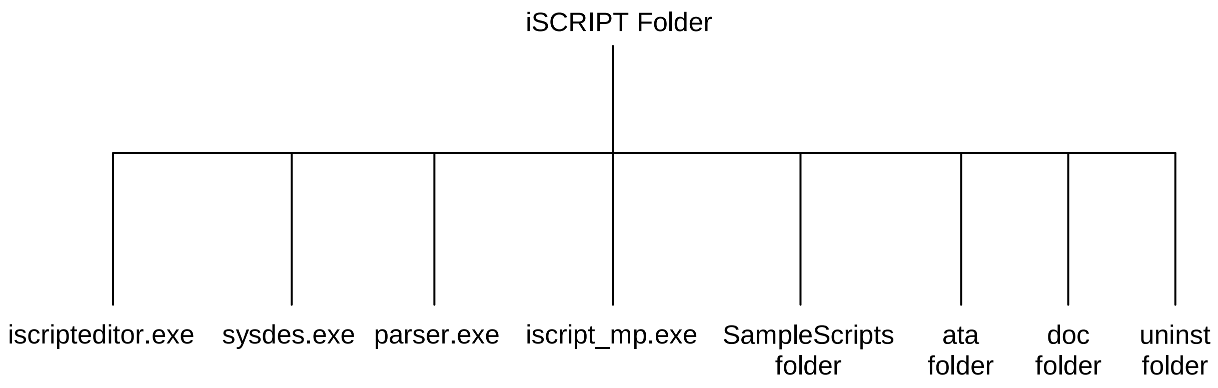
c:\program files\mpich\mpd\bin\mpirun.exe

20 %

More information about the installation of MPICH can be found on the MPICH website.

(3) Installed Files

After iSCRIPT is installed, several folders and executable files are installed in the installation directory (default is C:\iSCRIPT). These main files or folders are:



iscripteditor.exe –	iSCRIPT editor, an iSCRIPT code developing environment.
sysdes.exe –	sequential iSCRIPT executable file. No diagnostic information is printed out by this executable (a quiet mode compared with Parser.exe).
parser.exe –	iSCRIPT analyzer, a sequential iSCRIPT executable file, which is intended to print diagnostics to an output file during execution.
iscript_mp.exe –	a parallel iSCRIPT executable file, which can calculate the problem in parallel machines
SampleScripts –	contains the iSCRIPT sample problems
ata –	contains the iSCRIPT codes for the ata project
doc –	contains the integrated help files for iSCRIPT Editor
unist –	contains the iSCRIPT “uninstall” information

2.4. iSCRIPT Files

There are two types of iSCRIPT files: project files and script files, with the file extensions *.ipr and *.isc, respectively. Sample iSCRIPT codes are presented in Chapter 3. *In this manual, iSCRIPT files and the source scripts developed to solve a specific problem are referred to as an iSCRIPT solution.*

Note: iSCRIPT is not case-sensitive.

2.4.1. Script Files (*.isc)

An iSCRIPT file contains source code. Each file may contain a single program, subprogram, function, or component subprogram, or simply lines of codes not wrapped in any particular structure (program, subprogram, or function). A component subprogram may be considered as a special type of subprogram that computes or solves equations necessary for the performance analysis of the component. An iSCRIPT file may also contain several subprograms and/or functions.

2.4.2. Main Program

When an iSCRIPT solution includes at least one subroutine or function, it is considered as having a program structure. Every iSCRIPT solution with a program structure must contain a main program. There must be only one main program in a solution. When a solution has a program structure, the execution of an iSCRIPT code starts from the main program. When iSCRIPT is used in a component modeling form, the main program usually consists of two parts. The first part is the component creation and component variable declaration, and the second part contains the system evaluation and/or optimization commands.

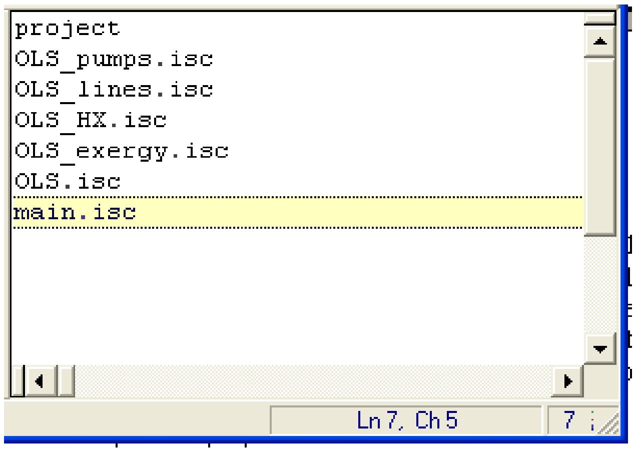
When a solution has no program structure (and necessarily comprises of a single file), the execution simply starts from the iSCRIPT file at the first line.

2.4.3. Project Files (*.ipr)

When a solution developed in iSCRIPT includes more than one iSCRIPT file, an iSCRIPT project file (*.ipr) is needed to aggregate all the files. The project file obeys the following rules:

(1) The file content must start with a unique keyword "project" on the first line.
(2) Both the name and directory of the script files should be included inside the project file. When the script files and the project file are in the same folder, the directory information of the script files may be omitted.
(3) At least one of the script files must contain a main program, which is described in 2.4.2.

An example of a project file is shown below. In the example below, no path information is included, as all the iSCRIPT files associated with the solution and listed in the project file are contained in the same folder.



```
project
OLS_pumps.isc
OLS_lines.isc
OLS_HX.isc
OLS_exergy.isc
OLS.isc
main.isc
```

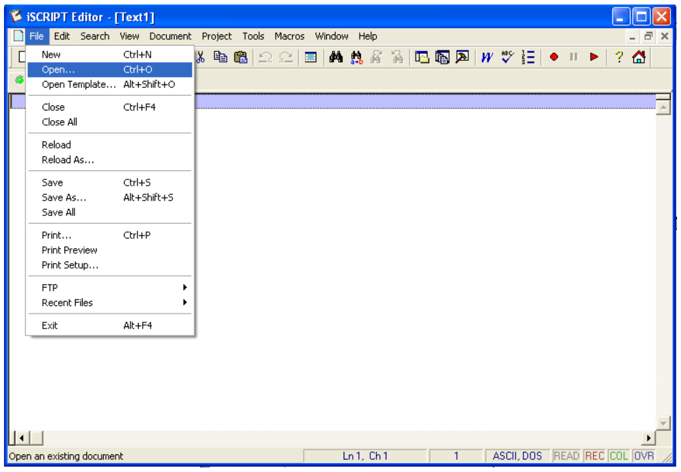
Figure 2.1. Sample project file.

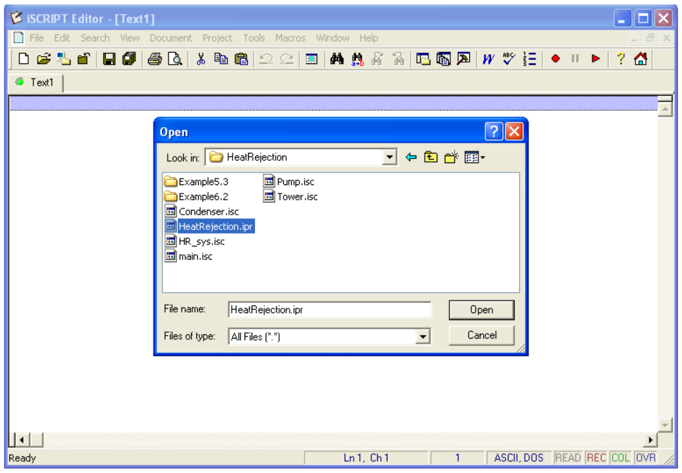
2.5. Running Sample Problems

Sample problems are provided for you in the */SampleScripts* subdirectory of the folder in which you installed iSCRIPT. They can be opened, edited, and run in iSCRIPT Editor environment. In this manual, several examples are provided and described in detail in Chapters 4 and 5. The procedure for launching the editing environment is described below.

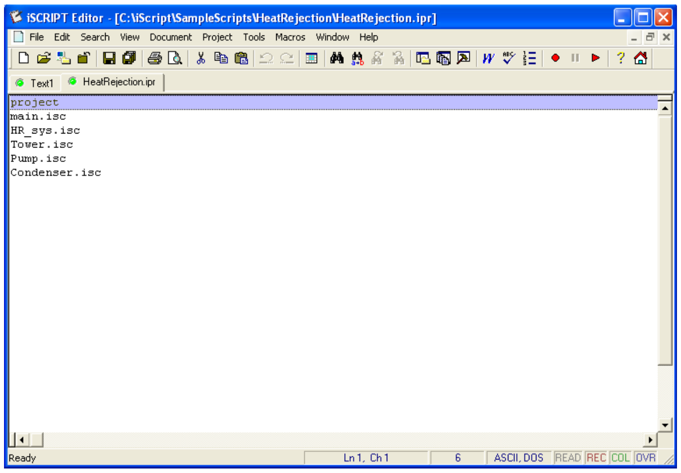
(1) Open iSCRIPT Editor.

(2) In iSCRIPT Editor, open the iSCRIPT project file *HeatRejection.ipr* located in the */SampleScripts/HeatRejection* subfolder of the iSCRIPT installation folder.



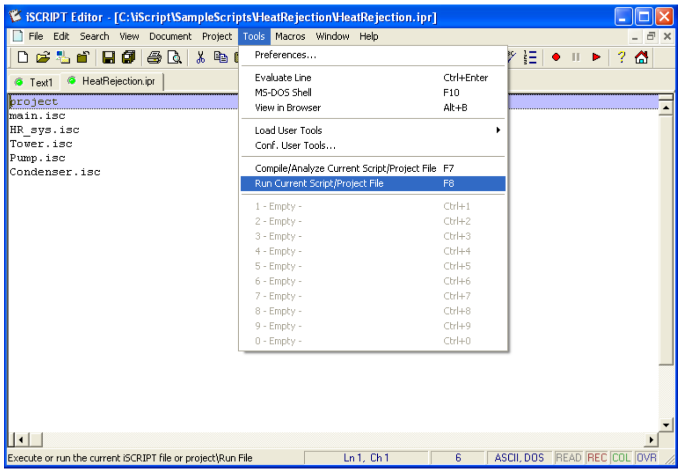


- The project file will open, as shown below.


iSCRIPT Editor - [C:\iScript\SampleScripts\HeatRejection\HeatRejection.ipr]
File Edit Search View Document Project Tools Macros Window Help
Text1
HeatRejection.ipr
Ready
Ln 1, Ch 1
6
ASCII, DOS
READ REC COL OVR

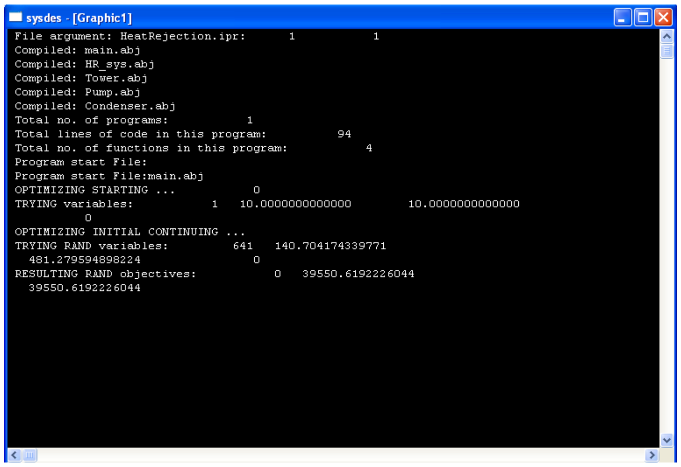
```
project
main.isc
HR_sys.isc
Tower.isc
Pump.isc
Condenser.isc
```

(3) On the toolbar, select “*Tools > Run Current Script/Project File*” to run the project file.



```
project
main.isc
HR_sys.isc
Tower.isc
Pump.isc
Condenser.isc
```

- The following screen will appear, containing the output of the program.



```
File argument: HeatRejection.ipr:      1           1
Compiled: main.abj
Compiled: HR_sys.abj
Compiled: Tower.abj
Compiled: Pump.abj
Compiled: Condenser.abj
Total no. of programs:           1
Total lines of code in this program:          94
Total no. of functions in this program:           4
Program start File:
Program start File:main.abj
OPTIMIZING STARTING ...           0
TRYING variables:          1   10.0000000000000        10.0000000000000
          0
OPTIMIZING INITIAL CONTINUING ...
TRYING RAND variables:         641   140.704174339771
  481.279594898224                0
RESULTING RAND objectives:           0   39550.6192226044
  39550.6192226044
```

2.6. Output Files

The results of a calculation are stored in the file *outputscript.txt* in the same folder of the project file, which contains the computation results of all component variables. You can also add your own I/O using iSCRIPT I/O commands. For projects involving optimization, additional information can be found in the file *optimize.txt*. The information includes the initial values of the optimization variables and the objective function, the various realizations of the system being evaluated, the array of viable systems or realizations (population of individuals in genetic algorithm parlance) by generation or as the optimization progresses, and the final optimized results.

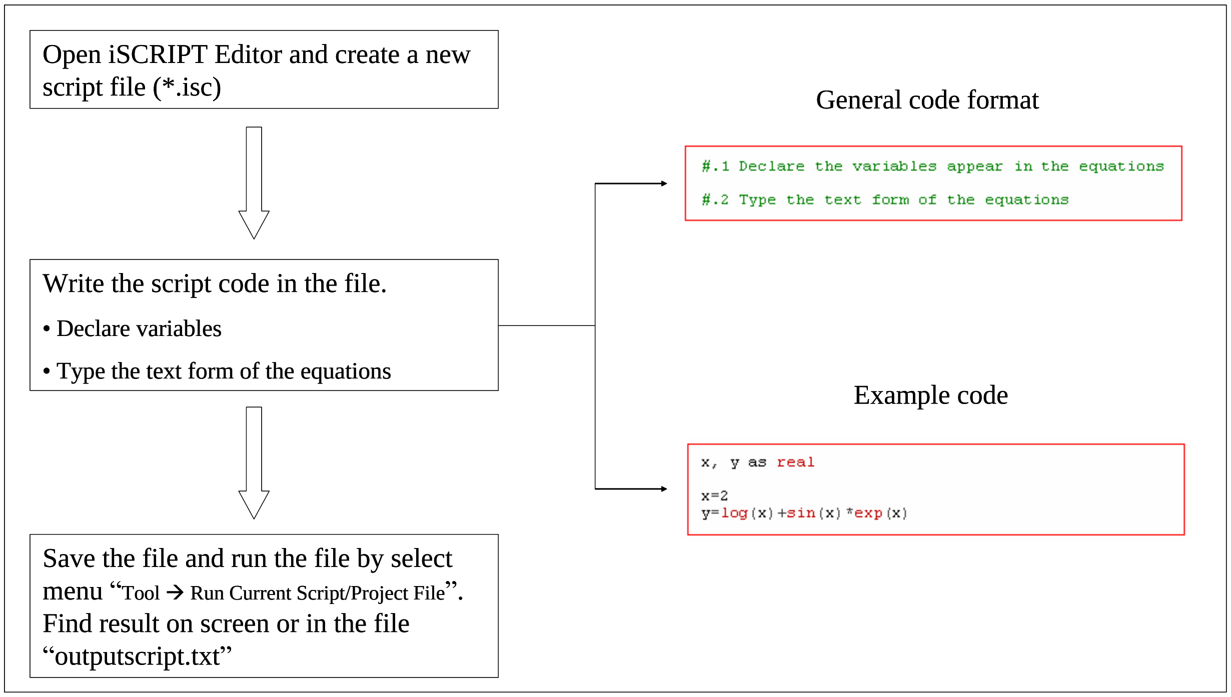
3. Developing and Executing an iSCRIPT Project

In this chapter, we introduce the most basic iSCRIPT procedures for solving an arithmetic and algebraic problem, evaluating the performance of an engineering system, and optimizing an engineering system.

3.1. Using iSCRIPT to Solve Basic Arithmetic and Algebraic Problems

iSCRIPT has a built-in functionality for arithmetic and algebraic operations. This enables iSCRIPT to perform most mathematic calculations, like some common symbolic computation software, such as Matlab, Maple, Mathematica, and Mathcad, and programming languages, such as FORTRAN and C.

In the current version of iSCRIPT, a math problem is solved by wring a small script. This is most conveniently done in the iSCRIPT Editor environment. A GUI is being developed that will allow the user to directly evaluate the values of mathematic equations in a command window.



Open iSCRIPT Editor and create a new script file (*.isc)
Write the script code in the file.
• Declare variables
• Type the text form of the equations
Save the file and run the file by select menu "Tool → Run Current Script/Project File". Find result on screen or in the file "outputscript.txt"
General code format
Example code

```
#.1 Declare the variables appear in the equations
#.2 Type the text form of the equations
```

```
x, y as real

x=2
y=log(x)+sin(x)*exp(x)
```

Figure 3.1. The basic procedure for solving an arithmetic and algebraic problem in iSCRIPT.

Figure 3.1 summarizes the basic procedure for solving an arithmetic and algebraic problem in iSCRIPT. The steps are also listed below:

Step 1. Open iSCRIPT Editor by selecting "iSCRIPT Editor" from the Windows "Start" menu.
Step 2. Create a new file in iSCRIPT Editor and save it as an *.isc file.
Step 3. Type the word "program *main*" in the first line of the file.
Step 4. Declare all the variables in the equations by typing the expressions like "*x as real,*" "*x, y as integer,*" etc. Variable types and their declaration in iSCRIPT can be found in Appendix A.1.1.
Step 5. Type the equations in text form, such as "*x=2+3,*" "*y=log(x)+sin(x)exp(x^2),*" etc. The mathematical operations supported in iSCRIPT can be found in Appendix A.1.
Step 6. Type the word "end program" in the last line of the file.
Step 7. Save the file.
Step 8. Select "*Tools > Run Current Script/Project File*" from the menu to run the iSCRIPT file.
Step 9. View the output on screen or in the file "*outputscript.txt*."

Note that Steps 3 and 6 may be omitted if you do not wish to adopt a program structure for your solution. However, you must adopt a program structure if you must use subroutines or functions in a solution. The details of these procedures are illustrated in sample problems in Chapters 4 and 5.

3.2. Using iSCRIPT for Performance Analysis of Engineering Systems

iSCRIPT uses a decomposition technique to simulate an engineering system. Outside of iSCRIPT, the system is first decomposed into several components, either physically, conceptually, or along disciplinary lines. The variables of each component as well as the constraints (engineering limits, material constraints, etc.) to the variables are then identified. An example could be the fluid inlet temperature into the Burner component of an aircraft engine. This variable may be constrained to temperatures less than 2800K for safe operation of the burner material. Operating variables are also identified, and are considered as system variables, as they do not belong to a particular component. An example could be the flight Mach number and altitude at which an aircraft engine is operating.

In iSCRIPT, each component is modeled in a separate subprogram (a piece of iSCRIPT code containing the equations of the component). The component routine may also access or call other functions, subprograms, or other component routines as part of its model. Interactions between a component and other components may also be included in a component subprogram. The entire system may be modeled by modeling (calling each component model) each of its components. The system model may also include interactions between the components of the system.

Figure 3.2 summarizes the basic procedures for performance analysis of an engineering system in iSCRIPT, using a simple gas exhaust system for illustration. The basic steps are also listed below:

Step 1. Open iSCRIPT Editor by selecting "iSCRIPT Editor" from the Windows "Start" menu.

Step 2. Create a new file in iSCRIPT Editor and save it as an *.isc file.

Step 3. Type the words "program *programname*" and "end program" to create a main program. In the body of the main program:

a. declare all the system's components and their variables. To declare a component, use the CreateComponent statement. To declare a component variable, use the CreateVariable statement
b. declare the overall system and its variable using CreateComponent and CreateVariable statements
c. assign the input values to the system or component variables by typing "ComponentName.VariableName = expression"
d. type "SystemName.Execute*"* to evaluate the overall system.

Step 4. After the main program is written, type words "subroutine ComponentName()*"* and "end subroutine" to create the component subprograms for each of the system components. In the body of the subroutine:

a. type all the equations modeling the component.

Step 5. After the component subroutines, type the keyword "subroutine SystemName()*"* and "end subroutine" to create the system subprogram for the overall system. In the body of the subroutine:

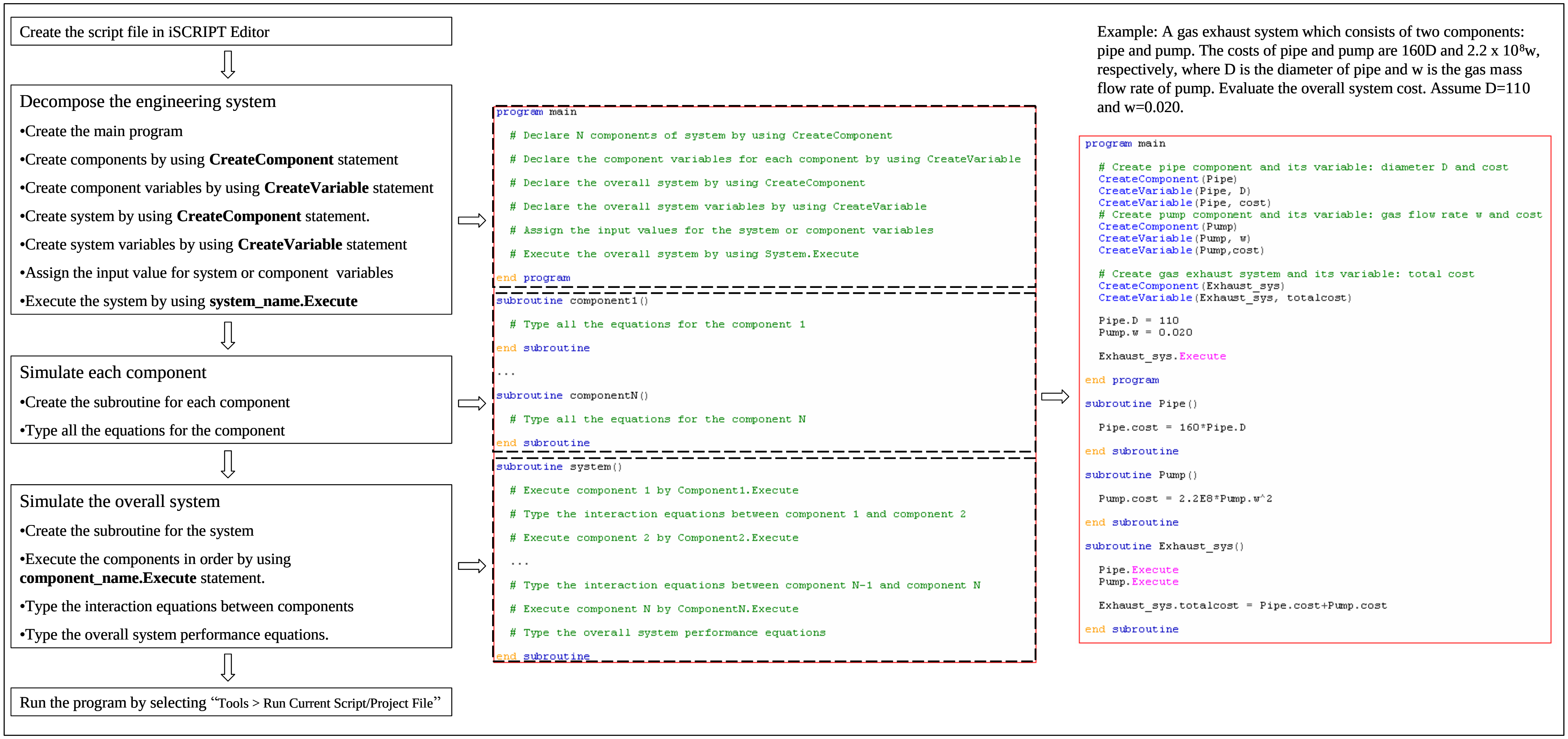
a. evaluate each component of the system by typing "ComponentName.Execute" in the system subprogram
b. if there are interactions between the components, type the interaction equations
c. type the overall system performance evaluation equations.

Step 6. Save the file in iSCRIPT Editor as an *.isc file.

Step 7. Select "*Tools >Run Current Script/Project File*" from the menu to run the iSCRIPT code.

Step 8. View the output on screen or in the file "*outputscript.txt*."

The details of these procedures are illustrated in sample problems in Chapter 5.



Create the script file in iSCRIPT Editor

Decompose the engineering system

- •Create the main program
- •Create components by using **CreateComponent** statement
- •Create component variables by using **CreateVariable** statement
- •Create system by using **CreateComponent** statement.
- •Create system variables by using **CreateVariable** statement
- •Assign the input value for system or component variables
- •Execute the system by using **system_name.Execute**

Simulate each component

- •Create the subroutine for each component
- •Type all the equations for the component

Simulate the overall system

- •Create the subroutine for the system
- •Execute the components in order by using **component_name.Execute** statement.
- •Type the interaction equations between components
- •Type the overall system performance equations.

Run the program by selecting "Tools > Run Current Script/Project File"

```
program main

  # Declare N components of system by using CreateComponent

  # Declare the component variables for each component by using CreateVariable

  # Declare the overall system by using CreateComponent

  # Declare the overall system variables by using CreateVariable

  # Assign the input values for the system or component variables

  # Execute the overall system by using System.Execute

end program
```

```
subroutine component1()

  # Type all the equations for the component 1

end subroutine

...

subroutine componentN()

  # Type all the equations for the component N

end subroutine
```

```
subroutine system()

  # Execute component 1 by Component1.Execute

  # Type the interaction equations between component 1 and component 2

  # Execute component 2 by Component2.Execute

  ...

  # Type the interaction equations between component N-1 and component N

  # Execute component N by ComponentN.Execute

  # Type the overall system performance equations

end subroutine
```

Example: A gas exhaust system which consists of two components: pipe and pump. The costs of pipe and pump are 160D and 2.2 x 10^8w, respectively, where D is the diameter of pipe and w is the gas mass flow rate of pump. Evaluate the overall system cost. Assume D=110 and w=0.020.

```
program main

  # Create pipe component and its variable: diameter D and cost
  CreateComponent(Pipe)
  CreateVariable(Pipe, D)
  CreateVariable(Pipe, cost)
  # Create pump component and its variable: gas flow rate w and cost
  CreateComponent(Pump)
  CreateVariable(Pump, w)
  CreateVariable(Pump,cost)

  # Create gas exhaust system and its variable: total cost
  CreateComponent(Exhaust_sys)
  CreateVariable(Exhaust_sys, totalcost)

  Pipe.D = 110
  Pump.w = 0.020

  Exhaust_sys.Execute

end program

subroutine Pipe()

  Pipe.cost = 160*Pipe.D

end subroutine

subroutine Pump()

  Pump.cost = 2.2E8*Pump.w^2

end subroutine

subroutine Exhaust_sys()

  Pipe.Execute
  Pump.Execute

  Exhaust_sys.totalcost = Pipe.cost+Pump.cost

end subroutine
```

Figure 3.2. The basic procedure for performance analysis of an engineering system in iSCRIPT.

3.3. Using iSCRIPT to Optimize an Engineering System

iSCRIPT's built-in procedures can automatically optimize an engineering system that is formulated in component decomposition form, as illustrated in Section 3.2. The way to optimize an engineering system (assuming a single objective variable) is similar to that for performance analysis, except:

- an objective variable must be indicated, in addition to the requirements for system performance evaluation
- optimization variables must be indicated, in addition to the requirements for system performance evaluation
- indicate the relations between the system components and the system
- "System.Optimize" command is used instead of "System.Execute."

Figure 3.3 summarizes the basic procedures required to optimize an engineering system in iSCRIPT. Note that the objective variable (the variable that we want to optimize) is declared by using the statement "AddObjective." The optimization variables (the variables that are free to be changed during the optimization) are declared by using statement "AddVarObjective." The basic steps are listed below:

Step 1. Open iSCRIPT Editor by selecting "iSCRIPT Editor" from the Windows "Start" menu.

Step 2. Create a new file in iSCRIPT Editor and save it as an *.isc file.

Step 3. Type the words "program programname" and "end program" to create a main program structure. In the body of the main program:

a. declare all the system's components and their variables. To declare a component, use the CreateComponent statement. To declare a component variable, use the CreateVariable statement.
b. declare the overall system and its variable using CreateComponent and CreateVariable statements.
c. assign the input values to the system or component variables by typing "ComponentName.VariableName = expression"
d. declare the objective variable by using "AddObjective" statements
e. declare the optimization variables by using "AddVarObjective" statements
f. indicate the relations between the system components and the system by using "AddSubComponent" statements
g. type SystemName.Optimize to optimize the overall system.

Step 4. After the main program, type keywords "subroutine ComponentName()" and "end subroutine" to create the component subroutines for each component. In the body of the subroutine:

a. type all the equations modeling the component.

Step 5. After the component subroutines, type the word "subroutine SystemName()" and "end subroutine" to create the system subroutine. In the body of the subroutine:

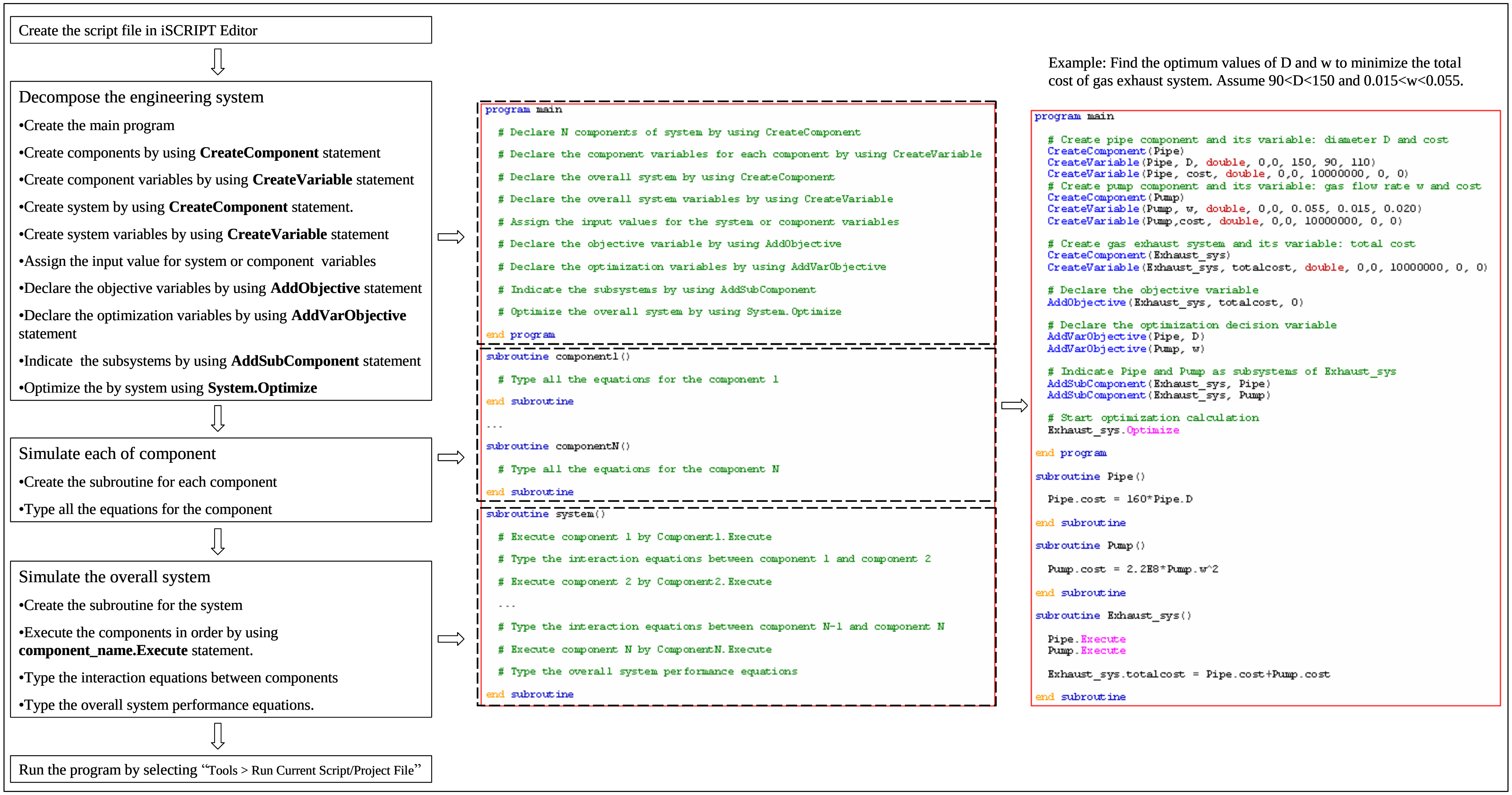
   a. evaluate each of the system's components by typing "ComponentName.Execute"
   b. if there are interactions between the components, type the interaction equations between the components
   c. type the overall system performance evaluation equations.

Step 6. Save the file.

Step 7. Select "*Tools > Run Current Script/Project File*" from the menu to run the iSCRIPT code.

Step 8. View the output on screen or in the file "*outputscript.txt*."

The details of these procedures are illustrated in sample problems in Chapter 5.



Create the script file in iSCRIPT Editor

Decompose the engineering system

•Create the main program

•Create components by using **CreateComponent** statement

•Create component variables by using **CreateVariable** statement

•Create system by using **CreateComponent** statement.

•Create system variables by using **CreateVariable** statement

•Assign the input value for system or component variables

•Declare the objective variables by using **AddObjective** statement

•Declare the optimization variables by using **AddVarObjective** statement

•Indicate the subsystems by using **AddSubComponent** statement

•Optimize the by system using **System.Optimize**

Simulate each of component

•Create the subroutine for each component

•Type all the equations for the component

Simulate the overall system

•Create the subroutine for the system

•Execute the components in order by using **component_name.Execute** statement.

•Type the interaction equations between components

•Type the overall system performance equations.

Run the program by selecting "Tools > Run Current Script/Project File"

```
program main

  # Declare N components of system by using CreateComponent

  # Declare the component variables for each component by using CreateVariable

  # Declare the overall system by using CreateComponent

  # Declare the overall system variables by using CreateVariable

  # Assign the input values for the system or component variables

  # Declare the objective variable by using AddObjective

  # Declare the optimization variables by using AddVarObjective

  # Indicate the subsystems by using AddSubComponent

  # Optimize the overall system by using System.Optimize

end program
```

```
subroutine component1()

  # Type all the equations for the component 1

end subroutine

...

subroutine componentN()

  # Type all the equations for the component N

end subroutine
```

```
subroutine system()

  # Execute component 1 by Component1.Execute

  # Type the interaction equations between component 1 and component 2

  # Execute component 2 by Component2.Execute

  ...

  # Type the interaction equations between component N-1 and component N

  # Execute component N by ComponentN.Execute

  # Type the overall system performance equations

end subroutine
```

Example: Find the optimum values of D and w to minimize the total cost of gas exhaust system. Assume 90<D<150 and 0.015<w<0.055.

```
program main

  # Create pipe component and its variable: diameter D and cost
  CreateComponent(Pipe)
  CreateVariable(Pipe, D, double, 0,0, 150, 90, 110)
  CreateVariable(Pipe, cost, double, 0,0, 10000000, 0, 0)
  # Create pump component and its variable: gas flow rate w and cost
  CreateComponent(Pump)
  CreateVariable(Pump, w, double, 0,0, 0.055, 0.015, 0.020)
  CreateVariable(Pump,cost, double, 0,0, 10000000, 0, 0)

  # Create gas exhaust system and its variable: total cost
  CreateComponent(Exhaust_sys)
  CreateVariable(Exhaust_sys, totalcost, double, 0,0, 10000000, 0, 0)

  # Declare the objective variable
  AddObjective(Exhaust_sys, totalcost, 0)

  # Declare the optimization decision variable
  AddVarObjective(Pipe, D)
  AddVarObjective(Pump, w)

  # Indicate Pipe and Pump as subsystems of Exhaust_sys
  AddSubComponent(Exhaust_sys, Pipe)
  AddSubComponent(Exhaust_sys, Pump)

  # Start optimization calculation
  Exhaust_sys.Optimize

end program

subroutine Pipe()

  Pipe.cost = 160*Pipe.D

end subroutine

subroutine Pump()

  Pump.cost = 2.2E8*Pump.w^2

end subroutine

subroutine Exhaust_sys()

  Pipe.Execute
  Pump.Execute

  Exhaust_sys.totalcost = Pipe.cost+Pump.cost

end subroutine
```

Figure 3.3. General procedures for optimizing an engineering system in iSCRIPT.

4. Using iSCRIPT in Simple Programs

In this chapter, we will introduce the most basic iSCRIPT statements and use them to write some simple programs. The simplicity of the sample problems does not limit the usefulness of the basic iSCRIPT procedures.

4.1. Getting Started

Using iSCRIPT for the first time is easy. In this and the following sections, we will introduce you to the iSCRIPT Editor environment and show you how to develop and run a simple iSCRIPT code.

To start iSCRIPT Editor, use the “Start” menu to locate the program. The default iSCRIPT Editor screen, which opens each time you start the program, is shown in Figure 4.1.

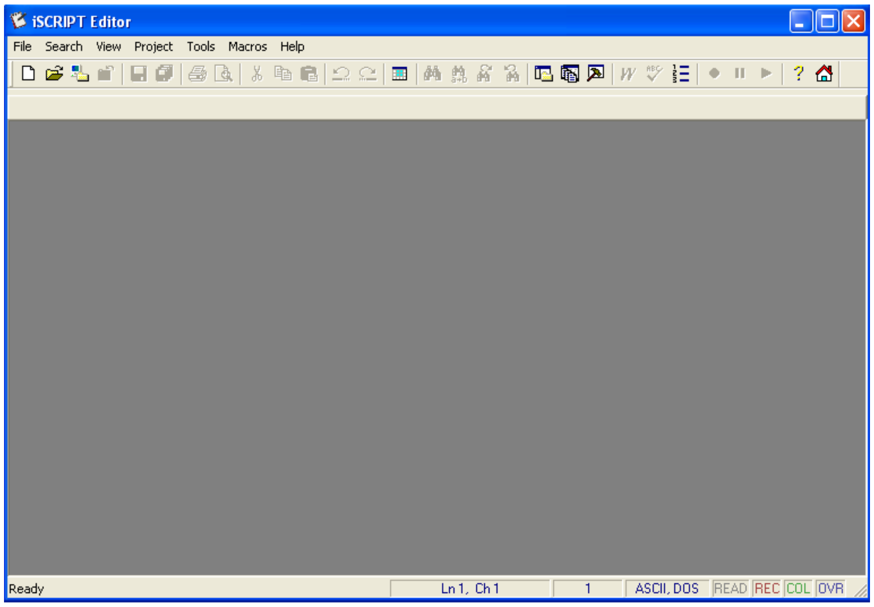


Figure 4.1. iSCRIPT Editor opening window.

The iSCRIPT Editor opening window is similar to most text-editing or program development environments. Detail editor commands may be accessed by clicking “*Help*” on the menu. However, only the basic command menus required for creating and editing iSCRIPT codes are presented in this manual. These menu items are presented below.

- iSCRIPT file operations
    - To create a new code file, you can click the new (blank) document icon or select "*File > New*" from the menu.
    - To open an existing code file, you can click the open file icon or select "*File > Open*" from the menu to open a file opening dialog box. Change the current file directory and select the file you want to open.
    - To save a code file, you can click the save file icon or select "*File > Save*" from the menu.
- Operations for running iSCRIPT
    - To run iSCRIPT, you can select "*Tools > Run Current Script/Project FIle*" from the menu or use the hot key *F8*.
    - To analyze a code, you can select "*Tools > Compile/Analyze Current Script/Project File*" from the menu or use the hot key *F7*.

    Note: The "*ISCRIPT Analyzer*" command outputs detailed information related to the results of evaluating every line of an iSCRIPT code for debugging purposes, while "*ISCRIPT*" command does not provide detailed output. For this reason, "*ISCRIPT*" command executes faster than the Analyzer.

4.2. A Simple iSCRIPT Program: Calculate "2+3"

In this section, we will introduce the steps required to develop a script for an arithmetic and algebraic problem. We use the simple arithmetic problem "*2+3=?*" as our first example. We will write a short iSCRIPT program, which we call "main," to solve this problem. The procedure follows the general procedure outlined for arithmetic and algebraic problems in Section 3.1. The steps involved in solving this problem are listed below.

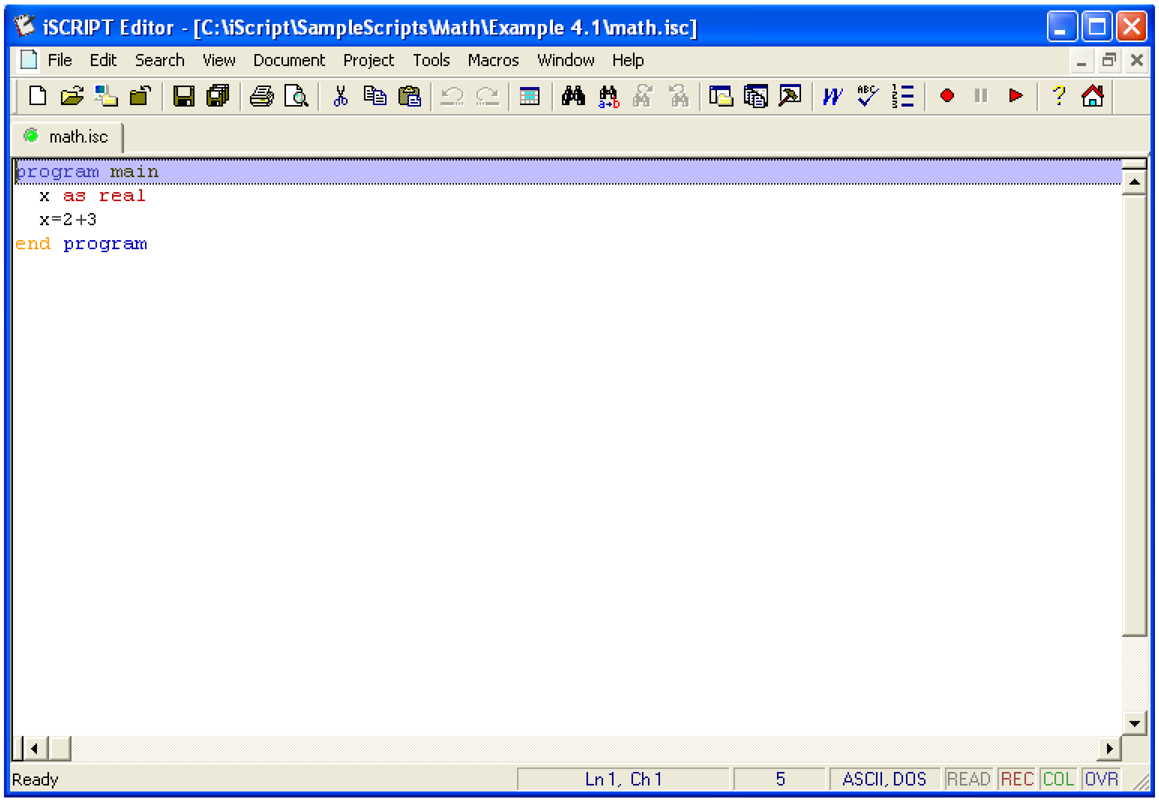
Problem 4.1: Calculate "2+3" in iSCRIPT

4.2.1. Using a Program Structure

1. Open iSCRIPT Editor by selecting the program from the Windows "Start" menu.
2. Create a new file by selecting "*File > New*" from the menu and save it as an "*.isc" file (e.g., *math.isc*).

3. In the first line of the file, type the word "program main."
4. In line 2, declare a real number variable *x* by typing "*x* as real."
5. In line 3, type the equation "x=2+3."
6. In line 4, type the word "end program."
7. Run the program by selecting "*Tools > Run Current Script/Project File*" from the menu.
8. After the program is run, the results can be viewed on the screen or in the file "*outputscript.txt*" located in the same folder in which you saved the "*math.isc*" file.

The complete script is presented in Figure 4.2. You can also find the script file "*math.isc*" in the /*SampleScripts/Math/Example4.1* subfolder of the iSCRIPT installation folder.


iSCRIPT Editor - [C:\iScript\SampleScripts\Math\Example 4.1\math.isc]
File Edit Search View Document Project Tools Macros Window Help
math.isc
Ready
Ln 1, Ch 1
5
ASCII, DOS
READ REC COL OVR

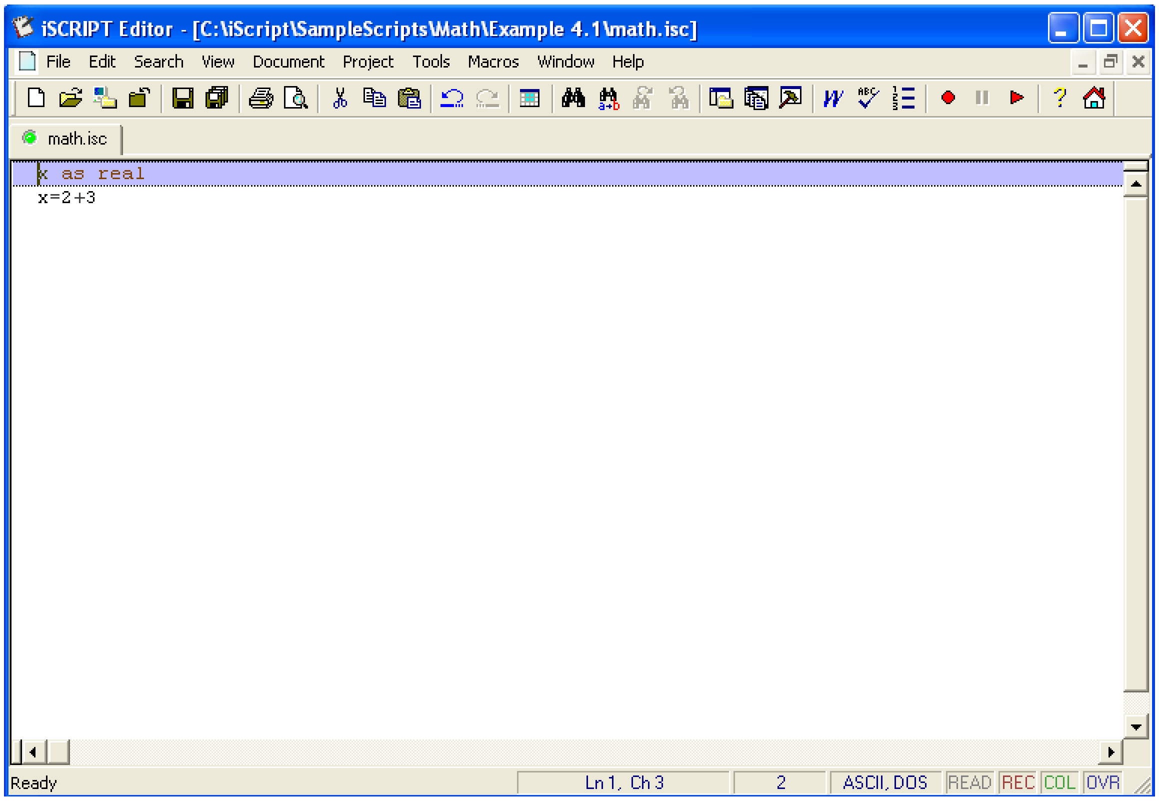
```
program main
  x as real
  x=2+3
end program
```

Figure 4.2. iSCRIPT code for the computation of "2+3" using a program structure.

Let's take a quick took at the code. The first and last lines define a program that has the name "*main*." The second line declares a real variable "*x*." The third line evaluates the expression "*2+3*" and assigns the result to "*x*." An output statement is not required since iSCRIPT will automatically output the declared variable.

4.2.2. Without a Program Structure

Follow the same steps as in 4.2.1. However, in the current case, omit steps 3 and 6. The resulting program should be as shown in Figure 4.3.


iSCRIPT Editor - [C:\iScript\SampleScripts\Math\Example 4.1\math.isc]
File Edit Search View Document Project Tools Macros Window Help
math.isc
Ready
Ln 1, Ch 3
2
ASCII, DOS
READ REC COL OVR

```
x as real
x=2+3
```

Figure 4.3. iSCRIPT code for the computation of “2+3” without a program structure.

Table 4.1 gives a few sample mathematical expressions and their corresponding iSCRIPT codes.

Math Expression	**iSCRIPT Code**

$x=(2.5-1)*\sqrt{3}$	`x as real`
	`x=(2.5-1)*3.0^0.5`

$x=2$
$y=\sin xe^{x}+\ln(x+1)+\log_{10}x$
	`x,y as real`
	`x=2`
	`y=sin(x)*exp(x)+log(x+1)+log10(x)`

$A=\begin{bmatrix}1.2 & -0.3\\ 0.7 & 2.5\end{bmatrix}$
$B=\begin{bmatrix}5.0 & 7.7\\ -12.5 & -2.3\end{bmatrix}$
$C=AB$
	`A(2,2),B(2,2),C(2,2) as real`
	`A=[1.2,-0.3;0.7,2.5]`
	`B=[5.0,7.7;-12.5,-2.3]`
	`C=A*B`

Table 4.1. Sample mathematical expressions and their iSCRIPT equivalents.

4.3. Another Simple Problem: Aircraft Drag Calculation

Let's consider another simple arithmetic problem, involving aircraft drag calculation. The problem is described below:

Problem 4.2: Aircraft Drag Calculation

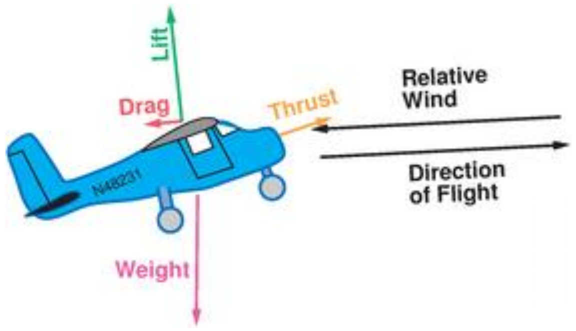


Figure 4.4 Aerodynamics Forces on an Aircraft

In general, the aircraft drag can be computed by

$$drag = C_d \frac{\rho V^2 A}{2} \quad , \tag{4.1}$$

where

C_d = drag coefficient, which is usually determined experimentally, in a wind tunnel,
ρ = air density,
V = velocity of aircraft,
A = reference area (the surface area over which the air flows)

Although the drag coefficient is not a constant, it can be assumed to be at low speeds (less than 200 mph). Suppose the following data were measured in a wind tunnel:

drag	*20,000* N
ρ	1×10^{-6} kg/m^3
V	*100* mph
A	*1* m^2

Calculate the drag coefficient, then use the computed drag coefficient to predict how much drag force will be exerted on the aircraft at velocities from 0 to 200 mph.

To solve this problem in iSCRIPT, we need to write a program that computes the drag coefficient C_d from the wind tunnel data. The drag coefficient is then used to calculate the drag force on the aircraft at a range of velocities between 0 and 200 mph.

As for the math problem in Section 4.2, the script may be created with a program structure. In iSCRIPT, the program structure starts with a keyword "program *programname*" and ends with a keyword "end program." The format of the program is:

```
program programname

        …


end program
```

… represents one or more lines of scripting language segments and is referred to as the body of the program.

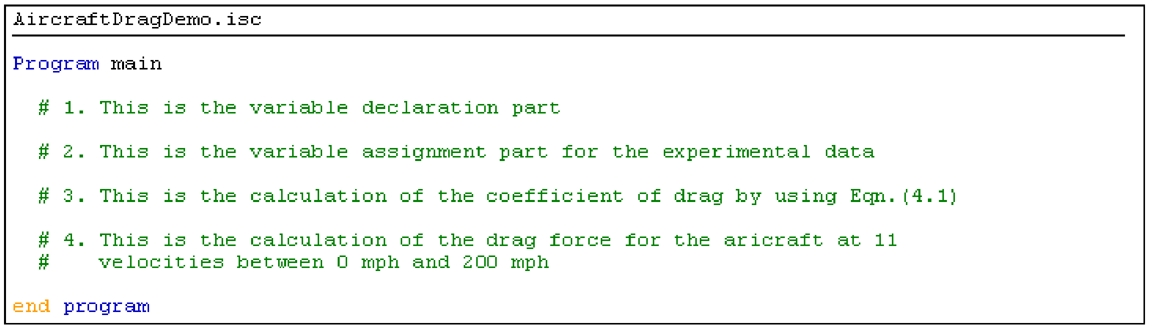
The first few lines of the program should consist of the variable declaration. In iSCRIPT, all of the local variables (with the data type of variable) should be declared before any executable statements. The format of data declaration in iSCRIPT is:

var1, *var2* as type

var1, *var2* – variable names satisfying the variable naming convention.
as – declaration keyword
type – may take the values: logical, short, long, real, and double (see Appendix A.1.1)

After the variable declaration, the input data should be assigned to the variables. More details about variable names, variable types, and permissible expressions can be obtained in the iSCRIPT Language Reference in Appendix A.

With the experimental data assigned, the drag coefficient can be calculated by using Equation 4.1. Let us calculate the drag force on the aircraft for 11 velocities evenly distributed between 0 and 200 mph. The corresponding drag force results and the velocities can be stored in two different arrays. Note that iSCRIPT supports array operations and most of the intrinsic functions directly support vector and matrix operations. More information on working with arrays is provided in the iSCRIPT Language Reference in Appendix A.



```
AircraftDragDemo.isc

Program main

  # 1. This is the variable declaration part

  # 2. This is the variable assignment part for the experimental data

  # 3. This is the calculation of the coefficient of drag by using Eqn.(4.1)

  # 4. This is the calculation of the drag force for the aricraft at 11
  #    velocities between 0 mph and 200 mph

end program
```

Figure 4.5. Program outline for problem 4.2.

Figure 4.5 shows the program outline for this problem. You may type this iSCRIPT code yourself in iSCRIPT Editor and replace the comment statements of steps 1 through 4 with the appropriate scripts.

(Note: In iSCRIPT, any line starting with “#” or “%” is a comment line. Any part of a line starting with “#” or “%” not within a string quote ‘’ is also a comment. Comments are not evaluated and are provided only for the convenience of the modeler to communicate details of the model or script to themselves or others.)

While writing the code, you may need to use the decision structure and loop structure. iSCRIPT supports decision and loop statements of most popular programming languages. For example, the most common decision statement, the *if* statement, can be used in iSCRIPT as

```
if (expression) then
 …
else
 …
end if
```

“…” represents one or more lines of scripting language segment
If *expression* is true, the first “…” part will be executed. Otherwise, the second “…” part will be executed

The most common loop statement, the *do* statement, can be used in iSCRIPT as

```
do ii = expression1 : expression2
 …

end do
```

"…" represents one or more lines of scripting language segment.
ii is incremented by 1 and the body of the loop executed until *expression1* is greater than *expression2*.

Other loop and decision structures can be found in the iSCRIPT Language Reference in Appendix A.

The complete program for Problem 4.2 is shown in Figure 4.6. If you compare this code with Figure 4.5, you can understand the meaning of each part of the code. The code can be found in the subfolder */SampleScripts/AircraftDrag/Example4.2* of the iSCRIPT installation folder.

```
AirplaneDrag.isc

Program main

  drag, rho, V, A as real
  Cd as real
  V_res(11), drag_res(11) as real
  i as integer

  drag = 20000
  rho = 1.0e-6
  V = 100*0.4470
  A = 1.0

  Cd = 2*drag/(rho*V^2*A)

  do i=1:11
    V_res(i)=200*0.4470/10*(i-1)
    drag_res(i) = Cd*rho*V_res(i)^2*A/2
  end do

end program
```

Figure 4.6. An iSCRIPT program for evaluating the drag on an aircraft at a range of low speeds.

To run this code, you can carry out the following steps:

1. Open iSCRIPT Editor by selecting the program from the Windows "Start" menu.
2. Select "*File* > *Open*" from the menu.
   ⇒ The File Open dialog box appears.
3. Navigate to the subfolder */SampleScripts/AircraftDrag* of the iSCRIPT installation folder.
4. Open the file *AircraftDrag.isc*.
5. Select "*Tools* > *Run Current Script/Project File*" from the menu.
6. The results will be shown on the screen when the program is completed. You can also find the result in *outputscript.txt* in the same folder as the script file.

To create this code, you can follow the following procedures in Section 4.2, but type in the program instructions as shown in Figure 4.6.

1. Open iSCRIPT Editor by selecting the program from the Windows “Start” menu.
2. Create a new file by selecting “*File > New*” from the menu and save it as an “*.isc” file (e.g., *AircraftDrag.isc*).
3. On the first line of the file, type the word “program main.”
4. On line 2, declare a real number variable *x* by typing “*x* as real.”
5. On line 3, type the instructions as shown in Figure 4.5.
6. On line 4, type the word “end program.”
7. Run the program by selecting “*Tools > Run Current Script/Project File*” from the menu.
8. After the program is run, the results can be viewed on the screen or in the file “*outputscript.txt*,” which is located in the same folder in which you saved the “*AircraftDrag.isc*” file.

The results of the calculation are:

Cd	$2.0019 \text{ X } 10^7$										
V_res (m/s)	0	8.94	17.88	26.82	35.76	44.70	53.64	62.58	71.52	80.46	89.40
Drag_res (N)	0	800	3200	7200	12800	20000	28800	39200	51200	64800	80000

<u>Analysis of the lines of the program</u>

```
3:   drag, rho, V, A as real
4:   Cd as real
5:   V_res(11), drag_res(11) as real
6:   i as integer
```

Lines 3 through 6 declare the variables that are used in the program. Accepted variable types include logical, short, integer, real, and double. An exhaustive list can be found in the iSCRIPT Language Reference in Appendix A. Also notice that variables *V_res* and *drag_res* are declared as real arrays intended to accommodate 11 real values.

```
 8:   drag = 20000
 9:   rho = 1.0e-6
10:   V = 100*0.4470
11:   A = 1.0
```

Lines 8 through 11 are called assignment statements. These assign specific values to the variables.

```
13:   Cd = 2*drag/(rho*V^2*A)
```

Line 13 represents the calculation of the drag coefficient.

```
15:   do i=1:11
16:     V_res(i)=200*0.4470/10*(i-1)
17:     drag_res(i) = Cd*rho*V_res(i)^2*A/2
18:   end do
```

Lines 15 through 18 represent the calculation of the drag for 11 speeds ranging from 0 to 200 mph. Notice the conversion of the units of speed to SI units (m/s) by multiplying by the factor 0.4470.

4.4. Another Arithmetic Problem: Circular Cylinder Geometry

We choose the “circular cylinder surface area and volume calculation” problem as another example to illustrate the use of iSCRIPT for a simple arithmetic problem. In this problem, we will describe how to use the subroutine and function structures in iSCRIPT. The problem is described as follows:

Problem 4.3: Calculate the Surface Area and Volume of a Circular Cylinder

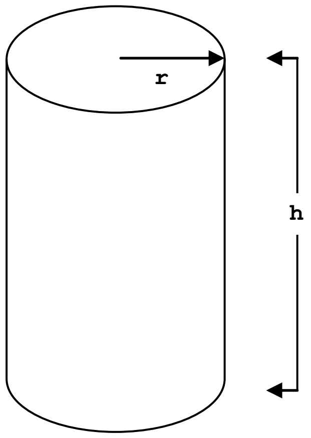


Figure 4.7. Physical problem and its primary variables.

The equations for calculating the surface area *S* and volume *V* of a circular cylinder with radius *r* and height *h* are

$$S = 2\pi r^2 + 2\pi rh \tag{4.2}$$

$$V = \pi r^2 h \quad , \tag{4.3}$$

where $r = 5$ and $h = 10$.

We will present a crash course in subroutines and functions in this section, so that we can use them for the sample calculations being discussed. Please refer to Appendix A.5.2 through A.5.5 for more details on subroutines and functions in iSCRIPT. iSCRIPT supports the use of subroutines and functions. In iSCRIPT, a subroutine may be written as

```
subroutine subroutinename (arg1, arg2, …, argN)
   variable declaration statements
```

```
      …

  end subroutine
```

"…" represents one or more lines of scripting language segments
arg1, *arg2*, …, *argN* are the arguments to the subroutine.
variable declaration statements represent variable declaration statements for the arguments.

A subroutine may be called by

```
  call subroutinename (arg1, arg2, …, argN)
```

where *arg1*, *arg2*, …, *argN* are the actual arguments to the subroutine.

Similar to a subroutine, a function may be written as

```
  function functionname (arg1, arg2, …, argN)
      variable declaration statements
      …


  end function
```

"…" represents one or more lines of scripting language segment,
arg1, *arg2*, …, *argN* are the arguments to the function.
variable declaration statements represent variable declaration statements for the arguments.

Functions may be called simply by using them in an expression in lieu of a variable.

The statement

```
  var = functionname (arg1, arg2, …, argN)
```

where *arg1*, *arg2*, …, *argN* are the actual arguments to the function, assigns the value of the function to the variable "var."

To solve this problem, it is convenient to create a function (*get_area*) to calculate the surface area of the cylinder, and a subroutine (*get_volume*) to calculate the volume of cylinder. The complete code is shown in Figure 4.8. The calls to the function *get_area* and the subroutine *get_volume* are in lines 9 and 11 of the main program. Lines 15 to 24 define the *get_area* function. Note that the type of the function is declared on line 17 (*get_area as real*). This is required when writing a function in iSCRIPT. Lines 26 to 35 define the *get_volume* subroutine. Note that the return value of (*volume*) must be an argument of the subroutine. This value will be computed when the program is executed and the computed value will be returned to the main program.

Note:

- For the subroutine/function, the names of the dummy arguments and the actual arguments do not have to be same, but they are required to have the same type.
- In iSCRIPT, at runtime, the values of the actual arguments, if modified within the subroutine, are returned to the calling program or subprogram.
- After control is returned to the calling routine, the local variables in the subroutine or function are automatically freed.

```
CircularCylinder.isc

 1: program main
 2:
 3:   radius, height, surface_area, volume as real
 4:   pi as real
 5:
 6:   radius = 5
 7:   height = 10
 8:
 9:   surface_area = get_area(radius, height)
10:
11:   call get_volume(volume, radius, height)
12:
13: end program
14:
15: function get_area(r,h)
16:
17:   get_area, r, h as real
18:   pi as real
19:
20:   pi = 3.1415926535
21:
22:   get_area = 2*pi*r^2 + 2*pi*r*h
23:
24: end function
25:
26: subroutine get_volume(v,r,h)
27:
28:   v, r, h as real
29:   pi as real
30:
31:   pi = 3.1415926535
32:
33:   v = pi*r^2*h
34:
35: end subroutine
```

Figure 4.8. iSCRIPT program for evaluating the cylinder geometry.

This code can be found in the subfolder /*SampleScripts/CircularCylinder/Example4.3* of the iSCRIPT installation folder. Follow these steps to run the code:

Step 1. Open iSCRIPT Editor.
Step 2. Select “*File* > *Open*” from the menu.
⇒ The File Open dialog box appears.
Step 3. Navigate to the subfolder /*SampleScripts/CircularCylinder/Example4.3* of the iSCRIPT installation folder.
Step 4. Open the file *CircularCylinder.isc*.
Step 5. Select “*Tools* > *Run Current Script/Project File*” from the menu.

The results for the surface area and volume are 471.2389 and 785.3982, respectively.

5. Engineering Component Modeling in iSCRIPT

In this chapter, we present the component modeling technique in iSCRIPT. Examples are given to show the procedure for creating and executing engineering component modeling mode in iSCRIPT. The procedure for developing an iSCRIPT solution in several script files is also presented.

5.1. Component Modeling in iSCRIPT

iSCRIPT has features that allow you to define components and the variables associated with the components. These variables will have an engineering context including limits (constraints), as well as engineering units, where appropriate. To complete the modeling of a component, a component subprogram must be written that contains the equations to model the component. The model equations for a component can be evaluated with the statement Component.Execute.

In iSCRIPT, a component is declared by using a *CreateComponent* statement. All components of a system must be declared in the main program.

CreateComponent (*component_name* [,*description*])

Note: Segments enclosed in square brackets are optional and may be omitted.

component_name – A name for the component (a string limited to 24 characters). Two components may not have the same name. Component names obey the same formation rule as those for variables.

description – A description for the component (a string limited to 50 characters). Optional.

After the component is declared, variables can be attached to this component by using a *CreateVariable* statement.

CreateVariable (*component_name*, *variable_name* [,*type*] [,*dimension*] [,*size*] [,*upper_bound*] [,*lower_bound*] [,*default_value*] [,*unit*])

Note: Segments enclosed in square brackets are optional and may be *omitted.*

component_name –	The component to which the variable belongs (a string limited to 24 characters). Two component variables may not have the same name. Component variable names obey the same formation rule as those for variables.
variable_name –	A name for the variable (a string limited to 24 characters). Two components may not have the same name. Component names obey the same formation rule as those for variables.
type –	A string accepting values such as "integer," "real," "double." A complete list of variable types can be found in Appendix A.1. This argument is optional. When not provided, component variables are assumed to be double values.
dimension –	Variable dimension for an array variable (integer). For example, a 2D matrix will have a dimension of 2. This argument is optional for scalar variables (*dimension* = 0 is default).
size –	Size for an array variable. This argument accepts a group of integers in a bracket separated by a semicolon ";" with a limit of five integers. For example, a 3 x 3 matrix will have a size of (3;3). This input is required when dimension > 0.
upper_bound –	An upper bound for the variable (all the variables for an array variable). This argument is optional.
lower_bound –	A lower bound for the variable (all the variables for an array variable). This argument is optional.
default_value –	A default value for the variable (all the variables for an array variable). This argument is optional.
unit –	A string representing the engineering unit for the variable (e.g. m/s). This argument is optional and is should be limited to 20 characters, if provided.

A component may have several variables. A component variable can be referenced by using:

```
ComponentName.VariableName

Example:

        Cylinder.radius = 5

        Cylinder.height = 10

        Cylinder.Area = 2*pi*(Cylinder.radius+Cylinder.height)
```

A component variable can be used and its value can be changed in any program, subprogram, or component.

Each component must have a unique subroutine that has the same name as the component name. The component subroutine requires no argument. This subroutine is automatically executed once the Execute command is called:

```
Component_Name.Execute

or

Call Component_Name.Execute
```

The component may also be optimized by using the Optimize command as follows:

```
Component_Name.Optimize

or

Call Component_Name.Optimize
```

With the above statements, iSCRIPT provides a component modeling method for the design and optimization of a complicated system. A large system can be decomposed into several components and each component is modeled in its own executable subprogram. We will give an example in the next section to illustrate how to use this component modeling technique to solve a real problem.

5.2. Creating Subsystems and Systems in iSCRIPT

Conceptually, subsystems have the same data structure and properties as components. They include variables at the subsystem level similar to component variables. The subsystem variables are variables at the subsystem level that cannot be isolated within any of the components. For instance, in an airframe subsystem consisting of components including, say, a fuselage, wind, tail, and ailerons, the total surface area as a variable would be a subsystem variable. On the other hand, the wing span, sweep angle and thickness would be the wing component's variables and the fuselage length and diameter would be the fuselage component's variables. Subsystems are also declared using the CreateComponent command, and subsystem variables are also declared using the CreateVariable command. The way subsystems differ from components is that they have other components associated within them. The association is indicated using the AddSubComponent command. However, this formal association is only necessary for optimization tasks (Chapter 6 and Problem 6.2 of Section 6.3). The mere evaluation of the subsystem model consisting of the evaluation of its component models is sufficient for the performance analysis of the subsystem (see Figure 5.12 of Section 5.6).

Systems do not need to be declared in iSCRIPT. By default, each iSCRIPT project is considered to be a system. All subsystems (and their components) declared in a specific project are therefore assumed to belong to the system. Problem 5.3 illustrates the relationship between components, subsystems, and systems in iSCRIPT.

5.3. Solving a Problem Using Component Modeling or Decomposition Method

We still use the "circular cylinder surface area and volume calculation" problem of Section 4.4 to illustrate the procedure. We will now solve that same problem using the component method.

Problem 5.1: Solve Problem 4.4 Using the Component Technique in iSCRIPT

For this problem, we define a single component *Cylinder*, which has four component variables: *radius, height, surface_area*, and *volume*. The detailed procedure to solve the problem follows the general procedure in Section 3.2. The steps are as follows:

Step 1. Open iSCRIPT Editor.
Step 2. Create a new file and save it as an "*.isc" file.
Step 3. Type the words "program *main*" and "*end program*" to create a main program. In the body of the main program:
   a. declare the *Cylinder* component by using CreateComponent statement

b. declare the four component variables (*radius*, *height*, *surface_area*, *volume*) of the *Cylinder* system using CreateVariable statement
c. assign the input values for *Cylinder.radius* and *Cylinder.height*
d. type "Cylinder.Execute" to evaluate the component.

Step 4. After the main program is written, type the words "*subroutine Cylinder()*" and "*end subroutine*" to create the system subroutines for the *Cylinder* system. In the body of the subroutine:
a. type in the equations to calculate the surface area and volume.

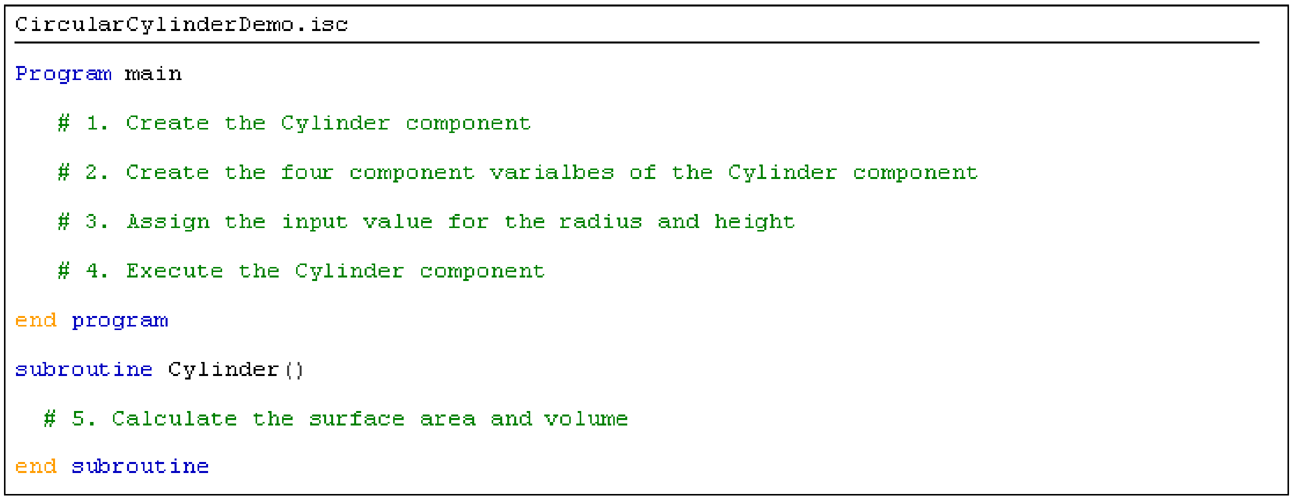
Step 5. Save the file.

Step 6. Select "*Tools > Run Current Script/Project File*" from the menu to run the iSCRIPT code.

Step 7. Find the solution on the screen or in the file "*outputscript.txt*."

The first part of the main program should consist of the component declaration of the *Cylinder* component using a CreateComponent statement, and the declaration of its four component variables using a CreateVariable statement. Then, the cylinder radius and height should be assigned. The *Cylinder* component may be executed to calculate the surface area and volume by using *Cylinder.Execute*. Beside the main program, an executable subroutine, which has the same name "*Cylinder*," should be created for the "Cylinder component" to compute the surface area and volume.

Figure 5.1 shows the program outline for this problem. You may replace the comment statements of steps 1 through 5 with your own iSCRIPT code segment to complete the program. A sample completed program is shown in Figure 5.2.



```
CircularCylinderDemo.isc

Program main

   # 1. Create the Cylinder component

   # 2. Create the four component varialbes of the Cylinder component

   # 3. Assign the input value for the radius and height

   # 4. Execute the Cylinder component

end program

subroutine Cylinder()

  # 5. Calculate the surface area and volume

end subroutine
```

Figure 5.1. Program outline for Problem 5.1.

We will now compare the script in Figure 5.2 with the program outline in Figure 5.1. Line 4 is the statement used to create the component, and Lines 5 through 8 describe the variables of the component. Note that here we have used only three parameters with the *CreateVariable* statment: component name, variable name, and variable type. We chose to not supply other additional, optional properties of the variables, including providing a default value, setting limits or constraints on the variables, or providing an engineering unit. The default value in this case will be automatically set to be zero. In later problems, we will demonstrate how to use those properties of the iSCRIPT component variable.

```
CircularCylinder.isc

 1: Program main
 2:
 3:    # Create the Cylinder component and its variables
 4:    CreateComponent(Cylinder, the_circular_cylinder)
 5:    CreateVariable(Cylinder, radius, double)
 6:    CreateVariable(Cylinder, height, double)
 7:    CreateVariable(Cylinder, surface_area, double)
 8:    CreateVariable(Cylinder, volume, double)
 9:
10:    Cylinder.radius = 5
11:    Cylinder.height = 10
12:
13:    Cylinder.Execute
14:
15: end program
16:
17: subroutine Cylinder()
18:
19:   pi as real
20:
21:   pi = 3.141592654
22:
23:   Cylinder.surface_area = 2*pi*Cylinder.radius^2 + 2*pi*Cylinder.radius*Cylinder.
        height
24:   Cylinder.volume = pi*Cylinder.radius^2*Cylinder.height
25:
26: end subroutine
```

Figure 5.2. iSCRIPT program for Problem 5.1.

Line 13 contains a statement to evaluate the component.

Lines 17 through 26 represent the actual model of the component implemented as a subroutine with the same name as the component. This subroutine includes statements implementing the equations to compute the surface area and volume. Note the way in which component variables are used, compared to the procedure for Problem 4.3 in Section 4.4.

The script file can be found in the subfolder /*SampleScripts/ CircularCylinder/Example5.1* of the iSCRIPT installation folder. If the program is run, the same result as in Problem 4.3 will be obtained.

5.4. Writing a Program in Several Script Files

iSCRIPT allows you to write a program in several files. Actually, this is highly recommended when the program contains several components. This will make the program more portable and easy to manage in a shared project environment. In addition, a component file can easily be re-used or shared by other systems that have the same component. It is also easy to add new components to the current system or modify the available components since the program structure matches the engineering decomposition (physical, conceptual, or disciplinary) of the system.

To link all the script files to a project, the user must write a single project file (*.ipr), which records the name and path information of all the script files. The project file must start with a keyword "*Project*" in the first line and each line can only contain the one script file name, while the path of each script file must be included with the file names (if all of the files are not in the same folder as the project file). Figure 5.3 shows an example of a project file. Note that the subfolders "PS", "OLS", "CHS", "ECS", VCPAOS", "FLS", and "AFS" are subfolders of the project file *ata.ipr*.

```
ata.ipr
--------------------------------------------------------------
project
main.isc
PS\PS.isc
OLS\OLS.isc
CHS\CHS.isc
ECS\ECS.isc
VCPAOS\VCPAOS.isc
FLS\FLS.isc
AFS\AFS.isc
```

Figure 5.3. An example of a project file.

To run a problem with a project file, simply open the project file and select "*Tool > ISCRIPT*" from the menu to run the whole project.

Note:

- In a project, there must be a program in a script file which will be the starting point of the program execution.
- If a script file is in the same folder as the project file, the directory or path information may be neglected (e.g., "*main.isc*" in Figure 5.3).
- The order of the script files is not important.

5.5. An Example of a Program Developed in Several Script Files

Here again, we use the *Circular Cylinder* problem as an example.

Problem 5.2: Solve Problem 4.3 Using Several iSCRIPT Files

The code developed in Problem 4.3 has one main program, one function, and one subroutine. These three program structures can each be written in a separate file. Thus, together with a project file, this project will now be written in 4 files. These file are listed in Figures 5.4 through 5.7.

```
Cyliner.ipr

project
Cylinder.isc
Area.isc
Volume.isc
```

Figure 5.4. Project file Cylinder.ipr for Problem 5.2.

```
Cylinder.isc

program main

  radius, height, surface_area, volume as real
  pi as real

  radius = 5;
  height = 10;

  surface_area = get_area(radius, height)

  call get_volume(volume, radius, height)

end program
```

Figure 5.5. Script file Cylinder.isc for Problem 5.2.

```
Area.isc

function get_area(r,h)

  get_area, r, h as real
  pi as real

  pi = 3.14159265355;

  get_area = 2*pi*r^2 + 2*pi*r*h;

end function
```

Figure 5.6. Script file Area.isc for Problem 5.2.

```
Volume.isc

subroutine get_volume(v,r,h)

  v, r, h as real
  pi as real

  pi = 3.14159265355;

  v = pi*r^2*h

end subroutine
```

Figure 5.7. Script file Volume.isc for Problem 5.2.

The code can be found in the subfolder /*SampleScripts /CircularCylinder/Example5.2* of the iSCRIPT installation folder. Follow the procedures below to run this code:

- Open iSCRIPT Editor.

- Select "*File > Open*" from the menu.
- The File Open dialog box appears.
- Navigate to the subfolder /*SampleScripts/CircularCylinder/Example5.2* of the iSCRIPT installation folder.
- Open the file *Cylinder.ipr*.
- Select "*Tools > Run Current Script/Project File*" from the menu.

If the program is run, same result will be obtained as in Problem 4.3.

5.6. An Example of a System with Several Components

We will use a heat rejection system to illustrate performance analysis of a system with several components.

Problem 5.3: Calculating the Cost of a Heat Rejection System

The system is illustrated in Figure 5.8.

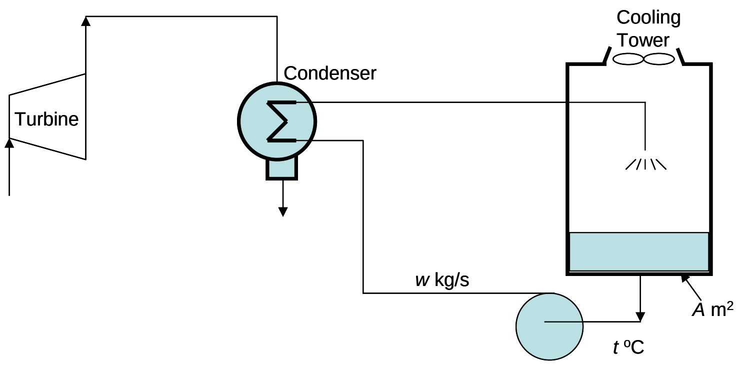


Figure 5.8. Cooling tower, pump, and condenser system of a heat rejection system.

The objective is to evaluate the initial plus operating costs of the system. The heat-rejection rate from the condenser is provided as 14MW. The following costs in dollars are included in the problem description:

- Initial cost of cooling tower, $800A^{0.6}$, where A = area, m^2
- Lifetime pumping cost, $0.0005w^3$, where w = flow rate of water, kg/s
- Lifetime penalty in power production due to elevation of temperature in cooling water, $270t$, where t = temperature of water entering the condenser, ^{o}C.
- The rate of heat transfer from the cooling tower can be represented adequately by the expression $q = 3.7w^{1.2}tA$ (W).

Assume $A=170$ m^2 and $w=200$ kg/s. What is the total cost of the system?

The heat rejection system can be physically decomposed into three components: cooling tower, pump, and condenser, as illustrated in Figure 5.8. Note that in a detailed model, each of the components may be subsystems comprised of other components.

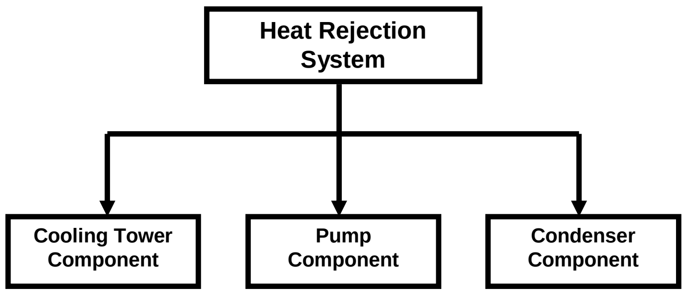


Figure 5.9. Decomposition of the system.

By virtue of the physical decomposition strategy adopted, the total cost C_{sys} can be represented as

$$C_{sys} = C_{tower} + C_{pump} + C_{condenser} \quad . \tag{5.1}$$

For the cooling tower, the cost is

$$C_{tower} = 800A^{0.6} \quad . \tag{5.2}$$

For the pump, the cost is

$$C_{pump} = 0.0005w^{3} \quad . \tag{5.3}$$

For the condenser, the cost is

$$C_{condenser} = 270t \quad . \tag{5.4}$$

The unknown variable *t* is calculated by the energy balance between the condenser and cooling tower:

$$Q_{tower} = Q_{condenser} \tag{5.5}$$

$$Q_{tower} = 3.7w^{1.2}tA \quad . \tag{5.6}$$

This problem assumes that the temperature of the water leaving the cooling tower is equal to the temperature of the water entering the condenser (i.e., there is no temperature change in the pump).

$$t_{tower} = t_{condenser} \tag{5.7}$$

The iSCRIPT code modeling the above system is developed to be consistent with the decomposition procedure adopted for the design. The variables used for the system and each of the subsystems are summarized in Table 5.1 below.

System	Variable	Remarks
Heat Rejection System	C_{sys}	The total cost of the heat rejection system
	w	Flow rate of cooling water (kg/s)
Components	**Variable**	
Tower	C_{tower}	First cost of cooling tower
	A	Area of cooling tower (m^2)
	Q_{tower}	Heat rejection rate of the cooling tower (W)
	t_{out}	Temperature of water leaving the cooling tower ($°C$)
Pump	C_{pump}	Life time pumping cost
Condenser	$C_{condenser}$	Life time penalty in power production due to elevation of temperature of cooling water
	$Q_{condenser}$	Heat absorption rate of the condenser (W)
	t_{in}	Temperature of water entering the condenser ($°C$)

Table 5.1. Components and variables of the heat rejection system.

The development of the iSCRIPT code follows the general procedures in Section 3.2. The outline of the script program is shown in Figure 5.10. The problem has one system, *HR_sys*, and three components: *Tower*, *Pump,* and *Condenser*. Both the system and the components, and their variables, need to be declared. Each component must be simulated in its own component subroutine. The component and component variable declaration are done in the main program.

```
HeatRejectionDemo2.isc

program main

   # 1. Declare the Tower, Pump, Condenser components and their variables

   # 2. Declare the HR_sys system and its variables

   # 3. Declare the objective variable HR_sys.C

   # 4. Assign the input value of Tower.A and HR_sys.w

   # 5. Execute HR_sys system

end program

subroutine HR_sys()

   # 6. Execute the Tower component

   # 7. Execute the Pump component

   # 8. Execute the Condenser component

   # 9. Calculate the total system cost (Eqn. 5.1)

end subroutine

subroutine Tower()

   # 10. Get the heat flux of Tower by energy balance (Eqn. 5.5)

   # 11. Calculate the temperature of water out of Tower (Eqn. 5.6)

   # 12. Calculate the cost of the Tower (Eqn. 5.2)

end subroutine

subroutine Pump()

   # 13. Calculate the cost of Pump (Eqn. 5.3)

end subroutine

subroutine Condenser()

   # 14. The temperature of entering water of Condenser is the temperature of leaving
       water of Tower (Eqn. 5.7)

   # 15. Calculate the cost of Condenser (Eqn. 5.4)

end subroutine
```

Figure 5.10. Program outline for Problem 5.5.

The complete code is written in six files (project file, main program file, HR_sys system file, Tower component file, Pump component file, and Condenser component file), and are shown in Figures 5.11 through 5.16.

```
HeatRejection.ipr

project
main.isc
HR_sys.isc
Tower.isc
Pump.isc
Condenser.isc
```

Figure 5.11. The project file for Problem 5.5.

```
main.isc
# This is the main program for the heat rejection system

program main

#.1 Create the subsystem

   # Create the Tower component and its variables
   CreateComponent(Tower, the cooling tower)
   CreateVariable(Tower, C, double, 0,0, 1.0E9, 0.0, 0.0, $)
   CreateVariable(Tower, A, double, 0,0, 500.0, 10.0, 105.0, m^2)
   CreateVariable(Tower, Q, double, 0,0, 1.0E9, 0.0, 0.0, W)
   CreateVariable(Tower, T_out, double, 0,0, 500, 0.0, 0.0, C)

   # Create the Pump component and its variables
   CreateComponent(Pump, the pump)
   CreateVariable(Pump, C, double, 0,0, 1.0E9, 0.0, 0.0, $)

   # Create the Condenser component and its variables
   CreateComponent(Condenser, the condenser)
   CreateVariable(Condenser, C, double, 0,0, 1.0E9, 0.0, 0.0, $)
   CreateVariable(Condenser, Q, double, 0,0, 1.0E9, 0.0, 0.0, W)
   CreateVariable(Condenser, T_in, double, 0,0, 500, 0.0, 0.0, C)

#.2 Create the systems

   # Create the heat rejection system and its variables
   CreateComponent(HR_sys,Heat_Rejection_System)
   CreateVariable(HR_sys, C, double, 0,0, 1.0E9, 0.0, 0.0, $)
   CreateVariable(HR_sys, w, double, 0,0, 500.0, 10.0, 220.0, kg/s)

#.3 Assign the input values

   Tower.A = 170
   HR_sys.w = 200

#.3 Assign the input values

   HR_Sys.Execute

end program
```

Figure 5.12. The main program for Problem 5.5.

In the main program, the upper bound, lower bound, and default values of the component variables are set during the component variable declaration by using CreateComponent statement. These values are summarized in Table 5.2 below.

System and Component	Variable	Upper Bound	Lower Bound	Default Value	Remarks
Heat Rejection System	C_{sys}	1.0E9	0	0	1. The upper bound and lower bound of a design variable is selected to fit the design constraints. 2. The default value can be assigned during the variable declaration to avoid extra input value assignments in the body of the program 3. The upper and lower bounds of unconstraint variables can be set to be large or smaller so that the variables can never reach the constraints.
	w	500.0	10.0	220.0	
Tower	C_{tower}	1.0E9	0	0	
	A	500.0	10.0	105.0	
	Q_{tower}	1.0E9	0	0	
	t_{out}	500	0	0	
Pump	C_{pump}	1.0E9	0	0	
Condenser	$C_{condenser}$	1.0E9	0	0	
	$Q_{condenser}$	10E9	0	0	
	t_{in}	500	0	0	

Table 5.2. The upper bound, lower bound, and default values provided for the component variables in Problem 5.5.

```
HR_sys.isc

# This is the main controlling routine for the Heat Rejection System

subroutine HR_sys()

   Condenser.Q = 0.14E+08

   Tower.Execute
   Pump.Execute
   Condenser.Execute

   HR_sys.C = Tower.C + Pump.C + Condenser.C

end subroutine
```

Figure 5.13. The subsystem program for Problem 5.5.

```
Tower.isc

# This is the routine for the cooling tower component

subroutine Tower()

   Tower.Q = Condenser.Q

   Tower.T_out = Tower.Q / 3.7 / Tower.A / (HR_sys.w**1.2)

   Tower.C = 800.0 * (Tower.A**0.6)

end subroutine
```

Figure 5.14. The tower component evaluation program for Problem 5.5.

```
Pump.isc

# This is the routine for the pump component

subroutine Pump()

   Pump.C = 0.0005 * HR_sys.w**3.0

end subroutine
```

Figure 5.15. The pump component evaluation program for Problem 5.5.

```
Condenser.isc

# This is the routine for the Condenser component

subroutine Condenser()

   Condenser.T_in = Tower.T_out

   Condenser.C = 270 * Condenser.T_in

end subroutine
```

Figure 5.16. The condenser component evaluation program for Problem 5.5.

The code can be found in the subfolder */SampleScripts/HeatRejection/Example5.3* of the iSCRIPT installation folder. Follow the steps below to run this code:

1. Open iSCRIPT Editor.
2. Select “*File > Open*” from the menu.

⇒ The File Open dialog box appears.

3. Navigate to the subfolder /*SampleScripts/HeatRejection/Example5.3* of the iSCRIPT installation folder.
4. Open the file *HeatRejection.ipr*.
5. Select “*Tools > Run Current Script/Project File*” from the menu.

The result obtained is $31,846.19 for the total cost of the system.

6. Optimization in iSCRIPT

In this chapter, we introduce the optimization procedure in iSCRIPT. Two examples will be given to illustrate system optimization.

6.1. Optimization Based on Component Modeling

iSCRIPT's built-in optimization procedure is based on the decomposition of systems into components. iSCRIPT can optimize a single component or a large system containing several subsystems or components by using the integrated local global optimization (ILGO) technique. In its most basic form, every iSCRIPT optimization job must consist of one system and at least one subsystem consisting of at least one component. (By inference, the system-subsystem-component hierarchy for the most basic job is then represented by one component which is also the subsystem, which is the system.)

To optimize a component or subsystem, we must indicate the variable that should be maximized or minimized. This variable is termed the "objective variable" and is indicated by using the AddObjective statement. The details of this statement are as follows:

AddObjective (*component*, *variable* [,maxmin])

Note: Segments enclosed in square brackets are optional and may be omitted.

component --	The component to be optimized (a string limited to 24 characters). This component must be a component previously declared with the CreateComponent command.
variable --	Name of the objective variable (a string limited to 24 characters). This variable must be a variable previously declared for this component using the CreateVariable command.
maxmin --	0 or 1. Indicates whether this is a minimization or maximization objective. Use 0 to minimize and 1 to obtain a maximum. This argument is optional. The default value is 0.

After the objective variable has been indicated, we also need to indicate which variables are free for optimization. These are the variables whose values can be varied in order to obtain the optimum value of the objective function. The variables are referred to as optimization or decision variables and are indicated by the AddVarObjective statement. The details of this statement are as follows:

```
AddVarObjective (component, variable [,delta])
```

Note: Segments enclosed in square brackets are optional and may be omitted.

component --	The component to which the variable belongs (a string limited to 24 characters). This component must be the component that will be optimized, or a subcomponent of it (see Section B.2.6). This component must be a component previously declared with the CreateComponent command.
variable --	Name of the variable (a string limited to 24 characters). This variable must be a variable previously declared for this *component* using the CreateVariable command.
delta --	This parameter further narrows the optimization search space for the optimization variable. If the optimization variable *x* were defined by CreateVariable to have the lower and upper bounds *L* and *M*, respectively, then $x = [L, M]$, and the size of the search space is $M - L$. The parameter, Δ, will narrow this space to from [*L*, *M*] to $[MAX(L, x_i - \Delta), MIN(M, x_i + \Delta)]$, where x_i is the default value of *x*, a computed initial value for *x*, or the value of *x* after a prior optimization step. This parameter may be used to further reduce the search space after a prior optimization step (or after an initial computation designed to obtain a good initial estimate narrowing the range of the optimum value of *x*) to speed up the optimization process.

Note:

- The objective and optimization variable are previously-declared component variables.
- There can be more than one optimization variable. Note that the fewer optimization variables there are, the faster the optimization will complete. Also, the smaller the search space of the optimization variable, the faster the optimization will complete.

After both the objective variable and the optimization variables are declared by *AddObjective* and *AddVarObjective* statements, the optimization calculation is executed by using the statement

```
Component.Optimize
```

6.2. Optimization of a System with a Single Component

Let us continue to use the *Circular Cylinder* problem for the example. The model used for the components of the system are very rudimentary, but serve to illustrate how iSCRIPT can be used to model engineering systems. The problem is revised as follows:

Problem 6.1: Minimize the Surface Area of a Circular Cylinder with a Fixed Value of Volume of 800 m^3.

The procedures required to solve this problem follow the general procedure presented in Section 3.3. The steps are as follows:

Step 1. Open iSCRIPT Editor.
Step 2. Create a new file and save it as an "*.isc" file.
Step 3. Type the words "program *main*" and "end program" to create a main program. In the body of the main program:
  a. declare the *Cylinder* system by using CreateComponent statement
  b. declare the four component variables (radius, height, surface_area, volume) of the *Cylinder* system using CreateVariable statement
  c. assign the input value for *Cylinder.volume*.
  e. declare the objective variable *Cylinder.surface_area* by using AddObjective statement
  f. declare the optimization variable *Cylinder.radius* by using AddVarObjective statement
  g. type "Cylinder.Optimize" to optimize the overall system.

Step 4. After the main program is written, type the words "subroutine *Cylinder()*" and "end subroutine" to create the system subroutines for the *Cylinder* component. In the body of the subroutine:
   a. type the equations for the height and surface_area.

Step 5. Save the file.

Step 6. Select "*Tools > Run Current Script/Project File*" from the menu to run the iSCRIPT code.

Step 7. View the output on screen or in the file "*outputscript.txt*."

Note that the only input variable for this problem is the *volume*. The task is to determine values of the other variables (*r* and *h*) that minimize the surface area of the model (cylinder). These (decision) variables do not require input values. The objective variable is the *surface_area* and the optimization variable is the *radius*. The *height* is a dependent variable computed from the *volume* and *radius*.

In the main program, the *Cylinder* component and its four component variables are declared first. The input values are then assigned, followed by the declaration of the objective and optimization variables. The last part of the main program is the optimization calculation.

In the Cylinder component model, the height of the cylinder is computed from the *volume* and the *radius*. Then the value of the objective variable *surface_area* is calculated. Figure 6.1 shows the program outline for this problem.

```
CircularCylinderOptimizeDemo.isc

Program main

   # 1. Create the Cylinder component

   # 2. Create the four component varialbes of the Cylinder component

   # 3. Assign the input value for the volume

   # 4. Declare the objective variable surface_area

   # 5. Declare the optimization decision variable radius

   # 6. Start optimization calculation

end program

subroutine Cylinder()

  # 7. Calculate the height from volume and radius

  # 8. Calculate the surface_area

end subroutine
```

Figure 6.1. Program outline for Problem 6.1.

The complete code is shown in Figure 6.2. If you compare it with the code in Figure 6.1, the meaning of each statement is obvious.

Note:

- The genetic algorithm in iSCRIPT will be used for the optimization procedure. Details of the genetic algorithm are provided in Appendix B.2.10.
- Since the optimization includes a search process with many iteration steps, both the objective variable and optimization variable must be set with a reasonable upper bound and lower bound (constrained). This is done in the component variable declaration statement.
- The speed and accuracy of the optimization will be increased if the range of the optimization variable is small and its default value is close to optimum value. (Care must be taken to ensure that the optimum value lies inside the constraint or domain of the optimization variable. Simply making the domain large enough will take care of this requirement. For instance, specifying a radius that lies within the range of 0 to 100 is a safe range within which the optimum radius will lie.)

```
CircularCylinder.isc
```

```
program main

   # Create the Cylinder component and its variables
   CreateComponent(Cylinder, the_circular_cylinder)
   CreateVariable(Cylinder, radius, double, 0,0, 100, 0.01, 5.0)
   CreateVariable(Cylinder, height, double, 0,0, 20000000, 0.0, 10.0)
   CreateVariable(Cylinder, surface_area, double, 0,0, 10000, 0.0, 0.0)
   CreateVariable(Cylinder, volume, double, 0,0, 10000, 0.0, 0.0)

   # Assign the input value
   Cylinder.volume = 800

   # Declare the objective variable
   AddObjective(Cylinder, surface_area, 0)

   # Declare the optimization decision variable
   AddVarObjective(Cylinder, radius)

   # Start optimization calculation
   Cylinder.Optimize

end program

subroutine Cylinder()

  pi as real

  pi = 3.141592654

  Cylinder.height = Cylinder.volume/pi/(Cylinder.radius^2)

  Cylinder.surface_area = 2*pi*Cylinder.radius^2 + 2*pi*Cylinder.radius*Cylinder.height

end subroutine
```

Figure 6.2. The complete code for Problem 6.1.

The code can be found in the subfolder /*SampleScripts/CircularCylinderExample6.1* of the iSCRIPT installation folder. Follow these steps to run this code:

1. Open iSCRIPT Editor
2. Select "*File > Open*" from the menu.
   $\Rightarrow$ The File Open dialog box appears.
3. Navigate to the subfolder /*SampleScripts/CircularCylinder/Example6.1* of the iSCRIPT installation folder.
4. Open the file *CircularCylinder.isc*.
5. Select "*Tools > Run Current Script/Project File*" from the menu.

The code returns the values of the minimized surface area and the corresponding cylinder radius as 477.06172 and 5.030, respectively. Note that due to the fact that a genetic algorithm is used, the results may change slightly with each execution. Also, note that default values of the optimization parameters in iSCRIPT have been used (which is why no parameters are specified). These defaults are usually acceptable for most problems. Details of the parameters are given in Appendix B.2.11.

The analytic results can be derived by differentiating the following equation and setting the value to zero to derive the optimum.

$$S = 2\pi(r+h) = 2\pi r\left(r + \frac{V}{\pi r^2}\right) \tag{6.1}$$

For this problem, the analytical results are $r = \sqrt[3]{\frac{V}{2\pi}} = 5.0308$ and $S = 3V^{2/3}(2\pi)^{1/3} = 477.0617$, respectively.

Although the analytic results are easily obtained, note the speed of the genetic (GA) optimization procedure used by iSCRIPT in calculating the results and the accuracy of the results. Conventional wisdom would claim that GA procedures are robust but take significantly longer to converge or compute the optimum of continuous functions compared with gradient-based methods. **However, the unique GA procedure in iSCRIPT has been tuned to perform exceptionally well, even for continuous functions, while retaining the robustness expected of a GA procedure.**

6.3. Optimization of a System with Multiple Components

Let us now consider a problem that contains more than one component. The heat-rejection system design problem in Problem 5.3 is used.

Problem 6.2: Minimize the Cost of the Heat Rejection System in Problem 5.3

The goal of the problem is to find the optimum value of *A* and *w* that will minimize the total cost *C*.

The objective variable of the problem is the total cost C_{sys} of the Heat Rejection System. There are two optimization variables: flow rate of water, *w*, of the Heat Rejection System and the area, *A*, of the Cooling Tower component. The objective function of this problem may be expressed as

$$C_{sys} = f(w, A). \tag{6.2}$$

The goal is to seek the optimum values of *w* and *A* that will minimize C_{sys}.

Note that in this problem, we have two optimization variables and they belong to different components. Therefore, we need to indicate the relationship between the Heat Rejection System and the Cooling Tower component. This is done by indicating which components belong to each subsystem. In a multi-subsystem environment, several subsystems may be present in the entire system, and the subsystem composition must be indicated. Otherwise, it would be impossible to determine which component belongs to which subsystem. In the current example, there is only one subsystem, thus this is also the system. The subsystem composition is indicated using the AddSubComponent command.

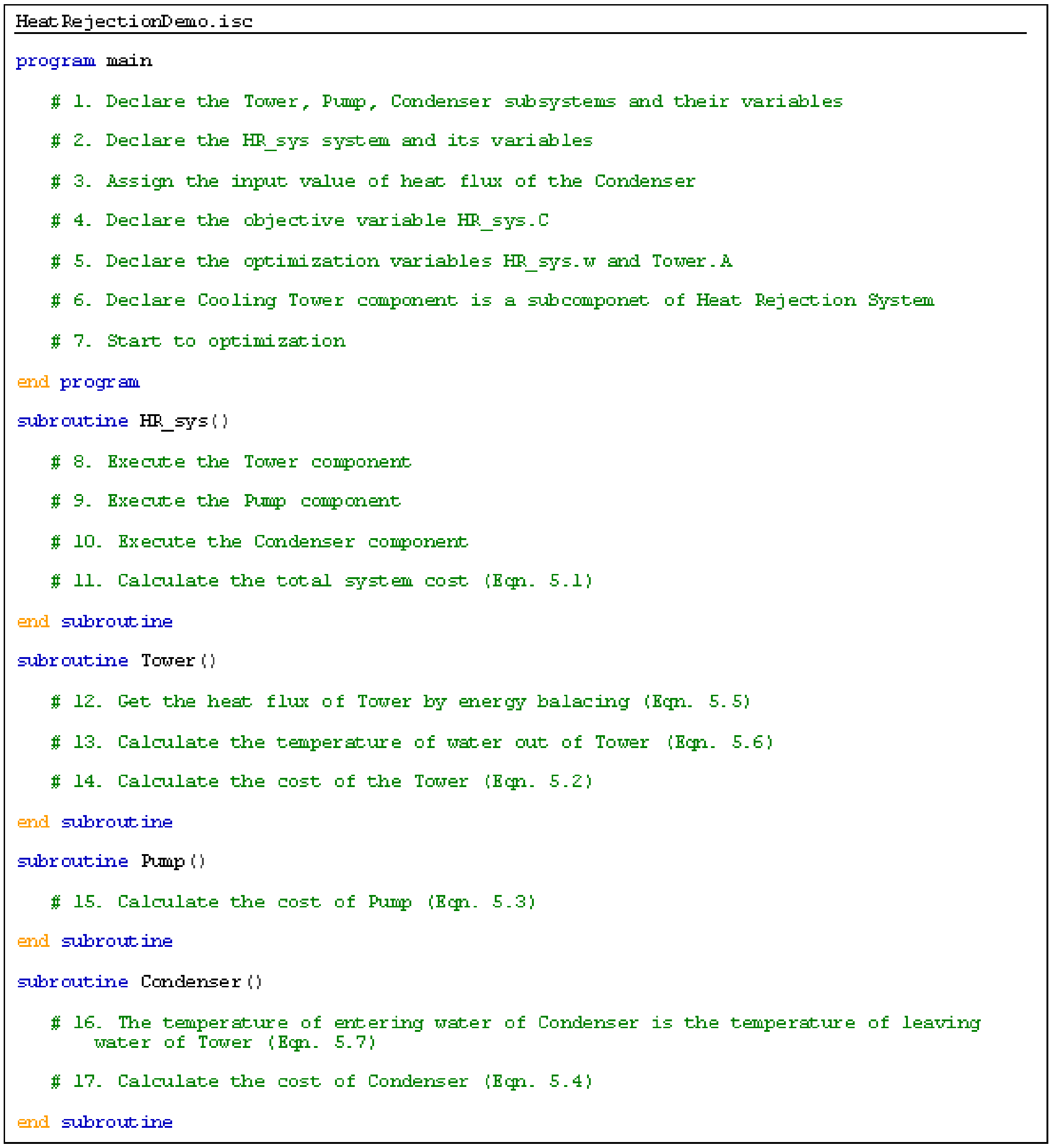
The component-subsystem-system relationship in iSCRIPT is summarized below:

- **Every iSCRIPT project is assumed to contain one system (a system is automatically created per project)**
- **All subsystems in a project automatically belong to the system. If there is only one subsystem, then this subsystem comprises the system.**
- **The component-subsystem relationship or hierarchy is formalized using the AddSubComponent command.**

The format for the AddSubComponent command is shown below:

AddSubsystem (*component, subcomponent*)

component --	The component consisting of other components or subsystem (a string limited to 24 characters). This component must be a component previously declared with the CreateComponent command.
subcomponent --	A component to be identified as a subcomponent of a *component*. This component must be a component previously declared with the CreateComponent command.



```
HeatRejectionDemo.isc
program main
   # 1. Declare the Tower, Pump, Condenser subsystems and their variables
   # 2. Declare the HR_sys system and its variables
   # 3. Assign the input value of heat flux of the Condenser
   # 4. Declare the objective variable HR_sys.C
   # 5. Declare the optimization variables HR_sys.w and Tower.A
   # 6. Declare Cooling Tower component is a subcomponet of Heat Rejection System
   # 7. Start to optimization
end program
subroutine HR_sys()
   # 8. Execute the Tower component
   # 9. Execute the Pump component
   # 10. Execute the Condenser component
   # 11. Calculate the total system cost (Eqn. 5.1)
end subroutine
subroutine Tower()
   # 12. Get the heat flux of Tower by energy balacing (Eqn. 5.5)
   # 13. Calculate the temperature of water out of Tower (Eqn. 5.6)
   # 14. Calculate the cost of the Tower (Eqn. 5.2)
end subroutine
subroutine Pump()
   # 15. Calculate the cost of Pump (Eqn. 5.3)
end subroutine
subroutine Condenser()
   # 16. The temperature of entering water of Condenser is the temperature of leaving
       water of Tower (Eqn. 5.7)
   # 17. Calculate the cost of Condenser (Eqn. 5.4)
end subroutine
```

Figure 6.3. Program outline for Problem 6.2.

The program outline is shown in Figure 6.3 above. Compared with Problem 5.3, we only need to make the following changes in the main program:

- Indicate the objective variables using AddObjective.
- Indicate the optimization variables using AddVarObjective.
- Start the optimization calculation using *HR_sys.*Optimize.

```
main.isc

# This is the main program for the heat rejection system optimization.

program main

#.1 Create the subsystem

   # Create the Tower subsystem and its variables
   CreateComponent(Tower, the cooling tower)
   CreateVariable(Tower, C, double, 0,0, 1.0E9, 0.0, 0.0, $)
   CreateVariable(Tower, A, double, 0,0, 500.0, 10.0, 105.0, m^2)
   CreateVariable(Tower, Q, double, 0,0, 1.0E9, 0.0, 0.0, W)
   CreateVariable(Tower, T_out, double, 0,0, 500, 0.0, 0.0, C)

   # Create the Pump subsystem and its variables
   CreateComponent(Pump, the pump)
   CreateVariable(Pump, C, double, 0,0, 1.0E9, 0.0, 0.0, $)

   # Create the Conderser subsystem and its variables
   CreateComponent(Condenser, the condenser)
   CreateVariable(Condenser, C, double, 0,0, 1.0E9, 0.0, 0.0, $)
   CreateVariable(Condenser, Q, double, 0,0, 1.0E9, 0.0, 0.0, W)
   CreateVariable(Condenser, T_in, double, 0,0, 500, 0.0, 0.0, C)

#.2 Create the systems

   # Create the heat rejection system and its variables
   CreateComponent(HR_sys,Heat_Rejection_System)
   CreateVariable(HR_sys, C, double, 0,0, 1.0E9, 0.0, 0.0, $)
   CreateVariable(HR_sys, w, double, 0,0, 500.0, 10.0, 220.0, kg/s)

#.3 optimization the system

   AddObjective(HR_Sys, C, 0)

   AddVarObjective(HR_Sys, w)
   AddVarObjective(Tower, A)

   AddSubComponent(HR_Sys,Tower)

   # Optimize the System

   Global.maxPopulation = 2000
   Global.maxInitialEvaluations = 5000
   Global.maxGenerations = 100
   Global.optconvergencelimit =   1.0E-7
   Global.mutationfreq =   0.1
   Global.sampsizepervariable = 500
   Global.maxILGOsteps = 30

   HR_Sys.Optimize

end program
```

Figure 6.4. The main program for Problem 6.2.

Figure 6.4 is the complete code for the main program. Since all of the other files are the same as those in Problem 5.3 (component models are portable from solution to solution), only the main program file is provided here.

Note the keywords *Global.maxPopulation, Global.maxInitialEvaluations, Global.maxGenerations, Global optconvergencelimit, Global,mutationfreq, sampsizepervariable,* and *Global.maxILGOsteps* are the parameters to control the performance (speed and accuracy) of the GA optimization calculation. The default values are usually sufficient to solve most problems. The default values are provided in the iSCRIPT Optimization Reference in Appendix B.2.11. However, in the above program, we have modified these defaults to obtain results faster. Detailed information of these parameters can also be found in Appendix B.2.11.

The scripts solving this problem can be found in the subfolder /*SampleScripts/HeatRejection/Example6.2* of the iSCRIPT installation folder. Follow these steps to run the code:

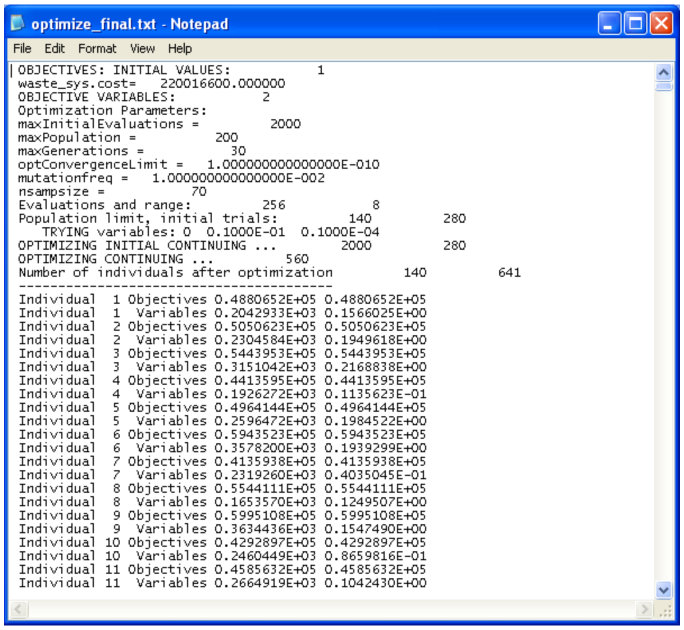
1. Open iSCRIPT Editor.
2. Select "*File > Open*" from the menu.
   ⇒ The File Open dialog box appears.
3. Navigate to the subfolder /*SampleScripts/HeatRejection/Example6.2* of the iSCRIPT installation folder.
4. Open the file *HeatRejection.ipr*.
5. Select "*Tools > Run Current Script/Project File*" from the menu.

The optimization calculation requires a few seconds to run. The final result is contained in the file *outputscript.txt*. The results are compared with those from Stoecker [5] in Table 6.1 below.

	A	w	C_{sys}
Stoecker	202.6	167.9	53812.6
iSCRIPT	202.52	167.95	53829.6

Table 6.1. Comparison of iSCRIPT optimization results for Problem 6.2 with those from Stoecker [5].

Details of the optimization process are contained in the file *optimize.txt*. The details include the initial values of the optimization variables and the objective function, the various realizations of the system, the array of viable systems or realizations (population of individuals in genetic algorithm parlance) by generation or as the optimization progresses, and the final results. A sample *optimize.txt* file is illustrated in Figure 6.5 below.



```
OBJECTIVES: INITIAL VALUES:           1
waste_sys.cost=   220016600.000000
OBJECTIVE VARIABLES:           2
Optimization Parameters:
maxInitialEvaluations =          2000
maxPopulation =          200
maxGenerations =           30
optConvergenceLimit =   1.000000000000000E-010
mutationfreq =   1.000000000000000E-002
nsampsize =           70
Evaluations and range:         256           8
Population limit, initial trials:         140         280
   TRYING variables: 0  0.1000E-01  0.1000E-04
OPTIMIZING INITIAL CONTINUING ...        2000         280
OPTIMIZING CONTINUING ...         560
Number of individuals after optimization         140         641
---------------------------------------------
Individual  1 Objectives 0.4880652E+05 0.4880652E+05
Individual  1  Variables 0.2042933E+03 0.1566025E+00
Individual  2 Objectives 0.5050623E+05 0.5050623E+05
Individual  2  Variables 0.2304584E+03 0.1949618E+00
Individual  3 Objectives 0.5443953E+05 0.5443953E+05
Individual  3  Variables 0.3151042E+03 0.2168838E+00
Individual  4 Objectives 0.4413595E+05 0.4413595E+05
Individual  4  Variables 0.1926272E+03 0.1135623E-01
Individual  5 Objectives 0.4964144E+05 0.4964144E+05
Individual  5  Variables 0.2596472E+03 0.1984522E+00
Individual  6 Objectives 0.5943523E+05 0.5943523E+05
Individual  6  Variables 0.3578200E+03 0.1939299E+00
Individual  7 Objectives 0.4135938E+05 0.4135938E+05
Individual  7  Variables 0.2319260E+03 0.4035045E-01
Individual  8 Objectives 0.5544111E+05 0.5544111E+05
Individual  8  Variables 0.1653570E+03 0.1249507E+00
Individual  9 Objectives 0.5995108E+05 0.5995108E+05
Individual  9  Variables 0.3634436E+03 0.1547490E+00
Individual 10 Objectives 0.4292897E+05 0.4292897E+05
Individual 10  Variables 0.2460449E+03 0.8659816E-01
Individual 11 Objectives 0.4585632E+05 0.4585632E+05
Individual 11  Variables 0.2664919E+03 0.1042430E+00
```

Figure 6.5. Sample *optimize.txt* output file.

The sample problem described in this section is a simple one. In this problem, the overall objective function can be expressed as a single equation,

$$C_{sys} = f(w, A) = 800A^{0.6} + 0.005w^{3} + 270\left(\frac{14\times10^{6}}{3.7\left(w^{1.2}\right)A}\right) . \qquad (6.3)$$

Therefore, the advantages of decomposition may not be apparent. On the other hand, a realistic industrial design/optimization problem could contain many components and a large number of variables. A system may contain several subsystems and the coupling between these subsystems may be complicated. iSCRIPT's decomposition-based modeling approach and ILGO optimization algorithm provide a way to handle these kinds of complicated system design and optimization problems.

7. Running iSCRIPT in Parallel

iSCRIPT can execute programs automatically in a parallel environment (without the user actually parallelizing their codes). However, currently, only the optimization portion of a program actually executes in parallel. Optimization commands are computationally intensive for fairly sized system models. As a result, executing them in parallel provides significant reduction in the time to obtain solutions. For instance, a program named ADVISOR, implemented in MATLAB, which simulates in-detail the model for an automobile drive train system, is reported to take about 25 seconds to run on a fairly sized PC. Optimizing such a program (which is doubtlessly a mixed integer non-linear, or MINL, problem) may require up to 40,000 evaluations of the model. This would result in a total time of about 11 days to obtain results. In a parallel environment, using 12 processors, results may be obtained in as quickly as 24 hours.

Any iSCRIPT program that can run on a single-processor computer can execute in parallel in a multi-processor environment. However, currently, only the optimization portion of the program gains from the parallel environment. This means that programs without any optimization calls will simply execute multiply on all the processors.

The system requirement for running iSCRIPT in parallel is described in the next section while procedures for running iSCRIPT programs in parallel is described in Section 6.2.

7.1. System Requirements

The iSCRIPT executable imparted with automatic parallel features and algorithm is named iscript_mp.exe. This program must be run on a computer (or network of computers) on which MPI has been set up. The complete requirements are as follows:

- iSCRIPT parallel executable (iscript_mp.exe, which is available from an iSCRIPT installation).

- Computer or network of computers on which MPI has been set up. Ideally, this should be a multi-processor environment, but MPI is also able to work on a single-processor installation, spawning virtual processes simulating a multi-processor environment.

- Each computer (or the single computer simulating a parallel environment) should be at least a Pentium PC running at a speed of at least 1GHz, with at least 128MB ram and 200MB free disk space.

7.2. Running an iSCRIPT Program in Parallel

The syntax for running an iSCRIPT program in parallel is as follows:

```
mpirun –np # iscript_mp.exe
```

or

```
mpirun –np # iscript_mp.exe program.isc
```

or

```
mpirun –np # iscript_mp.exe program.ipr
```

In the above syntax,

- # is the required number of processors.
- *program.isc* is any iSCRIPT program file.
- *program.ipr* is any iSCRIPT project file.
- If the first syntax is used, the iSCRIPT parallel program will initially (interactively) query the user for the script or project that the user wishes to execute in parallel.

7.3. A Sample Optimization Problem Run in Parallel

Any iSCRIPT program that can run on a single processor computer can execute in parallel in a multi-processor environment. This example illustrates the use of iSCRIPT in optimizing a problem in parallel.

This sample problem file contains a main program and a subroutine that evaluates a model. The model is the Rastrigin equation, shown in Equation 6.1 below.

$$f(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2) \tag{6.1}$$

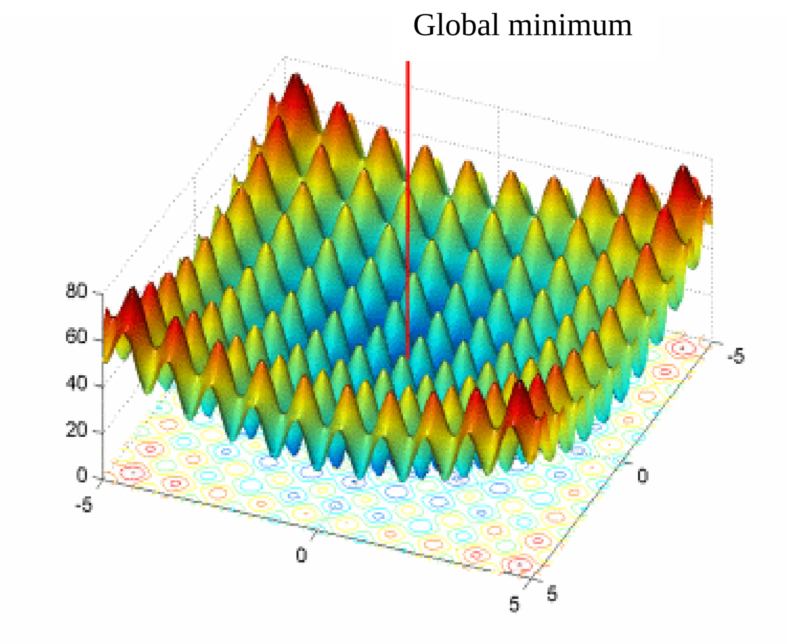


Figure 6.1. Plot of the Rastrigin function.

This function has several local minima, making it difficult for a gradient-based procedure to capture the actual minimum without the benefit of a good starting or guess value. The actual minimum value is 0 and occurs at the values of (x_1, x_2) = (0,0).

Figure 6.2 below shows the iSCRIPT program for finding the optimum of the Rastrigin function.

```
rastrigin.isc

# This is the system program.
# This program defines all components, subsystems, and systems

program main

  #global pi as double
  #pi = 4*atan(1.0)

#.1 Create the components

#.2 Create the components (This system consists of only one component)

   CreateComponent(Rastrigin, Models_entire_system)
   CreateVariable(Rastrigin, y, double, 0,0, 1.0E14, 0.0, 0.0)
   CreateVariable(Rastrigin, x1, double, 0,0, 10.0, -10.0, 0.5)
   CreateVariable(Rastrigin, x2, double, 0,0, 10.0, -10.0, 0.5)

   AddObjective(Rastrigin, y, 0)

   AddVarObjective(Rastrigin, x1)
   AddVarObjective(Rastrigin, x2)

#.3 Optimize the system
   #Rastrigin.Execute

   Global.maxPopulation = 70
   Global.maxInitialEvaluations = 200
   Global.maxGenerations = 10
   Global.optconvergencelimit =   1.0E-7
   Global.mutationfreq =   0.2
   Global.sampsizepervariable = 90


   Rastrigin.Optimize

end program


subroutine Rastrigin ()
#This subroutine is the model for the component - Rastrigin
  x1, x2, pi As Double
  x1 = Rastrigin.x1
  x2 = Rastrigin.x2
  pi = 4*atan(1.0)
  #pi = 3.1415926536
  Rastrigin.y = 20.0 + x1*x1 + x2*x2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
end subroutine
```

Figure 6.2. iSCRIPT program for finding the optimum of the Rastrigin function.

The program is created in a single file (rastrigin.isc) with the following parts:

Main Program

The main program starts on Line 3 with the statement "program main" and ends on line 31.

The first part of the main program (Lines 9 through 12) indicates a component with three variables, *x1*, *x2*, and *y*. This problem is considered to be a system consisted of one subsystem with a single component.

This part of the program also identifies *y* as the objective function and (*x1*, *x2*) as variables to be optimized in achieving the minimum value of the objective function (Lines 14 through 17).

The second part of the program sets the optimization parameters (Lines 22-27) and optimizes the component (Line 29).

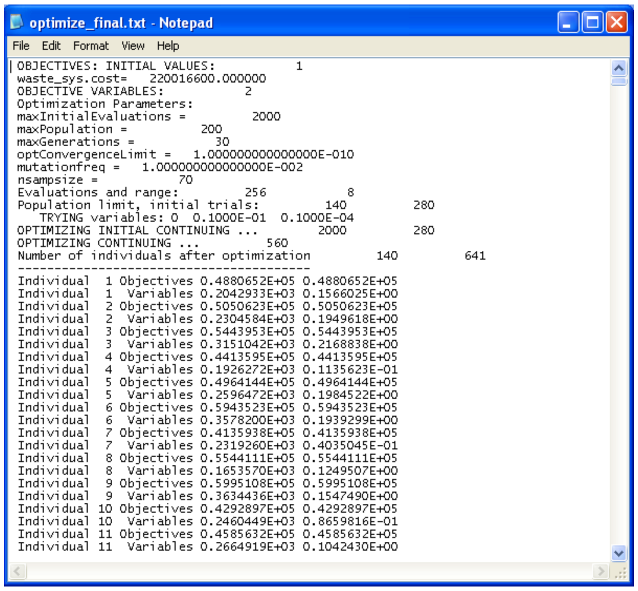
<u>Component Model</u>
The function (or component) is modeled in subroutine Rastrigin (Lines 34 through 41).

Variables are declared and initialized in Lines 33 through 39 of the program.

Equation 6.1 is implemented in Line 41 of the program.

<u>Output</u>
Again, the output file is *outputscript.txt*. The correct results were obtained in about 2.9 seconds on a Pentium workstation using only a population of 70 realizations.

Details of the optimization process are recorded in the file *optimize.txt*. These details include the initial values of the objective variable and the optimization variables, the various realizations of the system being evaluated, the array of viable systems or realizations (population of individuals in genetic algorithm parlance) by generation or as the optimization progresses, and the final results. A sample *optimize.txt* file is illustrated in Figure 6.3 below.



```
OBJECTIVES: INITIAL VALUES:           1
waste_sys.cost=   220016600.000000
OBJECTIVE VARIABLES:           2
Optimization Parameters:
maxInitialEvaluations =         2000
maxPopulation =          200
maxGenerations =           30
optConvergenceLimit =   1.000000000000000E-010
mutationfreq =   1.000000000000000E-002
nsampsize =          70
Evaluations and range:         256           8
Population limit, initial trials:         140         280
   TRYING variables: 0  0.1000E-01  0.1000E-04
OPTIMIZING INITIAL CONTINUING ...        2000         280
OPTIMIZING CONTINUING ...         560
Number of individuals after optimization        140         641
----------------------------------------
Individual  1 Objectives 0.4880652E+05 0.4880652E+05
Individual  1  Variables 0.2042933E+03 0.1566025E+00
Individual  2 Objectives 0.5050623E+05 0.5050623E+05
Individual  2  Variables 0.2304584E+03 0.1949618E+00
Individual  3 Objectives 0.5443953E+05 0.5443953E+05
Individual  3  Variables 0.3151042E+03 0.2168838E+00
Individual  4 Objectives 0.4413595E+05 0.4413595E+05
Individual  4  Variables 0.1926272E+03 0.1135623E-01
Individual  5 Objectives 0.4964144E+05 0.4964144E+05
Individual  5  Variables 0.2596472E+03 0.1984522E+00
Individual  6 Objectives 0.5943523E+05 0.5943523E+05
Individual  6  Variables 0.3578200E+03 0.1939299E+00
Individual  7 Objectives 0.4135938E+05 0.4135938E+05
Individual  7  Variables 0.2319260E+03 0.4035045E-01
Individual  8 Objectives 0.5544111E+05 0.5544111E+05
Individual  8  Variables 0.1653570E+03 0.1249507E+00
Individual  9 Objectives 0.5995108E+05 0.5995108E+05
Individual  9  Variables 0.3634436E+03 0.1547490E+00
Individual 10 Objectives 0.4292897E+05 0.4292897E+05
Individual 10  Variables 0.2460449E+03 0.8659816E-01
Individual 11 Objectives 0.4585632E+05 0.4585632E+05
Individual 11  Variables 0.2664919E+03 0.1042430E+00
```

Figure 6.3. Sample *optimize.txt* output file.

8. Interface with Other Software

8.1. Purpose

iSCRIPT provides several methods for interfacing with other third-party software. This is extremely important in providing functionality as a system-of-systems tool for engineers. The methods in iSCRIPT for interfacing with other software include:

1. Open a process and execute the third-party executable directly. This method is universal and works for any third-party software, as long as the software can be opened from a shell or the command line.

2. Provide direct interface to specific software include Microsoft Excel, TTC Technologies' INSTED software programs and Database, and TTC Technologies' AEROFLO multi-disciplinary CFD program.

3. Providing the syntax support necessary to run scripts developed in other environments (such as MATLAB) directly in iSCRIPT with minimum modification.

The procedures for executing a third-party executable are described in the next section.

8.2. Running a Third-Party Software or Executable

An external or third-party executable may be activated in iSCRIPT as if from a command line using the execute command. The syntax is shown below.

```
call execute (‘executable_filename’, [‘commandline_argument’])

or

iresult =  execute (‘executable_filename’, [‘commandline_argument’])
```

The above rules govern the process of calling the open command.

- *executable_filename* is a string representing the executable file. The string may include the path if the file is not in the iSCRIPT working directory at the time of the call (see Section 7.3). The filename must be enclosed in single quotes.
- *commandline_argument* is an optional string containing arguments that should be passed to the executable at runtime.
- The execute command may be used to run any executable at runtime, including Microsoft Excel, other CFD programs, and MATLAB (using MATLAB's component compile (MCC) tool).

- Combining the ability to run an external executable with the input/output procedures in Section 4.8 allows iSCRIPT to write input files for an external program, run the program, and read the output. This allows engineering models implemented in other environments to easily be integrated with an iSCRIPT solution. See sample problem 17 for an example solution integrating an aircraft engine model program written in MATLAB.

8.3. Setting/Changing the Working Directory

iSCRIPT provides commands for changing the execution directory in case a third-party executable does not reside in the same folder as an iSCRIPT program that wishes to run the executable. The syntax to accomplish this is:

```
call changedirectory ([‘path’])

or

ivar =  changedirectory ([‘path’])
```

The above rules govern the process of calling the changedirectory command.

- *path* is a string representing the directory to change to. When not provided, the command reverts to the executing iSCRIPT directory (See sample problem 18). Path must be enclosed in single quotes.
- The output of the changedirectory command is 1 if successful and 0 if an error occurred.

8.4. Determining the Working Directory

To determine the current directory (at iSCRIPT runtime), iSCRIPT has provided the keyword currentdirectory. When used with the write statements (Section 4.8), the current working directory is printed. See sample problem 18 which is described in the appendix.

8.5. Example of Running a MATLAB script

A short MATLAB script that uses some of the matrix manipulation commands in MATLAB is used to the extent to which iSCRIPT can execute MATLAB programs.

<u>Model</u>

$$\mathbf{x}=\begin{bmatrix}4&2&3\\4&6&6\\7&8&9\end{bmatrix},\mathbf{y}=\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix},\mathbf{b}=\begin{bmatrix}4&1&0\\2&1&2\\5&1&1\end{bmatrix}$$

$$\mathbf{z}=\mathbf{y}\times\mathbf{x}+\mathbf{b}\times\mathbf{x}+2\mathbf{y}/\mathbf{x}$$

8.5.1 Running the MATLAB script

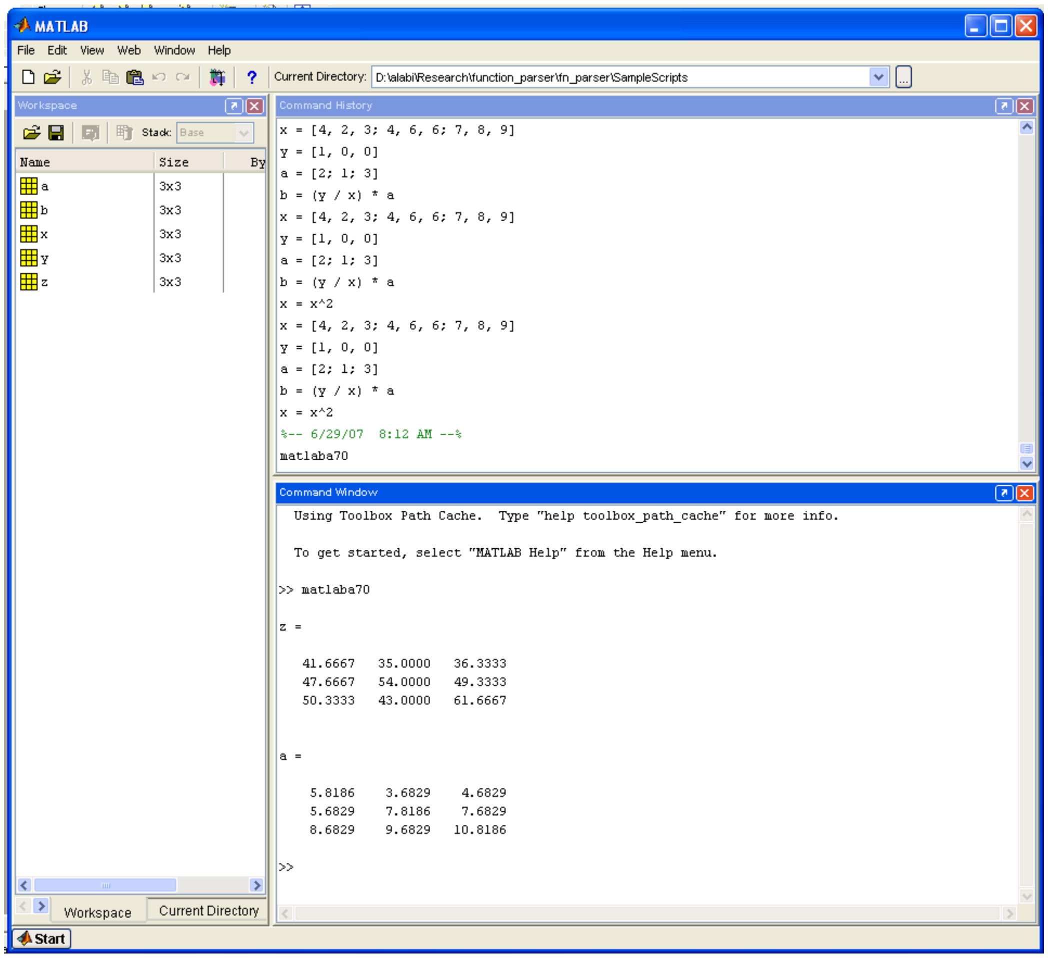
The MATLAB script is shown below.

```
x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x + 2*y/x
%z = 2*y/x;
a = x + 2 * sin(y)
```

Note that Lines 4 and 6 have the ending ";" removed so that MATLAB would print an output to screen. The file containing this script is *matlaba70.m* in the subfolder /*SampleScripts/MATLABprograms* of the iSCRIPT installation folder. There are two ways to run this script in MATLAB:

1. Method 1
    - Open the MATLAB program.
    - Simply copy and paste the above script (or from the open *matlaba70.m* file) into the MATLAB command window.
    - The output values of the matrix *z* and are printed out.

2. Method 2
    - Open the MATLAB program
    - Set the **Current Directory** to the /*SampleScripts/MATLABprograms* subfolder of the iSCRIPT installation folder as shown in Figure 7.1.
    - Type matlaba70 in the MATLAB command window.
    - The output values of the matrix *z* and are printed out.


MATLAB
File Edit View Web Window Help
Current Directory: D:\alabi\Research\function_parser\fn_parser\SampleScripts
Workspace
Stack: Base
Command History
Command Window
Workspace
Current Directory
Start

Name	Size	By
a	3x3	
b	3x3	
x	3x3	
y	3x3	
z	3x3	

```
x = [4, 2, 3; 4, 6, 6; 7, 8, 9]
y = [1, 0, 0]
a = [2; 1; 3]
b = (y / x) * a
x = [4, 2, 3; 4, 6, 6; 7, 8, 9]
y = [1, 0, 0]
a = [2; 1; 3]
b = (y / x) * a
x = x^2
x = [4, 2, 3; 4, 6, 6; 7, 8, 9]
y = [1, 0, 0]
a = [2; 1; 3]
b = (y / x) * a
x = x^2
%-- 6/29/07  8:12 AM --%
matlaba70
```

```
  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, select "MATLAB Help" from the Help menu.

>> matlaba70

z =

   41.6667   35.0000   36.3333
   47.6667   54.0000   49.3333
   50.3333   43.0000   61.6667


a =

    5.8186    3.6829    4.6829
    5.6829    7.8186    7.6829
    8.6829    9.6829   10.8186

>>
```

Figure 7.1. MATLAB program showing results of running *matlaba70.m* script file.

8.5.2 Running the program in iSCRIPT

The iSCRIPT version of the same program is presented below. Notice that the script is *exactly the same*, except that the variables used are declared at the start of the program. This is the most prominent modification that has to be made to MATLAB scripts to run them in iSCRIPT.

```
i,x(3,3),y(3,3),b(3,3) as integer
z(3,3),a(3,3) as real
x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x + 2*y/x;
%z = 2*y/x;
a = x + 2 * sin(y);
```

The file containing this script is *matlaba70.isc* in the subfolder */SampleScripts/MATLABprograms* of the iSCRIPT installation folder. Run this file in iSCRIPT as follows:

1. Open iScript Editor.
2. Select “*File* > *Open*” from the menu.

⇒ The File Open dialog box appears.

3. Navigate to the subfolder /*SampleScripts/MATLABprograms* of the iSCRIPT installation folder.
4. Open the file *matlaba70.isc*.
5. Select “*Tools* > *Run Current Script/Project File*” from the menu.

⇒ The program runs and the output values of the matrix z are printed out.

8.5.3 Running other MATLAB sample programs

Additional examples running scripts created in MATLAB are available in the subfolder /*SampleScripts/MATLABprograms* of the iSCRIPT installation folder and are described in sample problems 12 through 15 in Appendix C.

8.6. Running an Executable (or Third-Party Software)

The executable used for this illustration was actually generated from a MATLAB script. The procedure to compile the script into an executable is also described. The method for interfacing with a third-party software is illustrated in a system modeling context in which one (or all) of the subsystems are modeled in a different software.

Figure 7.2 presents a schematic of the procedure. In the figure, a complete aircraft is modeled, modeling each subsystem in a decomposed fashion. It is assumed that the propulsion subsystem (PS) is modeled in a separate executable file that receives parameter input representing variables, such as the operating point Mach number, altitude, and amount of bleed air extracted for the environmental control subsystem (ECS).The iSCRIPT model integrates the PS program into the complete aircraft model by writing the parameter input into a file in the required format of the PS model, running the PS program, and retrieving the output from the output file in the program’s output format.

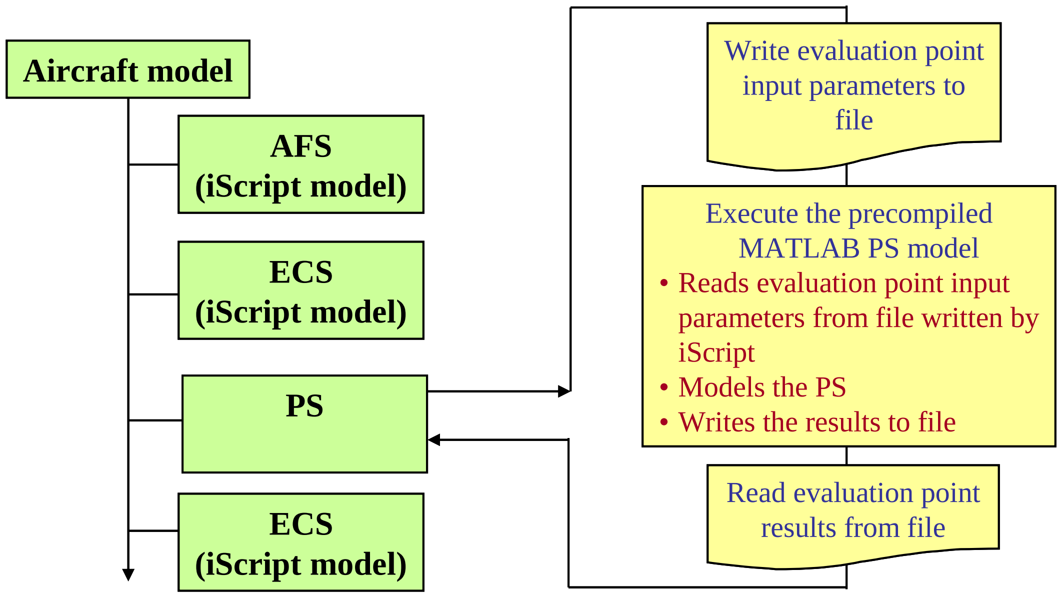

Aircraft model
AFS (iScript model)
ECS (iScript model)
PS
ECS (iScript model)
Write evaluation point input parameters to file
Execute the precompiled MATLAB PS model
• Reads evaluation point input parameters from file written by iScript
• Models the PS
• Writes the results to file
Read evaluation point results from file

Figure 7.2. An aircraft model with the PS subsystem modeled in a software or executable outside of iSCRIPT.

The solution consists of the following files in the */SampleScripts/PS_inMATLAB* subfolder of the iSCRIPT installation folder:

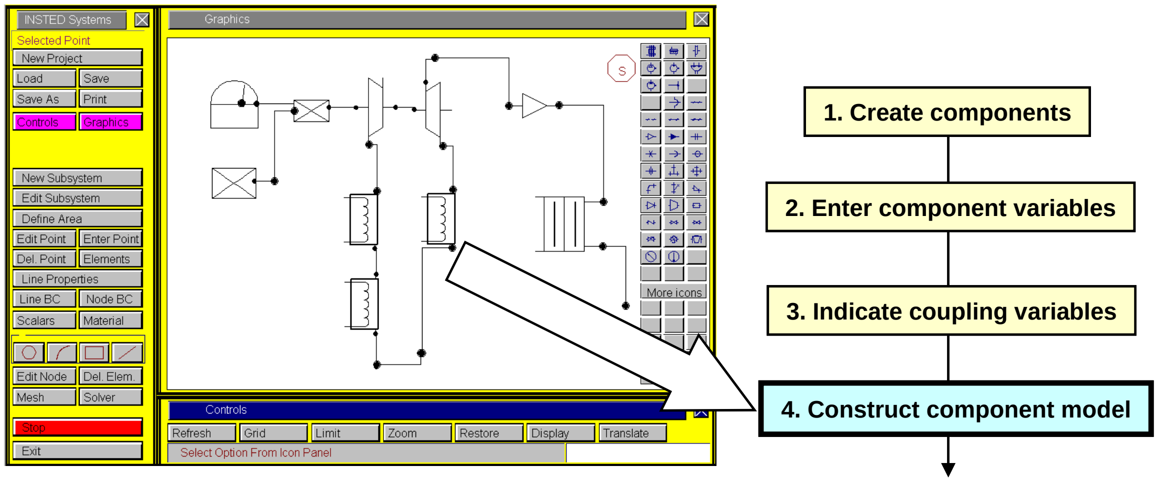
PS_conv.m	a MATLAB program that rates a low-bypass turbofan aircraft engine. The input to the program includes the altitude, Mach number, etc. This program reads the input from a file *PS_input.txt*. The output from the program includes several variables, including the thrust, fuel consumption etc. The output from the program is written to a file *PS_output.txt*.
ata.ipr	an iSCRIPT project file containing three files: *main.isc*, *ps_component.isc*, and *ps_setting.isc*. These files are described below.
Main.isc	a main program defining two components, PS_setting and PS_Component. PS_Setting simply sets the conditions for computing (rating) the aircraft engine.
PS_component.isc	an iSCRIPT component model file. This model file includes commands to execute the MATLAB model. An input file is created for the MATLAB program and the output from MATLAB is read. The output is further used to compute certain quantities, such as the total exergy destruction in the engine.

Procedures for Running this Solution

1. Run the MATLAB program in MATLAB to check the model of the aircraft engine.
2. Compile the MATLAB program into an executable. This step includes issuing the command `mcc –m PS_conv`. An executable named *PS_conv.exe* is generated. This executable was renamed to *PS.exe*
3. Run iSCRIPT. Enter the project file *ata.ipr* at the iSCRIPT prompt.
4. View the results.
5. Note that the model may be optimized further on the high-level using the optimization procedures present in the iSCRIPT program, as described in Chapter 6.

9. Conclusions

Future development of TTC's scripting language will continue to extend the current capabilities, adding more intrinsic functions, e.g., for non-linear analysis, dynamic analysis, solvers etc. The consistent motive would be to create an easy, powerful modeling and design/optimization tool compatible with the scripting languages that engineers use most often. A GUI procedure to utilize the power of the scripting language and optimization procedure in graphic block-building approach will also be separately available. Sample screen of this program is shown below. The underlying code modeling the components and subsystems depicted graphically would be the iSCRIPT platform.



(a) System consisting of several components

(b) Process for developing component model

Figure 9.1. A graphical system building tool to complement iSCRIPT.

Appendix A. iSCRIPT Language Reference

A.1. Variables and Expressions

In stand-alone scripts, variables are declared as follows:

> *var1*, *var2* as *type*
>
> *var1*, *var2* – variable names satisfying the variable naming convention.
> as – declaration keyword
> *type* – may take values: logical, short, long, real, and double

A.1.1. Types of Variables

Logical variables
Logical variables take on values of T or F. In addition, values of 0 or any real number may be assigned to logical variables. A numeric value of 0 will be converted to F prior to assignment, while other numbers will be converted to T.

Short variables
Short variables have values in the range -32,768 to 32,767. They are also type INTEGER(2) in FORTRAN. The syntax for the declaration also allows the use of INTEGER(2) or INTEGER*2 keywords.

Long variables
Long variables have values in the range -2,147,483,648 to 2,147,483,647. They are also type INTEGER in FORTRAN. The syntax for the declaration also allows the use of INTEGER, INTEGER(4) or INTEGER*4 keywords.

Real variables
Holds signed IEEE 32-bit (4-byte) single-precision floating-point numbers ranging in value from -3.4028235E+38 through -1.401298E-45 for negative values and from 1.401298E-45 through 3.4028235E+38 for positive values. The syntax for the declaration also allows the use of SINGLE, REAL(4), or REAL*4 keywords.

Double variables
Holds signed IEEE 64-bit (8-byte) double-precision floating-point numbers ranging in value from -1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values and from 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values. The syntax for the declaration also allows the use of REAL(8) or REAL*8 keywords.

A.1.2. Variable Names

Variable names may be up to 24 characters long and can be alphanumeric. However, variables must not begin with a "_" or a numeral, and must not be separated by spaces. Examples of valid variable names include:

•	ii	Reynolds_No	Ma
•	jNo	iwhat	Total_Value

A.1.3. Numbers

Conventional decimal notation is used, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter e to specify a power-of-ten scale factor. Some examples of legal numbers are

•	7	-82	0.00007
•	7.4382259	1.60210e-20	9.1345e14

All numbers are stored internally using the double type described in Section 2.1 and as specified by the IEEE floating-point standard.

A.1.4. Assignment Operator

The "=" symbol is used as the assignment operator.

```
var1 = 1.002
```

A.1.5. Arithmetic Operators

Arithmetic operators include:

Operator	Function
-	Subtraction. Subtracts a variable, number, or expression on the right from a variable, number, or expression on the left of the operator.
+	Addition. Adds a variable, number, or expression on the right to a variable, number, or expression on the left of the operator.
*	Multiplication. Multiplies a variable, number, or expression on the right to a variable, number, or expression on the left of the operator.
/	Division. Divides a variable, number, or expression on the left by a variable, number, or expression on the left of the operator.

^, **	Power. Raises a variable, number, or expression to the left to a power defined by a variable, number, or expression on the right.
( )	Brackets. Independently evaluate the value contained within the brackets.

The precedence of the operators is as shown from top to bottom, i.e. '**' are computed before '-' when they are in the same expression.

A.1.6. Relational Operators

Relational operators include:

Operator	Function
<	Less than. Compares a variable, number, or expression on the right to an variable, number or expression on the left of the operator. Returns a value of 1 or true if the value on the left is less than that on the right.
>	Greater than. Compares a variable, number, or expression on the right to an variable, number or expression on the left of the operator. Returns a value of 1 or true if the value on the left is greater than that on the right.
<=	Less than or equal. Compares a variable, number, or expression on the right to an variable, number or expression on the left of the operator. Returns a value of 1 or true if the value on the left is less than or equal that on the right.
>=	Greater than or equal. Compares a variable, number, or expression on the right to an variable, number or expression on the left of the operator. Returns a value of 1 or true if the value on the left is greater than or equal that on the right.
= =	Equal. Compares a variable, number, or expression on the right to an variable, number or expression on the left of the operator. Returns a value of 1 or true if the value on the left is equal that on the right.
!=	Not equal. Compares a variable, number, or expression on the right to an variable, number or expression on the left of the operator. Returns a value of 1 or true if the value on the left is not equal that on the right.

A.1.7. Logical Operators

Logical operators include:

Operator	Function
&	LOGICAL AND. Returns a value of 1 or true if the variable, number, or expression on the right is true (or non zero) and the variable, number, or expression on the right is also true.
|	LOGICAL OR. Returns a value of 1 or true if either the variable, number, or expression on the right is true (or non zero) or the variable, number, or expression on the right is true.
~	NOT. Returns a value of 1 or true if the variable, number, or expression on the right is not true (or is zero).

A.1.8. Expressions and Equations

Expressions may be generated as a combination of variables, numbers, and operators. Examples include:

```
Re = rho * U * L / mu

Speed_of_sound = (gamma * P/rho) ^ 0.5

iparameters_provided = Reynolds & Ma
```

iSCRIPT is **case-insensitive** and **free-form**. This also means that empty lines, spaces, and comments can be included as desired without consequence to the performance of the scripts. **This also means that programs can be indented and commented appropriately for easy code maintenance.**

A.1.9. Comments

Comments may be included in a script using the # or % symbols. Comments may occupy a whole line or be included after an expression. In either case, all input following a comment symbol is ignored. Examples include:

```
# The parameters of the flow are computed below
Re = rho * U * L / mu                # Reynolds no. Eqn(1.3)

Speed_of_sound = (gamma * P/rho) ^ 0.5      # Sound speed. Eqn(1.4)
```

A.2. Arrays

Scripts may include arrays of any size or dimensions. Arrays may be declared similar to other variables but must include the size and dimension of the arrays in brackets after the array names. Array sizes must be integer values. Examples include:

```
matrixA(3,3) as real

array1(5,2), inumber as integer
```

A.2.1. Referencing Array Elements

Array elements may be referenced using numbers, variables, or expressions. The type of the number, variable, or expression will be converted to integer at runtime. Examples include:

```
matrixA(1,1) = 12.2

matrixA(1,2) = Reynolds_No

matrixA(1,3) = rho * U * L / mu

matrixA(j + sin(t + r^2), k) = cos(omega * t)
```

Arrays can be included directly in expressions and array arithmetic performed.

A.2.2. Assigning Values to Arrays

Literal values can be directly assigned to array variables. For instance, the segment below generates a 3 x 3 matrix *x*.

```
x(3,3) as integer

x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
```

A.2.3. Matrix Arithmetic

Matrix arithmetic, such as multiplication, divisions, additions, etc., may be performed directly. An example is shown below.

```
i,x(3,3),y(3,3),b(3,3) as integer

z(3,3),a(3,3) as real

x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
```

```
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];

b = [4 1 0; 2 1 2; 5 1 1];

z = y*x + b*x + 2*y/x;

a = x + 2 * sin(y);
```

A.3. Decision Structure

iSCRIPT uses the if-elseif-else-endif statement to implement the conditional execution of segments of a program. The syntax is as shown below:

```
if (expression1) then
    …

elseif (expression2) then
    …

elseif (expression3) then
    …

else
    …

end if
```

The following rules govern the use of the if statement.

- *expression1*, *expression2*, and *expression3* are valid expressions constructed as in Section 4.1.
- … represents one or more lines of scripting language segments (which may include other if statements).
- else if may also be used instead of elseif.
- endif or end may also be used instead of end if.
- It is not mandatory to have the elseif or else portions of the if statement.
- There is no limit to the number of elseif segments that may be included in an if structure.
- There can be only one else statement in an if structure.
- if statements may be nested as desired.

A.4. Loop Structure

iSCRIPT uses the do, for, and while statements to implement the conditional execution of segments of a program. The syntax for each type is described below.

A.4.1. Do Loops

```
do *ii = expression1* : *expression2*
     …

end do
```

The following rules govern the use of the do statement.

- ii is a variable declared as in Section 4.1 (ii may be a short, long, real, or double variable).
- expression1 and expression2 are valid expressions constructed as in Section 4.1.
- … represents one or more lines of scripting language segments (which may include other do statements) and is referred to as the body of the loop.
- ii is incremented by 1 and the body of the loop executed until expression1 is greater than expression2.
- The body is not executed at all if expression1 is greater than expression2 at the start of the loop.
- enddo or end may also be used instead of end do.
- do statements may be nested as desired.

A.4.2. For Loops

```
for *ii = expression1* : *expression2*
     …

end for
```

The following rules govern the use of the for statement.

- ii is a variable declared as in Section 4.1 (ii may be a short, long, real, or double variable).
- expression1 and expression2 are valid expressions constructed as in Section 4.2.
- … represents one or more lines of scripting language segments (which may include other for statements) and is referred to as the body of the loop.
- ii is incremented by 1 and the body of the loop executed until expression1 is greater than expression2.
- The body is not executed at all if expression1 is greater than expression2 at the start of the loop.
- endfor or end may also be used instead of end for.
- for statements may be nested as desired.

A.4.3. While Loops

```
while (expression1)
    …

end
```

The following rules govern the use of the while statement.

- expression1 is a valid expressions constructed as in Section 4.2.
- … represents one or more lines of scripting language segments (which may include other for statements) and is referred to as the body of the loop.
- The body of the loop is executed until expression1 evaluates to false or 0.
- The body is not executed at all if expression1 is false or evaluates to 0 at the start of the loop.
- while statements may be nested as desired.

A.5. Subprogram and Function

iSCRIPT allows the use of subprograms to introduce program structure and allow the organization of parts of the model. For instance a subroutine or function may be generated separately and called multiple times to perform a specific purpose. Subprograms are useful in creating codes or models that are easily maintained, and help to avoid rewriting whole segments of code that may be required more than one time.

Scripts may be created without any particular start or end program indicator. In this case, the entire script is assumed to be one program and subprograms can not be used. To use subroutines and subfunctions, a start and end program indicator must separate the program and start and end subprogram indicators must also be used. All scripting elements outside of these demarcators are ignored. The exact syntax for structure limiters are described in this chapter.

A.5.1. Program Structure

```
program programname
    variable declaration statements
    …


end program
```

The following rules govern the use of program structure statements.

- programname is the name of the program and must be contrived according to variable naming conventions described in Section A.1.2.
- variable declaration statements represents several lines of variable declaration statements as described in Section A.1.1.
- … represents one or more lines of scripting language segments and is referred to as the body of the program.
- endprogram may also be used instead of end program.
- There can be only one program in a model of a component.

A.5.2. Subroutine Structure

```
subroutine subroutinename (arg1, arg2, …, argN)
    variable declaration statements
    …


end subroutine
```

The following rules govern the use of the subroutine structure statements.

- subroutinename is the name of the subroutine and must be named according to variable naming conventions described in Section A.1.2.
- variable declaration statements represents several lines of variable declaration statements as described in Section A.1.1.
- *arg1*, *arg2*, …, *argN* are variables or arrays named according to conventions described in Section A.1.2 and are called the dummy arguments to the subprogram.
- The dummy arguments to the subprogram must be declared in addition to any other variable declared within variable declaration statements.
- … represents one or more lines of scripting language segments and is referred to as the body of the subprogram.
- endsubroutine may also be used instead of end subroutine.

A.5.3. Calling a Subroutine

A subroutine may be called within a program, other subroutine, or function. A subroutine may also be called recursively. Subroutines may be called using the call keyword.

```
call subroutinename (arg1, arg2, …, argN)
```

The following rules govern the process of calling a subroutine.

- subroutinename is the name of the subroutine and must be the same as that used in the subroutine keyword in A.5.2.

- arg1, arg2, …, argN are variables or arrays named according to conventions described in Section A.1 and are called the actual arguments to the subprogram.
- The names of the actual arguments may be different from those of the dummy arguments in Section A.5.2. However, the type and array sizes must match if the actual and dummy arguments are arrays.
- The program or subprogram within which the above statement is placed is referred to as the calling program or subprogram.
- The actual arguments to the subprogram must be declared in addition to any other variable declared within the calling program or subprogram.
- At runtime the values of the actual arguments, if modified within the subroutine, is returned to the calling program or subprogram.

A.5.4. Function Structure

```
function functionname (arg1, arg2, …, argN)
   variable declaration statements
   …


end function
```

The following rules govern the use of the function structure statement.

- functionname is the name of the function and must be named according to variable naming conventions described in Section A.1.
- variable declaration statements represents several lines of variable declaration statements as described in Section A.1.
- arg1, arg2, …, argN are variables or arrays named according to conventions described in Section 4.1 and are called the dummy arguments to the subprogram.
- The dummy arguments to the subprogram must be declared in addition to any other variable declared within variable declaration statements.
- functionname must be declared in addition to any other variable declared within variable declaration statements. functionname **may be declared as an array**.
- … represents one or more lines of scripting language segments and is referred to as the body of the subprogram.
- endfunction may also be used instead of end function.

A.5.5. Calling a Function

A function may be called within a program, other subroutine, or function. A function may also be called recursively. **Functions may be called simply using them in an expression in lieu of a variable or array**.

var = functionname (*arg1*, *arg2*, …, *argN*)

Examples

Function Calls	Actual Function

…
Re = Reynolds_No (*rho, U, L, mu*)
…
Re_is = Reynolds_No (*rho, U, L, mu*)^gm
…
	Function Reynolds_No (r, V, L, mu)
	    r, V, L, mu as real
	    Reynolds_No = r * V * L / mu
	End Function

The following rules govern the process of calling a function.

- var is the name of a variable or array named according to conventions described in Section A.1.
- functionname is the name of the subroutine and must be the same as that used in the function keyword in Section A.5.2.
- arg1, arg2, …, argN are variables or arrays named according to conventions described in section A.1 and are called the actual arguments to the function.
- The names of the actual arguments may be different from those of the dummy arguments in Section A.5.2. However, the type and array sizes must match if the actual and dummy arguments are arrays.
- The program or subprogram within which the above statement is placed is referred to as the calling program or subprogram.
- The actual arguments to the function must be declared in addition to any other variable declared within the calling program or subprogram.
- At runtime the values of the actual arguments, if modified within the function, is returned to the calling program or subprogram.
- At runtime, the value of the functionname as a variable is returned to the calling program or subprogram and used to evaluate the expression to the right of the assignment symbol.

A.5.6. Return

When used in a function or subroutine, the return statement acts in exactly the same way as the end function or end subroutine statements. A sample of syntax is illustrated below:

```
subroutine subroutinename (arg1, arg2, …, argN)
    variable declaration statements
    …

    return
    …

end subroutine
```

In the above syntax, on encountering the return syntax, the subroutine is ended and rest of the subroutine is not executed. Used within decision statements, the return keyword may be used to conditionally end the execution of a subprogram when certain outcome has been attained.

A.5.7. Argument Passing Convention

Arguments are passed by reference in iSCRIPT similar to FORTRAN. However, if MATLAB syntax is selected, arguments are passed by value. Expressions and literal values are passed by value. Global and component variables are also passed by value (since they are global variables and do not need to be passed into subprograms if their values are intended to change in any subprogram).

A.6. Other Program Flow Structure

For compatibility with other engineering programming tools, iSCRIPT supports additional syntax including the break and continue keywords as well as labels and go to statements.

A.6.1. Break

This keyword is used only for scripts indicated as MATLAB source. The break statement terminates the execution of a loop segment. In nested loops, the break statement only exits the loop within which it occurs.

Examples:

```
for ii = expression1 : expression2
    …
    break
    …
end for
```

```
for ii = expression1 : expression2
    …
    if (expression3) then
            …

            break
            …
    end if
    …
```

```
end for
```

Both examples above cause the premature termination of the for loop on encountering the break statement. In the second case, the termination only occurs on the condition of expression3.

A.6.2. Continue

This keyword is used only as a place holder. For instance, the continue keyword may be used to establish a label. The presence of the continue keyword has no effect whatsoever in a scripting segment. Examples are included in A.6.3.

A.6.3. Go to and labels

The goto statement is used to influence program flow. This statement is used in conjunction with a label statement. The syntax is shown below.

```
…
goto :label
…
:label
```

The rules governing goto and label statements are as follows:

- label is an alphanumeric word defined according to the rules for naming variables as described in Section A.1. label words must not be declared.
- "goto" or "go to" may be used.
- … represents one or more lines of scripting language segments within the same program or subprogram.

Example 1 (using the continue keyword):

```
Re = rho * U * L / mu

If (Re <= 2500) then
    f = 16/Re
  go to :2000
end if

f = 0.0064 *Re ^ 0.4

:2000 continue
```

Example 2 (using a label with an expression):

```
Re = rho * U * L / mu
```

```
If (Re <= 2500) then
  go to :2000
end if

f = 0.0064 *Re ^ 0.4

:2000       f = 16/Re
```

A.7. Intrinsic Functions

Below is a list of supported intrinsic functions. Their arguments and characteristics are the same as their FORTRAN equivalents. This list is constantly increasing. Please check our website for an updated list at any time.

1. cos
2. sin
3. tan
4. exp
5. log
6. log10
7. sqrt
8. acos
9. asin
10. atan
11. cosh
12. sinh
13. tanh
14. anint
15. aint
16. abs
17. real
18. dble
19. alog10
20. alog

21 sizeof
22 length
23 sum
24 avg
25 min
26 max

A.8. Input/Output

iSCRIPT includes commands for input/output to screen, keyboard, and files. The commands are described in this section.

A.8.1. Opening a File

A file may be opened using the open command. The syntax is as shown below.

```
call open (unit, ‘filename’, [‘permission_mode’], arg1, arg2, …, argN)

or

unit =  open (‘filename’, [‘permission_mode’], arg1, arg2, …, argN)
```

The following rules govern the process of calling the *open* command.

- unit is an integer between 10 and 100 provided as a handle for opening the file. This handle should be used when reading from or writing to the file. When the unit is an output, iSCRIPT opens the file on an available unit and supplies the unit as the function output.
- filename is a string representing the name of the file to be opened and must be in accordance with the file naming rules on the operating system. Filename must be enclosed in single quotes.
- fopen may be used instead of the open keyword.
- Permission_mode is one of the options specified in the table below:

Permission Mode Specifiers

Permission_mode	**Description**
	Text Mode
'rt'	Open file for reading (default).
'wt'	Open file, or create new file, for writing; discard existing contents, if any.
'at'	Open file, or create new file, for writing; append data to the end of the file.
'rt+'	Open file for reading and writing.
'wt+'	Open file, or create new file, for reading and writing; discard existing contents, if any.
'at+'	Open file, or create new file, for reading and writing; append data to the end of the file.
	Binary Mode
'r'	Open file for reading (default).
'w'	Open file, or create new file, for writing; discard existing contents, if any.
'a'	Open file, or create new file, for writing; append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open file, or create new file, for reading and writing; discard existing contents, if any.
'a+'	Open file, or create new file, for reading and writing; append data to the end of the file.

A.8.2. Closing a File

An open file may be closed using the close command. The syntax is shown below.

```
call close (unit)

or

ivar =  close (unit)
```

The following rules govern the process of calling the *close* command.

- unit is an integer between 10 and 100 representing the handle for the open file.
- The output of the close command is 0 if successful and -1 if an error occurred. The exact error based on the operating system is written in the iSCRIPT log file.
- fclose may be used instead of the open keyword.

A.8.3. Reading from a File or the Keyboard

The syntax to read from an open file or the standard input (usually the keyboard) is shown below.

call *read* ([unit], [‘format’], [*arg1*], [*arg2*], …, [*argN*])

call *read* ([unit], [‘format’]) [*arg1*], [*arg2*], …, [*argN*]

or

A = *read* ([unit], [‘format’], [isize])

A = *fscanf* ([unit], [‘format’], [isize])

The following rules govern the process of calling the *read* command.

- unit is an integer between 10 and 100 representing the handle for the open file. No unit specified, units 1, 2, 5, or 6 refers to the keyboard.
- format is a string representing the read format. The format string may be omitted (simply provide an empty “,”) or an * used instead. The format string when read is currently ignored but is accepted for compatibility with future versions of iSCRIPT. The MATLAB format specifiers are accepted.
- fscanf may be used instead of the *read* keyword except when no argument is provided (empty read).
- arg1, arg2, …, argN are strings, variables or arrays. Strings must be enclosed in single quotes.
- *A* is a variable or array.

- When specified, isize refers to the total number of elements that should be read. When isize exceeds the size of the array, the size of the array is used.
- If an end of file occurs during a read, the command returns and the program resumes. An internal flag is set which may be queried using the eof command ((see Section A.8.6).

A.8.4. Writing to a File or the Screen

The syntax to write to an open file or the standard output (usually the screen) is shown below.

```
call write ([unit], [‘format’], [arg1], [arg2], …, [argN])

call write ([unit], [‘format’]) [arg1], [arg2], …, [argN]

or

isize =  write ([unit], [‘format’], [arg1], [arg2], …, [argN])

isize =  fprintf ([unit], [‘format’]) [arg1], [arg2], …, [argN]
```

The above rules govern the process of calling the write command.

- unit is an integer between 10 and 100 representing the handle for the open file. No unit specified, units 1, 2, 5, or 6 refers to the keyboard.
- format is a string representing the read format. The format string may be omitted (simply provide an empty “,”) or an * used instead. The format string when read is currently ignored but is accepted for compatibility with future versions of iSCRIPT. The MATLAB format specifiers are accepted.
- fprintf may be used instead of the write keyword.
- *arg1*, *arg2*, …, *argN* are strings, variables or arrays. Strings must be enclosed in single quotes.
- isize is a value returned representing the number of bytes or characters written.

A.8.5. Rewinding a File

An open file may be returned to the start of file using the close command. The syntax is shown below.

```
call rewind (unit)

or

ivar =  rewind (unit)
```

The following rules govern the process of calling the close command.

- unit is an integer between 10 and 100 representing the handle for the open file.
- The output of the rewind command is 0 if successful and -1 if an error occurred. The exact error based on the operating system is written in the iSCRIPT log file.
- frewind may be used instead of the open keyword.

A.8.6. End-of-File Function

An end of file (eof) command may be called to determine if end of file occurred during the last call to the read command on a specified file open handle. The syntax is as follows:

ivar = *eof* (unit)

The following rules govern the process of calling the *close* command.

- unit is an integer between 10 and 100 representing the handle for the open file.
- The output of the eof command is 1 if end-of-file occurred or 0 otherwise.
- feof may be used instead of the eof keyword.

A.9. Object-Oriented Features and Component Modeling

iSCRIPT has object-oriented features that allows you to define component objects (or structures) and variables attached to those structures. In iSCRIPT, the structures can be created as components and the properties of the component are referred to as component variables. However, unlike normal structures, all components automatically support the Component.Execute method. The syntax to define an object and the properties or variables of the objects is described below.

Note that a global component exists for every project as described in Section A.9.3. The global component has no execute file.

A.9.1. Defining a Component

An object may be defined using the following syntax:

CreateComponent (name [,description])

Note: Segments enclosed in square brackets are optional and may be omitted.

name –	A name for the component (a string limited to 24 characters). Two components may not have the same name. Component names obey the same formation rules as those for variables.
description –	A description for the component (a string limited to 50 characters). Optional.

NOTE: THERE IS NO NEED TO ENCLOSE CHARACTER ARGUMENTS IN QUOTES FOR THIS COMMAND.

Example:

```
program main
        T_in, T_out as real
        CreateComponent (Heat_Ex1)
        CreateVariable (Heat_Ex1, Tin)
        CreateVariable (Heat_Ex1,Tout)
        T_in = 286.16
        Heat_Ex1.Tin = T_in
        …
        …
```

A.9.2. Defining a Component Variable

A component variable may be defined using the following syntax:

```
CreateVariable (component, name [,type] [,dimension] [,size]
                  [,upper_bound] [,lower_bound] [,default_value] [,unit])
```

Note: Segments enclosed in square brackets are optional and may be omitted.

component –	The component to which the variable belongs (a string limited to 24 characters). Two component variables may not have the same name. Component variable names obey the same formation rules as those for variables.
name –	A name for the component (a string limited to 24 characters). Two components may not have the same name. Component names obey the same formation rule as those for variables.
type –	A string accepting values such as “integer,” “real,” “double.” A complete list of variable types can be found in Section A.1. This argument is optional. When not provided, component variables are assumed to be double values.
dimension –	Variable dimension for an array variable (integer). For example, a 2D matrix will have a dimension of 2. This argument is optional for scalar variables (dimension = 0 is default).

size –	Variable size for an array variable. This argument accepts an integer array with a limit of five integers. For example, a 3 x 3 matrix will have a size of (3,3). This input must be enclosed in brackets. This input is required when dimension > 0.
upper_bound –	An upper bound for the variable (all the variables for an array variable). The type of this argument depends on type. This argument is optional.
lower_bound –	A lower bound for the variable (all the variables for an array variable). The type of this argument depends on type. This argument is optional.
default_value –	A default value for the variable (all the variables for an array variable). The type of this argument depends on type. This argument is optional.
unit –	A string representing the engineering unit used in providing the variable values (e.g., m/s). This argument is also optional and when provided is limited to 20 characters.

NOTE: THERE IS NO NEED TO ENCLOSE CHARACTER ARGUMENTS IN QUOTES FOR THIS COMMAND.

A.9.3. Executing a Component

A component may be executed using the following syntax:

```
Component_name.execute

Or

Call Component_name.execute
```

Component_name – Character(24). The component name as defined in 4.9.1.

The execute routine must be a subroutine with the same name as Component_name **and require no arguments.**

A.9.4. Using Global Variables in the In-Built Global Component

Global variables may be created only in the main program. Global variables do not need to be created using the CreateVariable command (although this command may be used as well). Instead, global variables may be created simply by prefixing a declaration with the Global keyword, as in Section 2.1.

```
Global Re, Ma as real
```

Global variable names must be unique among variable names but may coincide with a local variable name. Reference to global variables is similar to that for all components, as illustrated in the example below:

```
program

         global emCp, Q1 as real
         localemCp, Q, r as real


          global.emCp = 209.4

          r = 4.0

          localemCp = global.emCp


 end program
```

Appendix B. iSCRIPT Optimization Reference

B.1. Design/Optimization Analysis Procedures

iSCRIPT provides functions and procedures to optimize components and systems. The functions utilize a combination of genetic and gradient-based algorithms. The integrated local global optimization (ILGO) procedure is a powerful option used to optimize a system consisting of several sub-systems. This procedure allows the optimization of large systems within a feasible time-frame, as compared to procedures that utilize nested optimization loops through several component optimization levels. The procedure is illustrated below:

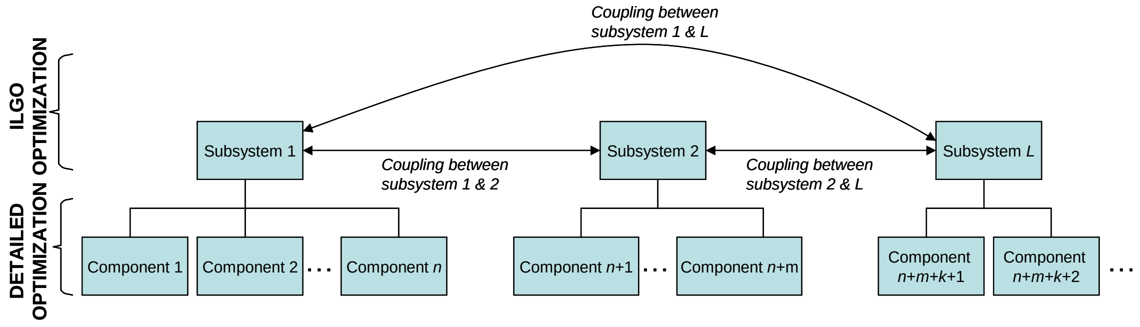


Figure 5.1. Optimization procedure.

DETAILED OPTIMIZATION

When an optimization command is invoked on any component, the system will launch a detailed optimization based on a combination of genetic and gradient-based algorithms. The relationship between sub-systems, systems, and components is utilized for optimization. The optimization free variables are determined from the variables of the component being optimized as well as other components flagged as its subcomponent. Essentially, subsystems are simply components that consist of several other components by virtue of the model equations (its model equations consist of the declaration of other components). If this component is a sub-system that can be defined integrally and separately from other sub-systems, the optimization procedure will proceed faster by indicating that the component is a subsystem and interacts with other subsystems via a finite and few number of variables. Then, the subsystem is optimized in detail. This procedure will proceed faster than calling an optimization command on each component and then calling an overall optimization command on the subsystem.

ILGO OPTIMIZATION (GLOBAL OPTIMIZATION)

When a project has been defined into several subsystems, each consisting of several components, as illustrated in Figure 5.1, an optimization command may be called at the project level. In this case, iSCRIPT will perform a detailed optimization of each subsystem and an overall optimization of the entire system (project) utilizing the sub-system level coupling between the sub-systems.

B.2. Procedures for Performing a Detailed Optimization in iSCRIPT

To optimize a component, the following information must be provided:

- The objective variable (from the list of component variables). Note that the equation to solve an objective function is contained within the component model in iSCRIPT and the result of evaluating the objective function is the objective variable, f, say, as:
  $\mathbf{H}(\mathbf{x}) = 0$
  based on component variables: $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ of n variables
  $f = f(\mathbf{x})$
  Optimize w.r.t. f.
  Then the component variables are: $\mathbf{x} = [x_1, x_2, \ldots, x_n, f]$ or n+1 variables, and the component model additionally includes the equation:
  $f - f(\mathbf{x}) = 0$.
  Note that iSCRIPT optimization functions are multi-objective and f can be a vector of objectives f_i.

- The list of variables in **x** that are free for optimization. Note that the fewer variables have a degree of freedom with respect to optimization, the faster the optimization will complete.

- The list of components encompassed within the component (which is technically a subsystem) to be optimized (to perform a subsystem level detailed optimization).

- The optimization command is invoked for the component.

Note that the model equation for the component may consist of execution and optimization commands for other components resulting in nested optimization loops. Care must be taken when setting up nested optimization loops, as the time required increases geometrically with the number of nesting.

The procedure for providing the above information is described in subsequent sections.

B.2.1. Indicating an Objective Variable

An objective variable may be defined using the following syntax:

AddObjective (component, variable [,maxmin])

Note: Segments enclosed in square brackets are optional and may be omitted.

component --	The component to be optimized (a string limited to 24 characters). This component must be a component previously declared with the CreateComponent command.
variable --	Name of the variable (a string limited to 24 characters). This variable must be a variable previously declared for this component using the CreateVariable command
maxmin --	0 or 1. Indicates whether this is a minimization or maximization objective. Use 0 to minimize this variable and 1 to obtain a maximum. This argument is optional. The default value is 0.

NOTE: THERE IS NO NEED TO ENCLOSE CHARACTER ARGUMENTS IN QUOTES FOR THIS COMMAND.

Example:

```
CreateComponent (Heat_Ex1)
CreateVariable (Heat_Ex1, Tin)
CreateVariable (Heat_Ex1,Tout)
CreateVariable (Heat_Ex1, Length)
CreateVariable (Heat_Ex1,Width)
CreateVariable (Heat_Ex1,Weight)
AddObjective(Heat_Ex1,Weight,0)
…
…
```

B.2.2. Indicating an Free Variable for Optimization

A variable to be varied in the search for an optimum is indicated as follows:

AddVarObjective (component, variable [,delta])

Note: Segments enclosed in square brackets are optional and may be omitted.

component --	The component to which the variable belongs (a string limited to 24 characters). This component must be the component to be optimized or a sub-component of it. This

component must be a component previously declared with the CreateComponent command.

variable --	Name of the variable (a string limited to 24 characters). This variable must be a variable previously declared for this *component* using the CreateVariable command.
delta --	This variable further narrows the optimization search space for this domain to $\left[MAX(L, x_i - \Delta), MIN(M, x_i + \Delta)\right]$ from [*L*, *M*], where *L* and *M* are the lower and upper bounds for *variable*, x_i, as defined in CreateComponentVariable, x_i is the current values of the variable, and *Δ* is *delta*. It is useful to further reduce the search space after a prior optimization step or after an initial computation based on initial conditions. This variable is also used internally to narrow the optimization search space in the ILGO procedure (discussed later) following the first ILGO step. The type of the argument depends on the type of the variable.

NOTE: THERE IS NO NEED TO ENCLOSE CHARACTER ARGUMENTS IN QUOTES FOR THIS COMMAND.

Example:

```
CreateComponent (Heat_Ex1)
CreateVariable (Heat_Ex1, Tin)
CreateVariable (Heat_Ex1,Tout)
CreateVariable (Heat_Ex1, Length)
CreateVariable (Heat_Ex1,Width)
CreateVariable (Heat_Ex1,Weight)
AddObjective(Heat_Ex1,Weight,0)
AddVarObjective(Heat_Ex1,Length)
AddVarObjective(Heat_Ex1,Width)
…
…
```

B.2.3. Indicating Component Relationships

A component may be flagged as contained within another component as follows:

```
AddSubComponent (component, sub-component)
```

component --	A component (subsequently a sub-system) to which another component is contained within (a string limited to 24 characters). This component must be the component to be optimized or a sub-component of it. This component must be a component previously declared with the CreateComponent command.
sub-component --	The component to which belongs to the sub-system (a string limited to 24 characters). This component must be the component to be optimized or a sub-component of it. This component must be a component previously declared with the CreateComponent command.

NOTE: THERE IS NO NEED TO ENCLOSE CHARACTER ARGUMENTS IN QUOTES FOR THIS COMMAND.

Example:

```
CreateComponent (Heat_Ex1)
CreateVariable (Heat_Ex1, Tin)
CreateVariable (Heat_Ex1,Tout)

CreateComponent (Heat_Ex2)
CreateVariable (Heat_Ex2, Tin)
CreateVariable (Heat_Ex2,Tout)

CreateComponent (ECS)
CreateVariable (ECS, Weight)
CreateVariable (ECS, Drag)

AddSubComponent (ECS, Heat_Ex1)
AddSubComponent (ECS, Heat_Ex2)
…
…
```

The indication of a component as belonging to another is only used when an optimization command is called. When the optimization command is called for a component, iSCRIPT will search for optimization variables from the component and all components belonging to it to perform an overall detailed optimization of the component.

B.2.4. Component Optimize Command

A component may be optimized using the following syntax:

```
Component.optimize
```

Component --	A component name (a string limited to 24 characters) as defined in A.10.1.

B.2.5. Procedures for Performing an ILGO Optimization

In addition to the information required for optimizing components, sub-systems require the following information:

- An indication of the subsystems in the project or system (each project is automatically assumed to represent one system).
- The list of components that make up the sub-system (as indicated in Section 11.4).
- An indication of the coupling variables between sub-systems.

The procedure for providing the above information is described in subsequent sections.

B.2.6. Indicating a Subsystem

A subsystem may be indicated using the following command:

AddSubsystem (*component*)

component --	The component to flag as a subsystem (a string limited to 24 characters). This component must be the component to be optimized or a sub-component of it. This component must be a component previously declared with the CreateComponent command.

B.2.7. Indicating Inter-Component (Subsystem) Coupling

A coupling between two components (subs-systems) as follows:

AddCoupling (*component1*, *variable1, component2*, *variable2*)

component1 --	The component to which the variable belongs (a string limited to 24 characters). This component must be the component to be

optimized or a sub-component of it. This component must be a component previously declared with the CreateComponent command.

variable1 --	Name of the variable (a string limited to at most 24 characters). This variable must be a variable previously declared for this *component* using the CreateVariable command.
component2 --	The component to which the variable belongs (a string limited to at most 24 characters). This component must be the component to be optimized or a sub-component of it. This component must be a component previously declared with the CreateComponent command.
variable2 --	Name of the variable (a string limited to at most 24 characters). This variable must be a variable previously declared for this *component* using the CreateVariable command

The above specification is interpreted as:

Component1.Variable1 = Component2.Variable2.

If the coupling is a function, e.g.

Heat_Ex1.Q = 1.2*SecondaryHeat_Ex.Q + MCP(T2 – T1)

An additional component variable may be created – SecondaryHeat_Ex.Q_Couple

In the system model, this variable may be set as:

SecondaryHeat_Ex.Q_Couple = 1.2*SecondaryHeat_Ex.Q + MCP*(T2 – T1)

This variable may then be coupled to Heat_Ex1.Q.

B.2.8. ILGO Optimize Command

A system may be optimized using the following syntax:

System.Optimize

When the system.optimize command is invoked, iSCRIPT searches for every sub-system defined in the project and optimizes each one. Then, an ILGO optimization is performed, as described in Section 11.6, using the coupling variables.

B.2.9. Performing Detailed Optimization at the System Level (without ILGO)

A system-level global optimization may also be performed the traditional way without using the ILGO procedure. This will usually result in nested optimization loop. In this case, subsystems do not need to be indicated and inter component coupling are simply contained within the models. Optimization is called for each component, as well as for the sub-system (which contains components for which optimizations are called). Optimization is also called for an overall system component, which is created and contains every other component. For the system in Figure 6, the calls will be as shown below:

```
Component1 model       Component2 model       Componentn+m+ model
equations              equations              equations
.                      .                      .
.                      .                      .
Component1.Optimize    Component1.Optimize    Componentn+m+.Optimize
```

```
Componentx model       Componenty model       Componentz model
(Subsystem 1)          (Subsystem 2)          (Subsystem L)
equations              equations              equations
.                      .                      .
.                      .                      .
Component1.Execute     Componentn+1.Execute   Componentn+m+k+1.Execute
.                      .                      .
Component2.Execute     Componentn+m.Execute   Componentn+m+k+2.Execute
.                      .                      .
Componentn.Execute     Componenty.Optimize    Componentz.Optimize
Componentx.Optimize
```

```
Componentα model
equations
.
.
Componentx.Execute
.
Componenty.Execute
.
```

Componentz.Execute
Componentα.Optimize

Note that for any fairly detailed system, the above nested optimization arrangement will be very time consuming even if the first level is eliminated and the subsystems are optimized integrally using the free variables from their components.

B.2.10. Optimization Genetic Algorithm

The genetic algorithm used in iSCRIPT is based on a modification of the method of Geoff Leyland. The principles used are as follows:

- Generate an initial population by sampling sparsely over the combinatorial search space of all variables combined.

- Improve the population over a number of generations by combining individuals within the population. Combination is created using the following operators:

    - Selection of combining or mating individuals based on a random selection process weighted to more likely select individuals at the top.

    - Combining the individuals using a blended function of the free variables.

    - Interrupt the process at a low frequency using a mutation operator to ensure that the algorithm does not settle into a non-optimal subspace.

    - Replace only the bottom half of every generation after every combination cycle.

Inherent in the above procedure is a thinning strategy that limits the population size to a specific value (for practical purposes). During both the initial population and improvement phases, new individuals are inserted into the sorted population such that worse individuals drop off once the population size is at a limit. The values of the population limit, initial sampling size, number of generations, and the mutation frequency are variables that affect the genetic algorithm. Default values have been set for these parameters in iSCRIPT but can be modified, as described in the next section.

B.2.11. Optimization Parameters

There are five parameters in iSCRIPT that control the genetic algorithm used for optimization in iSCRIPT. They are implemented internally as global variables with default settings (their values may be reset and altered in any model file). They are:

Parameter	**Default Value**
maxinitialevaluations	1000
maxpopulation	500
maxgenerations	8
optconvergencelimit	0.001
mutationfreq	0.01
maxilgosteps	5

maxinitialevaluations – this parameter limits the initial search space size. Otherwise, the algorithm attempts to sample each variable at 10 points in its search space. For a 10 variable problem, the sample size is potentially approaching a fraction of the number $^{10}C_{10}$. The parameter should be set to a lower number since other combination and mutation operators used in subsequent generations reduces the need to sample excessively for the initial population.

maxpopulation – this parameter limits the overall population size. Otherwise, the algorithm attempts to set a limit of 20 times the number of variables. The population size slows down the genetic algorithm procedure and this parameter can be used to control the population size effectively without compromising the ability of the process to obtain the true optima.

maxgenerations – this parameter the number of improvement generations to run. This parameter is intended to be used if the user wishes to run the algorithm in several cycles effectively restarting a new cycle after maxgenerations. Otherwise, this value should be set at a large number and the convergence limit (discussed next) used to terminate the improvement runs.

optconvergencelimit – the improvement runs are terminated after the individuals in the top half of the population are no different from the previous generations by this value using the $L2$ norm.

mutationfreq – this parameter sets the frequency of mutation per variable. The parameter can be effectively used to control the procedure. For instance, if a specific problem is noted to be very susceptible to local optima (or has a very narrow optimum window), a higher value of the mutationfreq (combined with more generation runs) will ensure that the true optima are found. (Otherwise, note that using high values of mutationfreq will only make it take longer to settle on the optimum value).

maxilgosteps – Similar to maxgenerations, this parameter determines the number of ILGO improvement steps. However, this value is usually small considering that it nests within it several optimization runs within it. A graph of the objective function over the ILGO steps is a good indicator of whether convergence has been reached.

Note that the parameters must be altered according to any global variable in iSCRIPT. For example, the maximum population size can be limited as follows:

```
Global.maxpopulation = 70
```

Appendix C: MORE EXAMPLES OF iSCRIPT SYNTAX

Additional sample problems were developed and used in illustrating specific parts of the scripting language syntax. The sample problems, their purpose and results are presented below. They are also included in */SampleScripts* folder of the iSCRIPT installation. You may use these sample problems to gain familiarity with iSCRIPT syntax or copy any portion of the files for use in your own script.

Sample Problem 1.
This problem illustrates the use the declaration of variables, the reading and interpretation of expressions and the results. Notice that the program is free flow and without a start or end program indicator. iSCRIPT is able to execute free flow scripts without any particular program structure or subroutines.

Model

$$T = \left(2.2901 \times 10^{-12} alt^3 + 8.60446 \times 10^{-8} alt^2 - 6.82246 \times 10^{-8} alt + 31.815\right)$$

$$P = \left(9.63714 \times 10^{-19} alt^4 - 1.18488 \times 10^{-13} alt^3 + 5.52991 \times 10^{-9} alt^2 - 1.18225 \times 10^{-4} alt + 1\right)101325$$

$$\mu = \left(1.61988 \times 10^{-7} T^3 - 4.66143 \times 10^{-4} T^2 + 7.25242 \times 10^{-1} T + 4.10454\right) \times 10^{-7}$$

$$\rho = \frac{P}{287T}$$

where *alt* is the altitude in m, *T* the temperature in K, *P* the pressure in Pa, and μ the viscosity in Ns/m^2, and ρ the density in kg/m^3. The input value of *alt* is supplied in ft (*alt_1*) and has to be initially converted to m in the script below.

Input: File *equations2.isc*

```
alt    T    P    mu    rho    alt_1
alt_1 = 3000
alt = alt_1 * 0.3048
T   = (2.29013E-12*alt*alt*alt+8.60446E-08*alt*alt-6.82246E-03*alt+3.18150E+02)
P   = 101325*(9.63714E-19*alt*alt*alt*alt - 1.18488E-13*alt*alt*alt)
P   = P + 101325*(5.52991E-9*alt*alt- 1.18225E-4*alt + 1)
mu  = (1.61988E-7*T*T*T-4.66143E-4*T*T+7.25242E-1*T+4.20454)*1e-7
rho = P/287/T
```

Output

The output is presented below. Only the print-out of the final value of all variables is presented. A hand calculation may be used to confirm the accuracy of the parsed results.

```
FINAL VALUES OF VARIABLES
 ========================
alt              =   914.4000
 t               =   311.9852
 p                =   90830.66
 mu                =  1.9001649E-05
 rho               =   1.014417
 alt_1              =   3000.000
```

Sample Problem 2.
This script is used to illustrate the use of if statements, nested if statements, and if statements with multiple brackets and single variable expression as the condition.

Model

$$\mathrm{Re} = \frac{\rho U L}{\mu}$$

$$f = \frac{64}{\mathrm{Re}} \qquad \textit{if}\ \mathrm{Re} < 2500$$

$$f = 16\,\mathrm{Re}^{-0.4} \qquad \textit{if}\ \mathrm{Re} \geq 2500$$

$$f = 16\,\mathrm{Re}^{-0.4} \quad \textit{if}\left(-\mathrm{Re}\ \mathrm{AND}\, L = 5\right)$$

Input: File *equations4.isc*

```
rho, v, L, mu, Re, f
rho = 1.05
v = 17.0e-4
L = 5
mu = 8.5E-5
Re = rho * v * L / mu
if (Re < 2500) then
        f = 64.0/Re
end
if (((Re >= 2500))) then
        f = 16.0 * Re ^ (-0.4)
end
if (-Re) then
        if (L == 5) then
                f = 0.06 * Re ^ (-0.4)
        end if
end

L = 4
```

Output

```
FINAL VALUES OF VARIABLES
 =========================
rho            =   1.050000
 v             =  1.7000000E-03
 l             =   4.000000
 mu              =  8.5000000E-05
 re            =   105.0000
 f             =  9.3255732E-03
```

Sample Problem 3.
This script is used to illustrate the use of do and for loops. For loops are treated equivalently as do loops for compatibility with MATLAB syntax. Comments and in-script documentation are also illustrated.

Model

$$\mathrm{Re} = \frac{\rho U L}{\mu}$$

$$f = \frac{64}{\mathrm{Re}} \qquad \mathit{if}\ \mathrm{Re} < 2500$$

$$f = 16\,\mathrm{Re}^{-0.4} \qquad \mathit{if}\ \mathrm{Re} \geq 2500$$

$$f = 16\,\mathrm{Re}^{-0.4} \qquad \mathit{if}\left(-\mathrm{Re}\ \mathrm{AND}\ L = 5\right)$$

$$\mathit{loop} : i = 1\ \mathrm{to}\ 2L\begin{bmatrix} f = f + 0.01 \\ \mathrm{Re} = \mathrm{Re} + 0.01 \end{bmatrix}$$

Input: File *equations6.isc*

```
 rho, v, L, mu, Re, f, i as real

# Initial Values

 rho = 1.05
 v = 17.0e-4
 L = 5./2.
 mu = 8.5E-5

# Models

 Re = rho * v * L / mu


 if (Re < 2500) then       % Testing an if segment
        f = 64.0./Re
 end
 if (((Re >= 2500))) then  % Testing a nested if segment
        f = 16.0 * Re ^ (-0.4)
 end
 if (-Re) then             % testing a logical statement
        if (L == 5) then
                f = 0.06 * Re ^ (-0.4)
        end if
 end

 L = 4   # Reset L

 for   i =  1:2*L          % testing a loop segment
        f = f  +   0.01
        Re = Re  +  0.01
 end

# Comment line here

 L = 5
```

Output

```
FINAL VALUES OF VARIABLES
 ========================
rho              =   1.05000000000000
 v               =  1.700000000000000E-003
 l               =   5.00000000000000
 mu              =  8.500000000000001E-005
 re              =   52.5800000000000
 f               =   1.29904761904762
 i               =   9.00000000000000
```

Sample Problem 4.
This script is used to illustrate the use of functions and subroutines. In addition, expressions within subprograms are also illustrated.

Model

$$\mathrm{Re} = \frac{\rho U L}{\mu}$$

$$f = \frac{64}{\mathrm{Re}} \qquad \mathit{if}\ \mathrm{Re} < 2500$$

$$f = 16\,\mathrm{Re}^{-0.4} \qquad \mathit{if}\ \mathrm{Re} \geq 2500$$

$$\mathit{if}\,(\mathrm{Re})$$

$$\mathit{if}\,(-\mathrm{Re})$$

$$0.06\,\mathrm{Re}^{-0.4} \qquad \mathit{if}\ L = 5$$

$$0.06\,\mathrm{Re}^{-0.4} \qquad \mathit{if}\ L = 4$$

$$loop : i = 1\,\mathrm{to}\,2L \begin{bmatrix} f = f + 0.01 \\ \mathrm{Re} = \mathrm{Re} + 0.01 \end{bmatrix}$$

Input: File *equations6d.isc*

```
program

rho, v, L, mu, Re, f, i as real

# Initial Values

 rho = 1.05
 v = 17.0e-4
 L = 5
 mu = 8.5E-5

# Models

 Re = Reynolds(rho, v, L, mu)

 mu = 8.5E-5
 call Computef(Re,f)
 if (Re) then            % testing a logical statement
 if (-Re) then            % testing a logical statement
        if (L == 5) then
                f = 0.06 * Re ^ (-0.4)
        end if
        if (L == 4) then
                f = 0.06 * Re ^ (-0.4)
        end if
 end
 end

 L = 4   # Reset L

 for   i =  1:2*L         % testing a loop segment
        f = f  +   0.01
        Re = Re  +  0.01
 end

# Comment line here

  L = 5

 end program


 function Reynolds(rho,u,L,mu)
        rho, u, L, mu, Reynolds  as real
        Reynolds = rho * u * L / mu
        u = 12.2
 end function


 subroutine Computef(Re, f)
        Re, f as real
        if (Re < 2500) then       % Testing an if segment in a subroutine
                f = 64.0/Re
        end
        if (((Re >= 2500))) then  % Testing a nested if segment in a subroutine
                f = 16.0 * Re ^ (-0.4)
        end
 end subroutine
```

Output

```
FINAL VALUES OF VARIABLES
 ==========================
 Logical Variables          0
 Integer2 Variables          0
 Integer Variables          0
 Real Variables          7
 rho                    =   1.050000
 v                      =   12.20000
 l                      =   5.000000
 mu                     =  8.5000000E-05
 re                     =   105.0800
 f                      =  8.9325570E-02
 i                      =   9.000000
 Double Variables          0
```

Sample Problem 5.
This script is used to illustrate the use of while statements and elseif and else statements.

Model

$$\mathrm{Re} = \frac{\rho U L}{\mu}$$

$$\begin{array}{ll} f = \dfrac{64}{\mathrm{Re}} & if\ \mathrm{Re} < 2500 \\ f = 16\,\mathrm{Re}^{-0.4} & if\ \mathrm{Re} \ge 2500 \end{array}$$

$$if\,(\mathrm{Re})$$

$$f\,(-\mathrm{Re})$$

$$\begin{array}{ll} f = 0.001 & L = 1 \\ f = 0.002 & L = 2 \\ f = 0.003 & L = 3 \\ f = 0.004 & L = 4 \\ f = 0.06\,\mathrm{Re}^{-0.4} & L = 5 \\ f = 0.006 & L = 6 \\ f = 0.007 & \text{any other value of } L \end{array}$$

$$i = 1$$

$$while : i < 2L \begin{bmatrix} f = f + 0.01 \\ \mathrm{Re} = \mathrm{Re} + 0.01 \\ i = +1 \end{bmatrix}$$

Input File: *equations6f.isc*

```
program

rho, v, L, mu, Re, f, i as real

# Initial Values

 rho = 1.05
 v = 17.0e-4
 L = 5
 mu = 8.5E-5

# Models

 Re = Reynolds(rho, v, L, mu)

 mu = 8.5E-5
 call Computef(Re,f)
 if (Re) then            % testing a logical statement
 if (-Re) then           % testing a logical statement
         if (L == 1) then
                 f = 0.001
         elseif (L == 2) then
                 f = 0.002
         elseif (L == 3) then
                 f = 0.003
         elseif (L == 4) then
                 f = 0.004
         elseif (L == 5) then
                 f = 0.06 * Re ^ (-0.4)
         elseif (L == 6) then
                 f = 0.006
         else
                 f = 0.007
         end if
 end
 end

 L = 4   # Reset L

 i = 1
 while (i <  2*L) then        % testing a while loop segment
         f = f  +   0.01
         Re = Re  +  0.01
         i = i + 1
 end

# Comment line here

  L = 5

 end program


 function Reynolds(rho,u,L,mu)
         rho, u, L, mu, Reynolds  as real
         Reynolds = rho * u * L / mu
         u = 12.2
 end function


 subroutine Computef(Re, f)
         Re, f as real
         if (Re < 2500) then       % Testing an if segment in a subroutine
                 f = 64.0/Re
         end
         if (((Re >= 2500))) then  % Testing a nested if segment in a subroutine
                 f = 16.0 * Re ^ (-0.4)
         end
 end subroutine
```

Output

```
Logical Variables          0
 Integer2 Variables          0
 Integer Variables          0
 Real Variables          7
 rho                    =   1.050000
 v                      =   12.20000
 l                      =   5.000000
 mu                     =  8.5000000E-05
 re                     =   105.0700
 f                      =  7.9325572E-02
 i                      =   8.000000
 Double Variables          0
```

Sample Problem 6.
This script is used to illustrate the use of while statements and elseif and else statements within subroutines and functions.

Model

$$\mathrm{Re} = \frac{\rho U L}{\mu}$$

$$f = \frac{64}{\mathrm{Re}} \qquad \mathit{if}\ \mathrm{Re} < 2500$$
$$f = 16\,\mathrm{Re}^{-0.4} \qquad \mathit{if}\ \mathrm{Re} \ge 2500$$

$$\mathit{if}\,(\mathrm{Re})$$
$$f\,(-\mathrm{Re})$$
$$\begin{array}{ll} f = 0.001 & L = 1 \\ f = 0.002 & L = 2 \\ f = 0.003 & L = 3 \\ f = 0.004 & L = 4 \\ f = 0.06\,\mathrm{Re}^{-0.4} & L = 5 \\ f = 0.006 & L = 6 \\ f = 0.007 & \text{any other value of } L \end{array}$$

$$i = 1$$
$$\mathit{while} : i < 2L \begin{bmatrix} f = f + 0.01 \\ \mathrm{Re} = \mathrm{Re} + 0.01 \\ i = +1 \end{bmatrix}$$

Input: File *equations6g.isc*

```
program

rho, v, L, mu, Re, f, i as real

# Initial Values

 rho = 1.05
 v = 17.0e-4
 L = 5
 mu = 8.5E-5

# Models

 Re = Reynolds(rho, v, L, mu)

 mu = 8.5E-5
 call Computef(Re,f)


# Comment line here

  L = 5

 end program


 function Reynolds(rho,u,L,mu)
          rho, u, L, mu, Reynolds  as real
          Reynolds = rho * u * L / mu
          u = 12.2
 end function


 subroutine Computef(Re, f)
          Re, f, L as real
          i as integer

          L = 5

          if (Re < 2500) then       % Testing an if segment in a subroutine
                   f = 64.0/Re
          end
          if (((Re >= 2500))) then  % Testing a nested if segment in a subroutine
                   f = 16.0 * Re ^ (-0.4)
          end
          if (Re) then            % testing a logical statement
          if (-Re) then           % testing a logical statement
                   if (L == 1) then
                            f = 0.001
                   elseif (L == 2) then
                            f = 0.002
                   elseif (L == 3) then
                            f = 0.003
                   elseif (L == 4) then
                            f = 0.004
                   elseif (L == 5) then
                            f = 0.06 * Re ^ (-0.4)
                   elseif (L == 6) then
                            f = 0.006
                   else
                            f = 0.007
                   end if
          end
          end

          L = 4   # Reset L

          i = 1
          while (i <  2*L) then        % testing a while loop segment
                   f = f  +   0.01
                   Re = Re  +  0.01
                   i = i + 1
          end
 end subroutine
```

Output

```
FINAL VALUES OF VARIABLES
 ==========================
 Logical Variables           0
 Integer2 Variables          0
 Integer Variables           0
 Real Variables              7
 rho                    =   1.050000
 v                      =   12.20000
 l                      =   5.000000
 mu                     =  8.5000000E-05
 re                     =   105.0700
 f                      =  7.9325572E-02
 i                      =  8.0000000E+00
```

Sample Problem 7.
This script is used to illustrate the use of array declaration and the use of arrays within expressions. This program is part of a model which computes the aerodynamic characteristics of an aircraft fuselage. This program is also useful for demonstrating the use of arrays in an iSCRIPT code.

<u>Model</u>

$$jj = 0$$
$$S_{ref} = 325$$
$$V = 856$$
$$b\log ic(2, 2 + 2\,jj) = 1$$
$$ik = b\log ic(2, 2 + 2\,jj)$$

$$\mathrm{Re}_{fuselage} = \frac{\rho UL}{\mu}$$
$$\mathrm{Re}_{cutoff\_sub} = 38.21\left(\frac{l}{k}\right)^{1.053}$$
$$\mathrm{Re}_{cutoff\_\sup} = 44.62\left(\frac{l}{k}\right)^{1.053} M^{1.16}$$
$$C_{f\ fuselage} = \frac{0.454}{\left\{\left[\log_{10}\left(\mathrm{Re}_{fuselage}\right)\right]^{2.58}\left[1 + 0.144M^2\right]^{0.65}\right\}}$$

$$A\max = \frac{20.9}{294 / S_{ref}} - 3.83$$
$$d = \frac{5.5}{294 / S_{ref}}$$
$$f_{fuselage} = \frac{l}{0.3048d}$$

$$FF_{fuselage} = 1 + \frac{60}{f^3_{fuselage}} + \frac{f_{fuselage}}{400}$$
$$C_{d0_{fuselage}} = C_{f_{fuselage}} \frac{S_{wet_{fuselage}}}{S_{ref}} FF_{fuselage}$$

Input: File *equations9.isc*

```
program main

icount, ik, jj, kk as integer

Sref,Wo,l,k,Swet_fuselage,alt,mu,rho,R_fuselage,R_cutoff_sub,Cf_fuselage,Amax,d,f_fuselage,R_cutoff_sup as real
FF_fuselage,Cdo_fuselage,V,M as real

iunit, ifile, i, j as integer

blogic(3,2),brun as logical

lexist,lrun as logical

il as integer*2
j3,j4 as integer*2

rlong, rlonger as double
rnumber, rnumb, rexp as double


   # ModelEquations
   #function Cdo_sup = Cdo_Sup
   #%this code uses the component built method for supersonics guiven by Raymer

   Sref              =  325;
   V               =   856;
   blogic(2,2+2*jj) = 1
   ik = blogic(2,4+2*jj)
   Swet_fuselage   =  588;

   #fuselage

   Wo              =  24000;
   l               =  45*0.3048;          #%a * Wo^c * 0.3048
   k               =  0.052 * 10^(-5);     #%for smooth composite

   alt   =  1800;
   mu    =  0.00008;
   rho   =  1.25;
   M     =  0.85;

   R_fuselage      = rho * V * l / mu;

   R_cutoff_sub    = 38.21*(l/k)^1.053;                  #%eq 12.28 Raymer
   R_cutoff_sup    = 44.62*(l/k)^1.053*(M)^1.16;           #%eq 12.29 Raymer
   Cf_fuselage     = 0.454 / ((log10(R_fuselage))^2.58 * (1+0.144*(M)^2)^0.65);

   Amax          = 20.9/(294/Sref) - 3.83;
   d             = 5.5/(294/Sref);                 #%sqrt(4/pi*Amax);
   f_fuselage    = l/(0.3048*d);
   FF_fuselage   = (1 + 60/f_fuselage^3 + f_fuselage / 400) ;

   Cdo_fuselage    = Cf_fuselage * Swet_fuselage / Sref  *  FF_fuselage ;

end program
```

Output

```
FINAL VALUES OF VARIABLES
 ==========================
Logical Variables          4
 blogic(3;2)   = F F F T F F
 brun          = F
 lexist        = F
 lrun          = F
 Integer2 Variables          3
 il            =     0
 j3            =     0
 j4            =     0
 Integer Variables          8
 icount        =          0
 ik            =          1
 jj            =          0
 kk            =          0
 iunit         =          0
 ifile         =          0
 i             =          0
 j             =          0
 Real Variables         19
 Sref          =   325.0000
 Wo            =   24000.00
 l             =   13.71600
 k             =  5.2000001E-07
 Swet_fuselage =   588.0000
 alt           =   1800.000
 mu            =  7.9999998E-05
 rho           =   1.250000
 R_fuselage    =  1.8345150E+08
 R_cutoff_sub  =  2.4930214E+09
 Cf_fuselage   =  1.8315958E-03
 Amax          =   19.27374
 d             =   6.079932
 f_fuselage    =   7.401399
 R_cutoff_sup  =  2.4110405E+09
 FF_fuselage   =   1.166486
 Cdo_fuselage  =  3.8654767E-03
 V             =   856.0000
 M             =  0.8500000
 Double Variables          5
 rlong         =  0.000000000000000E+000
 rlonger       =  0.000000000000000E+000
 rnumber       =  0.000000000000000E+000
 rnumb         =  0.000000000000000E+000
 rexp          =  0.000000000000000E+000
```

Sample Problem 8.
This script was derived from an old FORTRAN code to calculate friction factor. The script uses expressions for Darcy-Weisbach, Colebrook, or Churchill equations depending on Reynolds number calculated from the input. The Colebrook equations require an iterative condition to obtain convergence based on a convergence criterion. The output was compared to that of the equivalent FORTRAN program.

Model

$$\mathrm{Re} = \frac{\rho U L}{\mu}$$

Laminar Flow:

$$f = \frac{64}{R_e}; \quad R_e \le 2100$$

Colbrook-White Equation:

$$\frac{1}{\sqrt{f}} = -2\log_{10}\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{R_e\sqrt{f}}\right); \quad R_e > 4000$$

Churchill Equation:

$$f = 8\left[\left(\frac{8}{R_e}\right)^{12} + \frac{1}{(A+B)^{3/2}}\right]^{1/12}$$

$$A = \left[2.457\log_e \frac{1}{(7/R_e)^{0.9} + 0.27\varepsilon/D}\right]^{16}$$

$$B = \left(\frac{37530}{R_e}\right)^{16}$$

Input:
File: *equations13.isc*
(The input file is not reproduced here but can be obtained directly from the /*SampleScript* folder of the iSCRIPT installation).

Output:
```
FINAL VALUES OF VARIABLES
 ========================
 Logical Variables         0
 Integer2 Variables         0
 Integer Variables        5
 niter              =        0
 iter              =        3
 ilam               =        2
 ierr              =        0
 nitre              =       200
 Real Variables        32
diam                 =  1.000000000000000E-002
 rey                =   659800.000000000
 fric               =  3.794308979507086E-002
 fric2               =  3.804064733943083E-002
 fric3               =  9.699909063352531E-005
 friction             =  3.794308979507086E-002
 rougha               =  1.000000000000000E-004
 eps                =  1.000000000000000E-004
 rselect              =  1.000000000000000E-004
 doveps               =   100.000000000000
 epsovd               =  1.000000000000000E-002
```

```
fac1               =   5.14000000000000
fac2               =   1285.22349993048
fac3               =   1.00723609551218
fac4               =  6.262561683476402E-003
fac5               =   5.13373743831652
fac6               =  0.194789860606491
fac7               =   112.286369433646
fac8               =   135.300985397874
fac9               =   15192.4564311220
deltaf             =  2.070842958548824E-011
depst              =  1.000000000000000E-004
g                  =  3.794308979509621E-002
r                  = -2.073378274625171E-011
small2             =  1.000000000000000E-009
forlog             =   10.0000000000000
fsuggest           =  3.794308979507086E-002
drdf               =   1.00122429181116
flowleft           =  4.000000000000000E-004
flowmid            =  2.000000000000000E-004
flowmid2           =  5.000000000000000E-005
enorm              =  2.978714945569056E-019
Double Variables          0
```

FORTRAN program

Results from FORTRAN execution of moodytest.for

```
 ==============================================
 friction=  3.794308979507086E-002
 fric=  3.794308979507086E-002
 fric2=  3.804064733943083E-002
 fric3=  9.699909063352531E-005
 ilam=           2
 rey=   659800.000000000
```

Sample Problem 9.

This script was used to illustrate the use of the natural recursive characteristics of iSCRIPT functions. The script is used to calculate the factorial of a number. The factorial of 4 was computed.

Model

$$n! = n(n-1)...2\cdot 1$$

$$0! = 1$$

Input:

File: *equations6kk.isc*

```
# Recursice function test using the factorial of a number
 program

      rnumber, rfactorial as integer
      # Initial Values

      rnumber = 4

      # Models
      rfactorial = FACTORIAL(rnumber)

 end program

# Recursive calculation of the factorial of a number
 Function FACTORIAL(n)
      n, rn as integer
      FACTORIAL as integer
      if n == 0
              rn = 1
      elseif n == 1
              rn = 1
      else
              rn = n * FACTORIAL(n-1)
      end if
      FACTORIAL = rn
      return
 End Function FACTORIAL
```

Output:

```
FINAL VALUES OF VARIABLES
 =========================
 Logical Variables        0
 Integer2 Variables       0
 Integer Variables        2
 rnumber          =       4
 rfactorial       =      24
 Real Variables        0
 Double Variables        0
```

Sample Problem 10.

This script was used to illustrate the use of the passing of literal valued arguments and expression arguments to functions. In addition, the syntax function calls within functions are illustrated. The model is the same as in Sample Problem 6.

<u>Input:</u>
<u>File</u>: *equations6m.isc*
(The input file is not reproduced here but can be obtained directly from the */SampleScript* folder of the iSCRIPT installation).

<u>Output:</u>
```
FINAL VALUES OF VARIABLES
 =========================
Logical Variables           0
 Integer2 Variables           0
 Integer Variables           0
 Real Variables           7
rho                   =   1.05000000000000
 v                    =   12.2000000000000
 l                    =   5.00000000000000
 mu                    =  8.500000000000001E-005
 re                   =   105.000000000000
 f                    =  4.932557310067764E-002
 i                    =  0.000000000000000E+000
 Double Variables           0
```

Sample Problem 12. MATLAB Compatibility

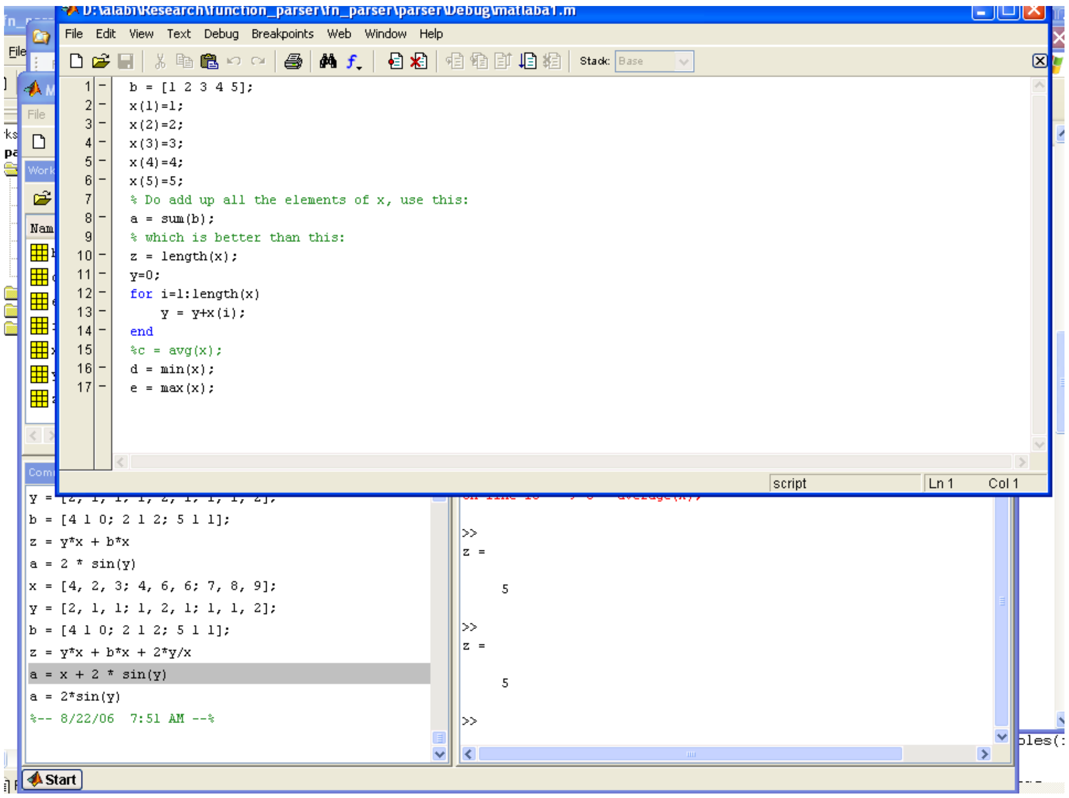
This script was used to illustrate compatibility with MATLAB matrix manipulation features. The equivalent MATLAB file is *matlaba1.m*.

Model	Input File: *matlaba1.isc*

$\mathbf{x} = [1 \quad 2 \quad 3 \quad 4 \quad 5]$
$\mathbf{b} = [1 \quad 2 \quad 3 \quad 4 \quad 5]$
$a = \sum_{i=1}^{5} b_i$
$z = size(\mathbf{x})$
$y = \sum_{i=1}^{5} x_i$
$d = \min(\mathbf{x})$
$e = \max(\mathbf{x})$
	i,y,x(5),b(5),z,a,c,d,e as integer
	b = [1 2 3 4 5];
	x(1)=1;
	x(2)=2;
	x(3)=3;
	x(4)=4;
	x(5)=5;
	% Do add up all the elements of x, use this:
	a = sum(b);
	% which is better than this:
	z = length(x);
	for i=1:length(x)
	    y = y+x(i);
	end
	%c = avg(x);
	d = min(x);
	e = max(x);

Output:

```
FINAL VALUES OF VARIABLES
 ========================
Logical Variables        0
 Integer2 Variables        0
 Integer Variables        9
 i                =        6
 y                =        15
 x(5)              =        1        2        3        4
        5
 b(5)              =        1        2        3        4
        5
 z                =        5
 a                =        15
 c                =        0
 d                =        1
 e                =        5
 Real Variables        0
 Double Variables        0
```


D:\alabi\Research\function_parser\fn_parser\parser\Debug\matlaba1.m

```
b = [1 2 3 4 5];
x(1)=1;
x(2)=2;
x(3)=3;
x(4)=4;
x(5)=5;
% Do add up all the elements of x, use this:
a = sum(b);
% which is better than this:
z = length(x);
y=0;
for i=1:length(x)
    y = y+x(i);
end
%c = avg(x);
d = min(x);
e = max(x);
```

script Ln 1 Col 1

```
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x
a = 2 * sin(y)
x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x + 2*y/x
a = x + 2 * sin(y)
a = 2*sin(y)
%-- 8/22/06  7:51 AM --%
```

```
>>
z =

     5

>>
z =

     5

>>
```

Sample Problem 13. MATLAB Compatibility

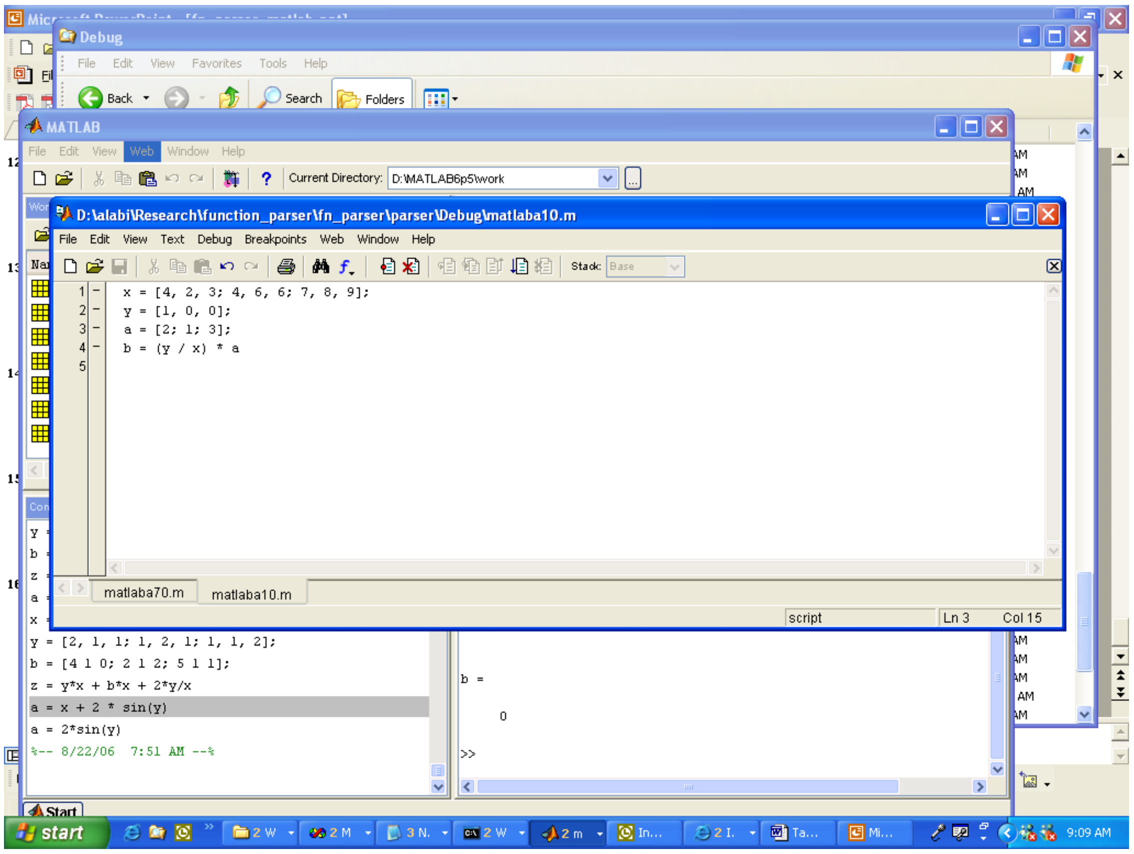
This script was used to illustrate compatibility with MATLAB matrix manipulation features – in particular the contraction of matrix order. The equivalent MATLAB file is *matlaba10.m*.

Model	Input:

$\mathbf{x} = \begin{bmatrix} 4 & 2 & 3 \\ 4 & 6 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{a} = \begin{bmatrix} 2 & 1 & 3 \end{bmatrix}$
$\mathbf{b} = (\mathbf{y} / \mathbf{x}) \times \mathbf{a}$
	File: *matlaba10.isc*
	x(3,3),y(3,1),a(1,3) as integer
	b as real
	x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
	y = [1, 0, 0];
	a = [2; 1; 3];
	b = (y / x) * a;

Output:

```
FINAL VALUES OF VARIABLES
 =========================
Logical Variables       0
 Integer2 Variables       0
 Integer Variables       3
 x(3;3)             =       4       2       3       4
        6       6       7       8       9
 y(3;1)             =       1       0       0
 a(1;3)             =       2       1       3
 Real Variables       1
 b                 =  0.0000000E+00
 Double Variables       0
```



D:\alabi\Research\function_parser\fn_parser\parser\Debug\matlaba10.m

Current Directory: D:\MATLAB6p5\work

```
x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
y = [1, 0, 0];
a = [2; 1; 3];
b = (y / x) * a
```

```
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x + 2*y/x
a = x + 2 * sin(y)
a = 2*sin(y)
%-- 8/22/06  7:51 AM --%
```

```
b =

     0

>>
```

Sample Problem 14. MATLAB Compatibility

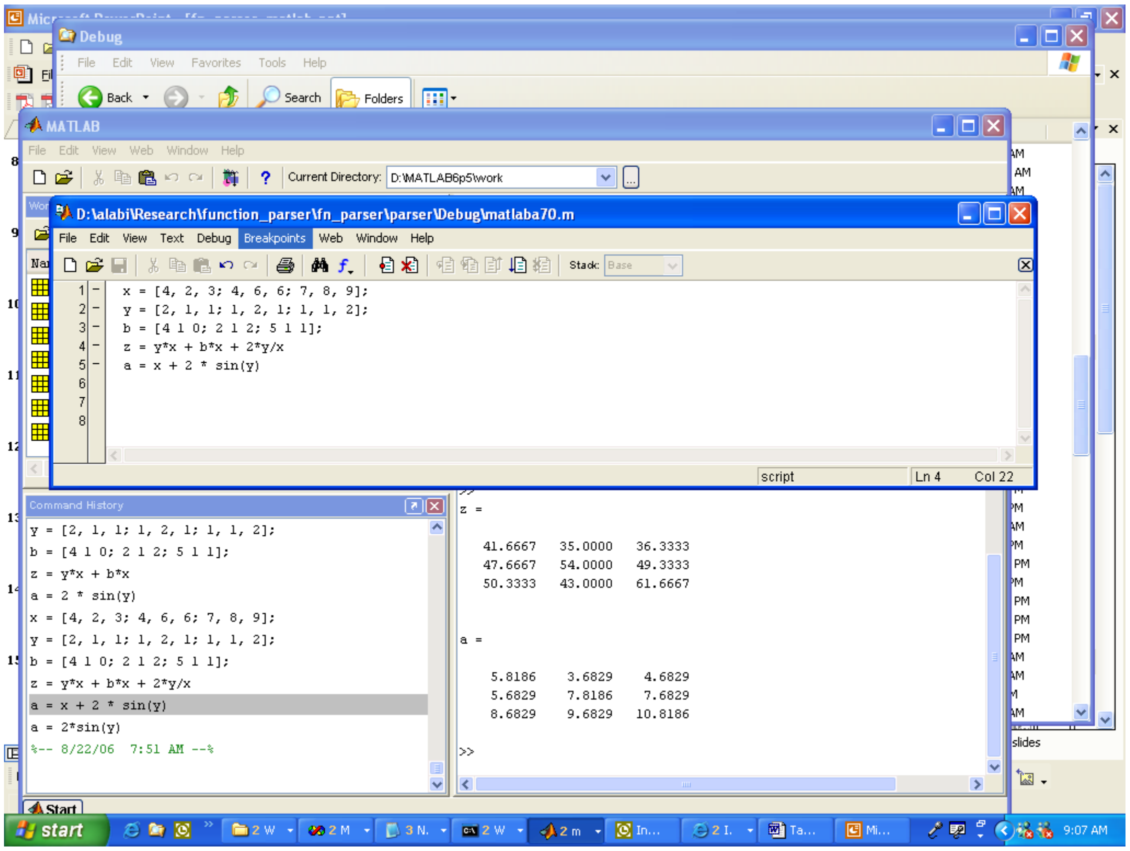
This script was used to illustrate compatibility with MATLAB matrix manipulation features. The equivalent MATLAB file is *matlaba70.m*.

Model	Input File: *matlaba70.isc*

$\mathbf{x}=\begin{bmatrix}4&2&3\\4&6&6\\7&8&9\end{bmatrix},\mathbf{y}=\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix},\mathbf{b}=\begin{bmatrix}4&1&0\\2&1&2\\5&1&1\end{bmatrix}$
 $\mathbf{z}=\mathbf{y}\times\mathbf{x}+\mathbf{b}\times\mathbf{x}+2\mathbf{y}/\mathbf{x}$
	`i,x(3,3),y(3,3),b(3,3) as integer`
	`z(3,3),a(3,3) as real`
	`x = [4, 2, 3; 4, 6, 6; 7, 8, 9];`
	`y = [2, 1, 1; 1, 2, 1; 1, 1, 2];`
	`b = [4 1 0; 2 1 2; 5 1 1];`
	`z = y*x + b*x + 2*y/x;`
	`a = x + 2 * sin(y);`

Output:

```
FINAL VALUES OF VARIABLES
 =========================
Logical Variables          0
 Integer2 Variables          0
 Integer Variables         4
 i                  =         0
 x(3;3)              =         4         2         3         4
         6         6         7         8         9
 y(3;3)              =         2         1         1         1
         2         1         1         1         2
 b(3;3)              =         4         1         0         2
         1         2         5         1         1
 Real Variables          2
 z(3;3)              =   41.66667       35.00000       36.33333
  47.66667       54.00000       49.33333       50.33333       43.00000
  61.66667
 a(3;3)              =   5.818595       3.682942       4.682942
  5.682942       7.818595       7.682942       8.682942       9.682942
  10.81859
 Double Variables          0
```


D:\alabi\Research\function_parser\fn_parser\parser\Debug\matlaba70.m

```
x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x + 2*y/x
a = x + 2 * sin(y)
```

Command History

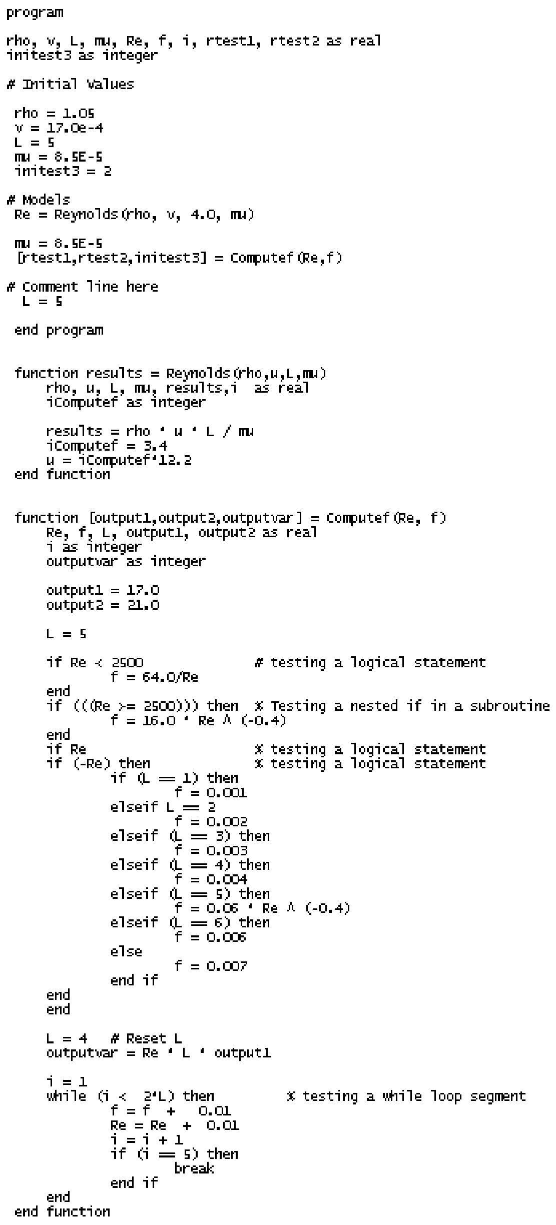
```
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x
a = 2 * sin(y)
x = [4, 2, 3; 4, 6, 6; 7, 8, 9];
y = [2, 1, 1; 1, 2, 1; 1, 1, 2];
b = [4 1 0; 2 1 2; 5 1 1];
z = y*x + b*x + 2*y/x
a = x + 2 * sin(y)
a = 2*sin(y)
%-- 8/22/06  7:51 AM --%
```

```
z =

   41.6667   35.0000   36.3333
   47.6667   54.0000   49.3333
   50.3333   43.0000   61.6667


a =

    5.8186    3.6829    4.6829
    5.6829    7.8186    7.6829
    8.6829    9.6829   10.8186

>>
```

Sample Problem 15. MATLAB Compatibility

This script was used to illustrate compatibility with MATLAB function features – including the management of function arguments and the use of multiple output arguments. The model is the similar to that in Sample Problem 6.

Input:
File: *matlaba14.isc*



```
program

rho, v, L, mu, Re, f, i, rtest1, rtest2 as real
initest3 as integer

# Initial Values

 rho = 1.05
 v = 17.0e-4
 L = 5
 mu = 8.5E-5
 initest3 = 2

# Models
 Re = Reynolds(rho, v, 4.0, mu)

 mu = 8.5E-5
 [rtest1,rtest2,initest3] = Computef(Re,f)

# Comment line here
  L = 5

 end program


 function results = Reynolds(rho,u,L,mu)
     rho, u, L, mu, results,i  as real
     iComputef as integer

     results = rho * u * L / mu
     iComputef = 3.4
     u = iComputef*12.2
 end function


 function [output1,output2,outputvar] = Computef(Re, f)
     Re, f, L, output1, output2 as real
     i as integer
     outputvar as integer

     output1 = 17.0
     output2 = 21.0

     L = 5

     if Re < 2500              # testing a logical statement
             f = 64.0/Re
     end
     if (((Re >= 2500))) then  % Testing a nested if in a subroutine
             f = 16.0 * Re ^ (-0.4)
     end
     if Re                     % testing a logical statement
     if (-Re) then             % testing a logical statement
             if (L == 1) then
                     f = 0.001
             elseif L == 2
                     f = 0.002
             elseif (L == 3) then
                     f = 0.003
             elseif (L == 4) then
                     f = 0.004
             elseif (L == 5) then
                     f = 0.06 * Re ^ (-0.4)
             elseif (L == 6) then
                     f = 0.006
             else
                     f = 0.007
             end if
     end
     end

     L = 4   # Reset L
     outputvar = Re * L * output1

     i = 1
     while (i <  2*L) then        % testing a while loop segment
             f = f  +   0.01
             Re = Re  +  0.01
             i = i + 1
             if (i == 5) then
                     break
             end if
     end
 end function
```

<u>Output:</u>

```
FINAL VALUES OF VARIABLES
 ==========================
 Logical Variables          0
 Integer2 Variables          0
 Integer Variables          1
 initest3               =       5712
 Real Variables          9
 rho                    =   1.050000
 v                     =  1.7000000E-03
 L                     =   5.000000
 mu                     =  8.5000000E-05
 Re                     =   84.00000
 f                     =  0.0000000E+00
 i                     =  0.0000000E+00
 rtest1                =   17.00000
 rtest2                =   84.00000
 Double Variables          0
```

Sample Problem 16. Multi-Source File projects

This script was used to illustrate the multi-source file project capabilities of iSCRIPT. The model is the similar to that in Sample Problem 6.The source files include:

Project1a.isc
Project1b. isc
Project1c. isc
Project1d. isc.

All source files were listed in a single project file: *project1.ipr*. The source files could also be individually entered at the command line.

Input:
File:*project1.ipr*

Output:
The output is similar to the Sample Problem 5.

Sample Problem 17. Integration with a MATLAB program,

This solution consists of the following files:

PS_conv.m	a MATLAB program that rates a low-bypass turbofan aircraft engine. The input to the program includes the altitude, Mach number etc. This program reads the input from a file PS_input.txt. The output from the program includes several variables including the thrust, fuel consumption etc. The output from the program is written to a file PS_output.txt.
ata.ipr	an iSCRIPT project file containing three files main.isc, ps_component.isc, ps_setting.isc. These files are described below.
Main.isc	a main program defining two components – PS_setting and PS_Component. PS_Setting simply sets the conditions for computing (rating) the aircraft engine.
PS_component.isc	an iSCRIPT component model file. This model file includes commands to execute the MATLAB model. An input file is created for the MATLAB program and the output from the MATLAB is read. The output is further used to compute certain quantities including the total exergy destruction in the engine.

Procedures for Running this Solution

1. Run the MATLAB program in MATLAB to check the model of the aircraft engine.
2. Compile the MATLAB program into an executable. This step includes issuing the command `mcc –m PS_conv`. An executable is generated names PS_conv.exe. This executable was renamed to PS.exe.
3. Run iSCRIPT. Enter the project file ata.ipr at the iSCript prompt
4. View the results.
5. Note that the model may be further optimized on the high-level using the optimization procedures present in the iSCRIPT program.

Sample Problem 18. Test of Input/Output and Directory Management Commands

This sample problem file illustrates the various input/output and directory manipulation commands. The file is *equations24.isc* and is listed below. The model and output is the same as that in Sample Problem 1.

```
    # Test of open/close read/write"
    alt,    T,    P,    mu,    rho,    alt_1, var2, var4 as real
    imyopen, imyoout, iout, irun as integer
    var3(3) as real

    imyopen = open('input.txt', 'rt+')
    call write(6,'File input file opened on unit: ', imyopen)
    imyoout = open('output.txt', 'wt+')
    call write(6,'File opened output file on unit: ', imyoout)

    call write(6) 'Press any key to continue'
    call read(6)

    call read(imyopen,'*%g6') alt_1
    var3 = fscanf(imyopen, *)

    write 'Enter a value for var4'
    call read (6) var4

    alt = alt_1 * 0.3048
    T   = (2.29013E-12*alt*alt*alt+8.60446E-08*alt*alt-6.82246E-
  03*alt+3.18150E+02)
    P   = 101325*(9.63714E-19*alt*alt*alt*alt - 1.18488E-13*alt*alt*alt)
    P   = P + 101325*(5.52991E-9*alt*alt- 1.18225E-4*alt + 1)
    mu  = (1.61988E-7*T*T*T-4.66143E-4*T*T+7.25242E-1*T+4.20454)*1e-7
    rho = P/287/T

    irun = execute('PS.exe')
    if (irun == -1) then
      call write(imyoout, *) 'Executable did not run or could not be located'
    end if

    call write(imyoout) 'My current directory is:', currentdirectory
    changedirectory('D:\alabi')
    call write(imyoout) 'My new directory is:', currentdirectory
    changedirectory('')
    call write(imyoout) 'My final directory is:', currentdirectory
    call write(imyoout,*)
    call fprintf(imyoout,'')
    call write(imyoout, 'rho = ', rho, ', T = ', T, ', P = ', P, ', mu = ', mu)
    call rewind(imyopen)
    call read(imyopen, *) var2
    call close(imyopen)
    call close(imyoout)
```

Instructions:

1. Run the above script.
2. Examine the output files and compare with the input I/O calls in the script as well as the output of Sample Problem 1.

Sample Problem 19. Executing an iSCRIPT program in Parallel

Any iSCRIPT program (that can run on a single processor computer) can execute in parallel in a multi-processor environment. However, this example illustrates the use of iSCRIPT in optimizing a problem is parallel.

This sample problem file contains a main program and a subroutine which evaluates a model. The model is the Rastrigin equation as shown below.

$$f(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2)$$

This function has several local minima making it difficult for a gradient-based procedure to capture the actual minimum without the benefit of a good starting or guess value. The actual minimum value is 0 and occurs at the values of (x_1, x_2) = (0,0).

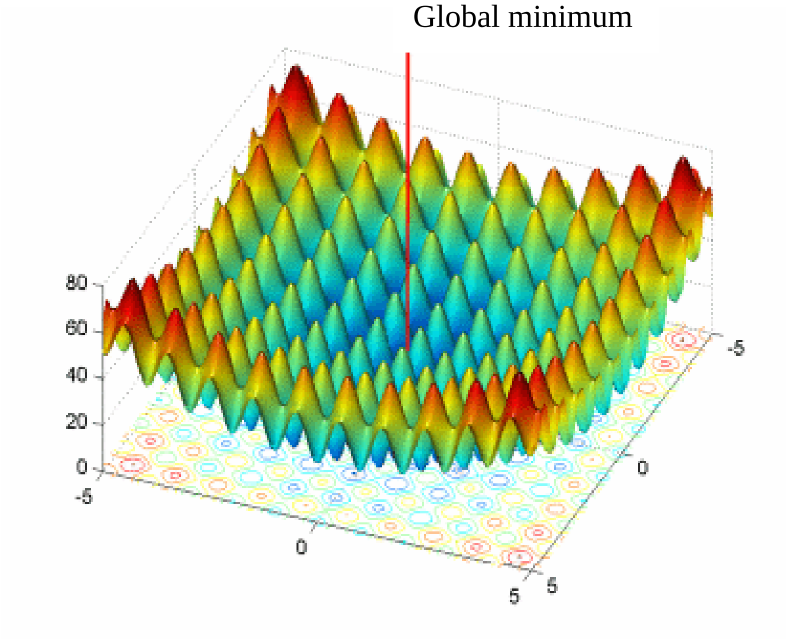


Figure P19.1. Plot of the Rastrigin function.

The file is *rastrigin.isc* and is listed below.

```
# This is the system program.
# This program defines all components, subsystems, and systems

program main

  #global pi as double
  #pi = 4*atan(1.0)
```

```
#.1 Create the components

#.2 Create the subsystems (One component, One subsystem, one system)

  # Create the entire system component and its variables
  CreateComponent(Rastrigin, Models_entire_system)
  CreateVariable(Rastrigin, y, double, 0,0, 1.0E14, 0.0, 0.0, $)
  CreateVariable(Rastrigin, x1, double, 0,0, 10.0, -10.0, 0.5, kg/s)
  CreateVariable(Rastrigin, x2, double, 0,0, 10.0, -10.0, 0.5, m2)

  AddObjective(Rastrigin, y, 0)

  AddVarObjective(Rastrigin, x1)
  AddVarObjective(Rastrigin, x2)

#.3 Evaluate the system at the initial conditions, then optimize
  #Rastrigin.Execute

  Global.maxPopulation = 4000
  Global.maxInitialEvaluations = 10000
  Global.maxGenerations = 200
  Global.optconvergencelimit =   1.0E-7
  Global.mutationfreq =   0.2
  Global.sampsizepervariable = 8000

  #Try
  Global.maxPopulation = 100
  Global.maxInitialEvaluations = 500
  Global.maxGenerations = 100
  Global.optconvergencelimit =   1.0E-7
  Global.mutationfreq =   0.2
  Global.sampsizepervariable = 200

  #Try
  Global.maxPopulation = 100
  Global.maxInitialEvaluations = 400
  Global.maxGenerations = 100
  Global.optconvergencelimit =   1.0E-7
  Global.mutationfreq =   0.2
  Global.sampsizepervariable = 100

  #Try
  Global.maxPopulation = 70
  Global.maxInitialEvaluations = 200
  Global.maxGenerations = 10
```

```
   Global.optconvergencelimit =   1.0E-7
   Global.mutationfreq =   0.2
   Global.sampsizepervariable = 90

   Rastrigin.Optimize

end program


subroutine Rastrigin ()
#This subroutine is an execute model for entire subsystem (and system)
  x1, x2, pi As Double
  x1 = Rastrigin.x1
  x2 = Rastrigin.x2
  pi = 4*atan(1.0)
  #pi = 3.1415926536
  Rastrigin.y = 20.0 + x1*x1 + x2*x2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
end subroutine
```

Output

The output file again is *outputscript.txt*. The correct results were obtained in about 2.9 seconds on a Pentium workstation using only a population of 70 realizations.
Details of the optimization process are recorded in the file *optimize.txt*. The details include the initial values of the optimization variables and the objective, the various realizations of the system being evaluated, the array of viable systems or realizations (population of individuals in genetic algorithm parlance) by generation or as the optimization progresses, and the final results.

References

[1]Munoz, J. R., "A Decomposition Strategy Based on Thermoeconomic Isolation Applied to the Optimal Synthesis/Design and Operation of an Advanced Fighter Aircraft System," M.S. Thesis, Advisor: M. R. von Spakovsky, Mechanical Engineering Dept., Virginia Polytechnic University, Blacksburg, VA, February 2002.

[2]Alabi, K., Ladeinde, F., von Spakovsky, M. R., Moorhouse, D., Camberos, J., "Assessing CFD Modeling of Entropy Generation for the Air Frame Subsystem in an Integrated Aircraft Design/Synthesis Procedure," AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, Jan 2006.

[3]Molyneaux, A., "A Practical Evolutionary Method for the Multi-Objective Optimization of Complex Energy Systems, including Drivetrains." M.Sc. Thesis, Ecole Polytechnique Federale De Lausanne, EPFL, 2002.

[4]Leyland G. B., "Multi-Objective Optimization Applied to Industrial Energy Problems," Ph.D. Thesis, Ecole Polytechnique Federale De Lausanne, EPFL, 2002.

[5]Stoecker W. F., "Design of Thermal Systems," Third Edition. McGraw Hill Book Company, 1989.