View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

NASA/TM—2014-216638

Toolbox for the Modeling and Analysis of Thermodynamic Systems (T-MATS) User's Guide

Jeffryes W. Chapman
Vantage Partners, LLC, Cleveland, Ohio

Thomas M. Lavelle
Glenn Research Center, Cleveland, Ohio

Ryan D. May
Vantage Partners, LLC, Cleveland, Ohio

Jonathan S. Litt and Ten-Huei Guo
Glenn Research Center, Cleveland, Ohio

January 2014

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

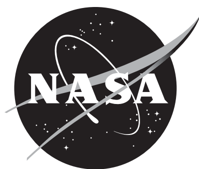
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*
- E-mail your question to *help@sti.nasa.gov*
- Fax your question to the NASA STI Information Desk at 443–757–5803
- Phone the NASA STI Information Desk at 443–757–5802
- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/TM—2014-216638



Toolbox for the Modeling and Analysis of Thermodynamic Systems (T-MATS) User's Guide

Jeffryes W. Chapman
Vantage Partners, LLC, Cleveland, Ohio

Thomas M. Lavelle
Glenn Research Center, Cleveland, Ohio

Ryan D. May
Vantage Partners, LLC, Cleveland, Ohio

Jonathan S. Litt and Ten-Huei Guo
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

January 2014

Acknowledgments

The authors would like to thank Alicia Zinnecker and Yuan Liu of N & R Engineering, and Jeffrey Csank of NASA Glenn Research Center for their work on T MATS testing and for general technical advice. We would also like to thank Sanjay Garg at NASA Glenn Research Center for his support in the development of T MATS, and the NASA Aviation Safety Program's Vehicle Systems Safety Technologies (VSST) project for funding this work.

Trade names and trademarks are used in this report for identification only. Their usage does not constitute an official endorsement, either expressed or implied, by the National Aeronautics and Space Administration.

Level of Review: This material has been technically reviewed by technical management.

Available from

NASA Center for Aerospace Information
7115 Standard Drive
Hanover, MD 21076–1320

National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Available electronically at http://www.sti.nasa.gov

Table of Contents

1.0 Introduction ... 1
1.1 Identification ... 1
1.2 Notation ... 2
1.3 System Overview ... 2
1.4 T-MATS Quick Start Guide ... 2
1.4.1 Installing T-MATS ... 2
1.4.2 Un-installing T-MATS ... 3
1.4.3 T-MATS examples ... 4
1.4.4 Block help ... 4
2.0 T-MATS Library Structure ... 5
3.0 T-MATS Tools ... 6
3.1 GF_Convert ... 6
3.2 iDesign_On ... 6
3.3 iDesign_Off ... 6
3.4 Block Link Setup ... 6
4.0 T-MATS Simulation Creation ... 7
4.1 Simulation Setup ... 7
4.2 T-MATS Formatting ... 9
4.2.1 Color coding ... 9
4.2.2 Simulink wiring ... 9
4.2.3 Mask format ... 10
4.2.4 S-functions ... 12
4.3 Solver Setup ... 12
4.4 Turbomachinery Plant Setup ... 12
4.4.1 Independent and dependent variables ... 13
4.4.2 Maps ... 13
4.4.3 Compressor variable geometries (IGVs and VSVs) ... 17
4.4.4 Bleeds and air bypass ... 17
4.4.5 Turbofan modeling options ... 18
4.4.6 iDesign ... 19
5.0 Customization ... 20
5.1 Library Blocks ... 20
5.1.1 Creating new Simulink Library Browser folders ... 20
5.2 S-functions ... 21
5.3 Tool Creation ... 21
6.0 Troubleshooting T-MATS ... 22
6.1 Convergence “Errors” ... 22
6.2 Crashing ... 23
7.0 T-MATS Tutorials ... 24
7.1 T-MATS Example 1: Newton-Raphson Equation Solver ... 24
7.2 T-MATS Example 2: Example_GasTurbine_SS ... 27
7.2.1 Creating the gas turbine plant ... 27
7.2.2 Plant solver integration and plant inputs ... 30
7.3 T-MATS Example 3: Example_GasTurbine_Dyn ... 32
7.3.1 General architecture ... 32
7.3.2 Adding control to the simulation ... 36
7.3.3 Plant interface with outer loop systems ... 38
7.3.4 Advanced simulation structure and formatting ... 38
Appendix A ... 40

Figure Index

Figure 1.—Sample simulation architecture for a dynamic system. 8
Figure 2.—Sample T-MATS function block parameters mask. 10
Figure 3.—Sample T-MATS function block parameters mask with vectored inputs and checkboxes. 11
Figure 4.—Sample turbojet block diagram. 13
Figure 5.—Sample compressor map (pressure ratio vs. corrected flow). 14
Figure 6.—Sample compressor map (efficiency vs. corrected flow). 15
Figure 7.—Sample turbine map (corrected flow vs. pressure ratio). 16
Figure 8.—Sample turbine map (efficiency vs. pressure ratio). 16
Figure 9.—Sample fan model creation schemes for a gas turbine with bypass. 19
Figure 10.—Sample MATLAB internal problem message. 23
Figure 11.—Model configuration parameters setup. 25
Figure 12.—Run equation solver. 26
Figure 13.—Gas turbine plant. 29
Figure 14.—Steady-state solver setup. 31
Figure 15.—`Iterative NR Solver w JacobianCalc` wiring. 33
Figure 16.—Dynamic gas turbine simple “outer” loop. 35
Figure 17.—Dynamic gas turbine with controller. 37

Table Index

Table 1.—T-MATS user’s manual notation 2
Table 2.—T-MATS color coding 9
Table 3.—Dynamic turbofan model independent and dependent variables 13
Table 4.—Compressor map equations 15
Table 5.—Compressor map equations 17
Table 6.—T-MATS library file structure 20
Table 7.—Steady-state turbofan engine example, unused outputs 27
Table 8.—Example 2 convergence outputs 30

1.0 Introduction

The Toolbox for the Modeling and Analysis of Thermodynamic Systems (T-MATS) is a Simulink toolbox intended for use in the modeling and simulation of thermodynamic systems and their controls. The package contains generic thermodynamic and controls components that may be combined with a variable input iterative solver and optimization algorithm to create complex systems to meet the needs of a developer.

The goal of the T-MATS software is to provide a toolbox for the development of thermodynamic system models, which contains a simulation framework, multi-loop solver techniques, and modular thermodynamic simulation blocks. While much of the capability in T-MATS is in transient thermodynamic simulation, the developer's main interests are in aero-thermal applications; as such, one highlight of the T-MATS software package is the turbomachinery block set. This set of Simulink blocks gives a developer the tools required to create virtually any steady-state or dynamic turbomachinery simulation, e.g., a gas turbine simulation. In systems where the control or other related systems are modeled in MATLAB/Simulink, the T-MATS developer has the ability to create the complete system in a single tool.

T-MATS turbomachinery blocks were created using a philosophy that combines the understandability and logic of physics-based models with the accuracy and tunability of empirically-developed models. This user's guide is written with the assumption that the user is familiar with modeling thermodynamic systems, and a user is encouraged to see the referenced documentation[1,2] for a review of general thermodynamic and modeling principles before building a simulation.

T-MATS is written in MATLAB/Simulink v2012b (The Mathworks, Inc.), is open source, and is intended for use by industry, government, and academia. All T-MATS equations were developed from public sources and all default maps and constants provided in the T-MATS software package are nonproprietary and available to the public.[3] The software is released under the Apache V2.0 license agreement. License terms and conditions can be found at http://www.apache.org/licenses/LICENSE-2.0.

1.1 Identification

The T-MATS user's guide applies to version 1.0 of the T-MATS software package, as developed by NASA Glenn Research Center. The package consists of a T-MATS Simulink library containing the following sub-libraries: Effectors and Controls, Numerical Methods, Solver, and Turbomachinery. In addition, the T-MATS package contains development tools, "T-MATS tools," as well as examples to demonstrate how systems may be created using the Simulink libraries.

[1] http://www2.chem.umd.edu/thermobook/: Referenced as of 9/2013
[2] Jones, S.M., "Steady-State Modeling of Gas Turbine Engines using the Numerical Propulsion System Simulation Code," GT2010–22350, ASME Turbo Expo, Glasgow, UK, June 14–18, 2010.
[3] Jones, S.M., "An Introduction to Thermodynamic Performance Analysis of Aircraft Gas Turbine Engine Cycles Using the Numerical Propulsion System Simulation," NASA/TM—2007-214690, March 2007.

1.2 Notation

The notation summarized in Table 1 will be used throughout the T-MATS User's Guide.

TABLE 1.—T-MATS USER'S MANUAL NOTATION

Type	Definition
`Courier Font`	MATLAB function, Simulink blocks, or code (i.e., Simulink: `From` block)
Times New Roman Font Italics	Variables (i.e., $f(x)$) or file names, block input/output
Arial Font	File Paths (i.e., TMATS_Library\MEX)
Platform: **`Name`** block	Signify what library a block is taken from (i.e., Simulink: `From` block)

1.3 System Overview

The Toolbox for the Modeling and Analysis of Thermodynamic Systems (T-MATS) is a generic thermodynamic simulation system built in MATLAB/Simulink. Developed to use graphical-based simulation techniques to meet the requirements of industry professionals as well as academics, T-MATS combines generic thermodynamic and controls modeling capability with a numerical iterative solver to create a framework for the creation of complex thermodynamic system simulations. Numerical methods utilized by T-MATS include Newton-Raphson iterative solving techniques and Jacobian calculation techniques.

1.4 T-MATS Quick Start Guide

1.4.1 Installing T-MATS

To install T-MATS execute the following steps:

1. Download T-MATS from the GIT server https://github.com/nasa/T-MATS, click the "Download ZIP" button, and extract the files to a folder that can be accessed by MATLAB, ensuring there are no spaces in the path name. Note that T-MATS was developed in the Microsoft Windows 7 operating system using MATLAB v2012b and Microsoft Software Development Kit (SDK) 7.1; it is not guaranteed to work on other operating systems, with other MATLAB versions, or with other compilers.
2. Navigate to the directory of the desired released version of T-MATS in the downloaded zip file, TMATS_vXX, in MATLAB.
3. Run *Install_TMATS.m*; this will set up the paths for T-MATS as well as create the mex files. If the user does not have elevated privileges, paths may not have been saved properly. If the paths have not been saved, new paths must manually be added to the *pathdef.m* file. Open the "Set Path" window by clicking on the "Set Path" button in the MATLAB toolbar on the HOME tab and add the paths (\TMATS_Library, \TMATS_Library\MEX, \TMATS_Library\TMATS_Support, and \TMATS_Tools) to the list. Click "Save" to save the path definitions; a message will pop up warning that administrator privileges are required and providing the option to instead save the paths to a new *pathdef.m* file. Save this file to a location on the MATLAB path. Note: this process will save the current path list; before saving, it should be verified that no unintended paths are on the list.

4. To add the T-MATS tools to the Simulink toolbar, create the MATLAB file, *startup.m*, with the following lines of code:

```
load_simulink
Menu_customization_TMATS
```

The *startup.m* file may be saved anywhere on the MATLAB path, and these lines function to load T-MATS editing tools to the Simulink window each time MATLAB is opened. If a *startup.m* file already exists, just add the above lines of code.
If, during MATLAB startup, a warning is displayed notifying the user that *Menu_customization_TMATS.m* cannot be found, it is probable that the T-MATS paths have not been added properly in MATLAB. The most common reasons for missing path definitions are that either the T-MATS paths were not loaded into the *pathdef* file (see step 3 to solve this issue) or the modified *startup.m* file is being used by a version of MATLAB that T-MATS was not installed on. The warning messages can be removed by either installing T-MATS on the other version or by placing the *startup.m* file in a path used only by the desired MATLAB version. It should be noted that, while these warnings may be annoying, they will have no effect on the performance or functionality of MATLAB if T-MATS is not used.

At this point the T-MATS library installation is complete. T-MATS modules may be found in the Simulink Library Browser under the heading "TMATS." T-MATS tools may be found under "Diagram" in the Simulink model window menu bar.

1.4.2 Un-installing T-MATS

To remove T-MATS, execute the following steps:

1. Execute *Uninstall_TMATS.m* from the same location that *Install_TMATS* was run. (By default, these files are both located in the TMATS folder).
    a. If paths were added manually to the *pathdef.m* file during installation, they must be removed manually by re-saving the *pathdef.m* file after the *Uninstall_TMATS.m* file has been run.
2. Remove the following lines from the *startup.m* file:

```
load_simulink
Menu_customization_TMATS
```

1.4.3 T-MATS examples

T-MATS includes a number of sample systems set up to help a user gain insight into how to use the T-MATS library to simulate thermodynamic systems. To gain access to these examples, navigate to the Examples folder under TMATS_Examples and run *TMATS_Run_Example.m*. A selection of different examples will be listed in the MATLAB window (as shown below); the user can then indicate the number corresponding to the example to open. An alternate way to open an example is to navigate directly to the folder containing the example and run the *_setup_everything.m* file (or by opening the *.slx* file, in cases where a setup file does not exist).

```
>> run ('C:\Builds\TMATS_C\TMATS_Examples\TMATS_Run_Example.m')
     Select the enumeration of the T-MATS Example to be set up:
     1) NewtonRaphson_Equation_Solver
     2) Steady State GasTurbine
     3) Dynamic GasTurbine
     4) Cancel Setup
```

The *_setup_everything.m* script will execute every command necessary to set up that particular example, which may include any of the following: loading variables to the workspace, creating paths, loading busses, and opening the Simulink model. The paths can be removed by running the corresponding *_Example_cleanup.m* file.

1.4.4 Block help

Each T-MATS module contains a help file that explains the particulars of how to set up that block, therefore the low level functionality of each block will not be discussed in this document.

2.0 T-MATS Library Structure

The T-MATS Library, as displayed in the Simulink Library Browser, contains four sub-libraries: Effectors and Controls, Numerical Methods, Solver, and Turbomachinery.

1. Effectors and Controls: This sub-library contains the building blocks of the outermost loop of a simulation. The library blocks are simple representations of effector and control system components. Depending on the fidelity requirements of the simulation, it may be necessary to develop more complex models.
2. Numerical Methods: This sub-library contains the most basic building blocks of the numerical solvers required to simulate a T-MATS system. These blocks are very generic and can be used to create iterative solvers for a wide range of systems, even outside of T-MATS.
3. Solver: This sub-library contains blocks that simplify simulation creation, including hybrid blocks, comprised of blocks from the "Numerical Methods" sub-library, and blocks that are useful for general simulation development.
4. Turbomachinery: This sub-library contains the building blocks necessary to create a gas turbine model for simulation. The blocks are based on basic thermodynamic equations and principles, and use a series of maps that define the characteristics of the system being modeled. The default maps and constants in T-MATS are intended only for demonstration.

T-MATS is intended to be an environment for building thermodynamic simulations in Simulink with the goal of dynamic analysis and controls design. The sub-libraries discussed above provide the basic building block of a simulation, however a user may be required to create new blocks, and/or tweak existing blocks, to meet his/her specific needs.

3.0 T-MATS Tools

T-MATS comes with a set of tools designed to make model creation easier and faster. These tools are located on the Simulink menu bar under Diagram -> T-MATS Tools. Each tool will be described in the following sections.

3.1 GF_Convert

The GF_Convert tool changes the selected Simulink: `GoTo` block into a Simulink: `From` block, or vice versa. It will also convert a Simulink: `Outport` into a Simulink: `Inport` and vice versa.

3.2 iDesign_On

The iDesign_On tool will check all iDesign checkboxes located in T-MATS blocks within the model. The iDesign mode is described in Section 4.4.6.

3.3 iDesign_Off

The iDesign_Off tool will un-check all iDesign checkboxes located in T-MATS blocks within the model. The iDesign mode is described in Section 4.4.6.

3.4 Block Link Setup

Block Link Setup will create Simulink: `GoTo` blocks and/or Simulink: `From` blocks that correspond to a particular subsystem's inputs and outputs, and then connect them with wires. T-MATS blocks will be dealt with on a case-by-case basis, with blocks and connections created as appropriate for that particular block. In many cases, additional Simulink: `GoTo` or Simulink: `From` blocks will be created that are intended to be used in other parts of the simulation. The Block Link Setup tool may also be used to create a Simulink: `Inport` or Simulink: `Output` block for a selected Simulink: `GoTo` or Simulink: `From` block, or vice versa.

4.0 T-MATS Simulation Creation

4.1 Simulation Setup

In many thermodynamic systems, the inputs to each component alone are not enough to determine how the system will respond. For example, the mass flow rate and pressure ratio across a compressor are not unique for a given shaft speed. In order to resolve this, a compressor map is used to solve for incoming flow to the compressor block. This calculated compressor flow and the actual incoming flow are then compared to calculate a flow error. During simulation, the flow error is driven to zero by adjusting the R-line value to change the position on the map at which the compressor is operating (see Section 4.4 for more details).

The T-MATS software package contains blocks that allow for easy creation of a thermodynamic model that requires "outer loop" iteration (over time) and "inner loop" iteration. The iterative solver blocks in the "Solver" sub-library apply a Newton-Raphson method to iteratively solve for plant independent variables by monitoring plant dependent variables. For the example in the previous paragraph, the dependent variable is flow error and the independent variable is R-line. As with all Newton-Raphson techniques, a Jacobian for the plant is calculated by the solver; this Jacobian is essentially a linear map from the plant independent variables to the dependent variables, which may change based on the system operating point and the degree of system nonlinearity. In cases where the Newton-Raphson solver fails to converge in a timely manner due to a non-linearity, the T-MATS solver blocks will add robustness by recalculating the Jacobian matrix.

An easy way to learn how to use the T-MATS "Solver" sub-library is to look at the examples. There are three examples included in the T-MATS software package that demonstrate progressively more complicated solver implementations; for more information on the examples and setup, see the tutorial (Section 7.0). In the first and simplest example, *NewtonRaphson_Equation_Solver*, the TMATS: `SS NR Solver w JacobianCalc` block is used to solve a system of three equations for three unknowns. It should be noted that the independent variables (x) and the dependent variables ($f(x)$) are both expressed in the form of a 3x1 vector. The input to the iterative solver block can be of any length, as long as the inputs, outputs, and appropriate mask input variables are all the same length.

The second example (*GasTurbine_SS_Template*) sets up a gas turbine with a steady-state solver using the T-MATS block set. In this example, the TMATS: `SS NR Solver w JacobianCalc` block is used as the system solver. The four inputs, or dependent variables (on the right hand side of the solver block), are the normalized flow errors from the nozzle, turbine, and compressor, as well as shaft acceleration. The independent variables are flow into the engine, R-line (compressor), PR-map (turbine), and mechanical shaft speed. Components in the "Turbomachinery" sub-library have been color-coded for convenience (more details on the color scheme are in Section 4.2.1).

The third example (*GasTurbine_Dyn_Template*) simulates a dynamic gas turbine system. This system requires setting up a hierarchy with the TMATS: `Iterative NR Solver w JacobianCalc` block, as shown in Figure 1. The solver block and the plant (shown here as the IterativeSolver and InnerLoopPlant blocks) must be placed in a Simulink: `While Iterator subsystem` block. During operation, this setup will iteratively solve for convergence at each time step, "pausing" time to converge

the system (if convergence has been lost), then begin evolving through time. To allow for dynamics, the Newton-Raphon solver does not solve for mechanical shaft speed as in the steady-state solver example; instead *Ndot* is integrated outside of the solver loop to calculate the shaft speed. In addition to the while-loop hierarchy, the dynamic gas turbine example also outlines how busses and model calls may be incorporated in the simulation structure. Most of this added structural complexity is not required and has been included for demonstrative purposes only. What is required is that the plant and the TMATS: `Iterative NR Solver w JacobianCalc` block are in one subsystem, with a Simulink: `While Iteration` block, and that all outside effectors are outside of that subsystem, as shown in Figure 1.

For more information on the example and setup, see the tutorial (Section 7.0).

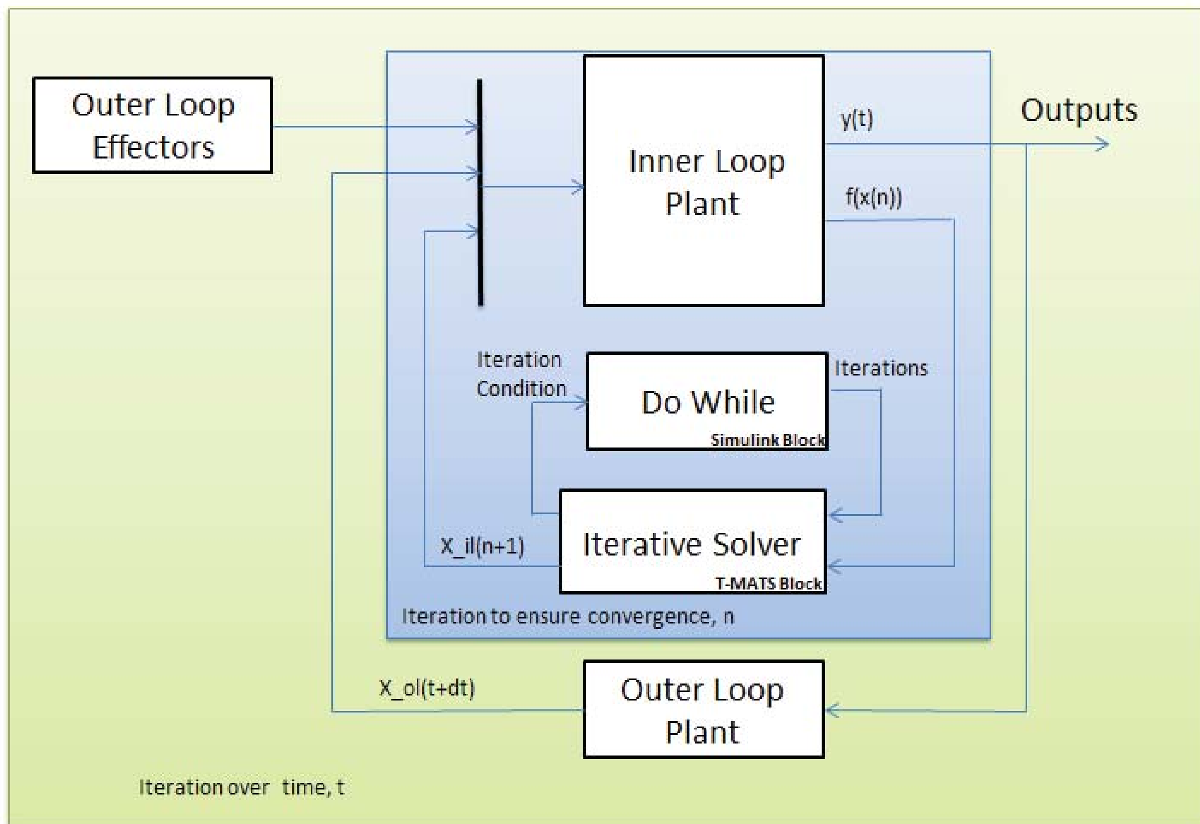


Figure 1.—Sample simulation architecture for a dynamic system.

4.2 T-MATS Formatting

4.2.1 Color coding

The inputs and outputs of T-MATS blocks are color coded to aid in simulation setup:

TABLE 2.—T-MATS COLOR CODING

Color	Description
Blue	Gas path information containing flow, enthalpy, total temperature, total pressure, and flow composition (typically fuel to air ratio (*FAR*))
Dark Green	Independent variable
Green	Dependent variable
Red	Independent variable that is obtained through integration in a dynamic simulation, but may be solved for by the solver in a steady-state simulation (e.g., shaft speed)
Magenta	Dependent variable that is integrated in a dynamic simulation, but may be used as a dependent variable in the iterative solver in the steady-state simulation (e.g., *Ndot*)
Pink	Contributor to calculation of a dependent variable (e.g., torque)
Yellow	Effector command (example fuel flow), not typically used by the solver

Typically, items of one color connect to items of the same color (red and magenta items may be exceptions, as noted in Table 2).

4.2.2 Simulink wiring

T-MATS uses a vectored input/output scheme to limit inputs and outputs to those that are absolutely necessary. This means that many of the wires represent arrays of numbers that must be delivered correctly for the simulation to work properly. The order of elements in these wires is given in the “help” page for each block; alternatively, the wire definitions may be viewed by looking under the mask or by running the wire into a bus selector. As a special case, the bleed outputs from the compressor and the cooling flow inputs to the turbine are both dynamically-sized vectors that reflect the number of designed bleeds (specified in the compressor or turbine mask). In general, if an output is not expected to be wired directly to another block, it will be placed in the **_Data* bus (e.g., stall margin is located in the *C_Data* output of the compressor block).

4.2.3 Mask format

T-MATS makes extensive use of the subsystem mask tools in Simulink. Each T-MATS block comes with a mask that has been created to allow the user to provide to the block any required parameter or setting without having to enter the block itself. The mask parameters for each T-MATS block may be viewed by double-clicking on that block; an example of a T-MATS block mask can be seen in Figure 2.

At the top of the popup will be the name of the library block followed by a block description. The variables that may be changed in the mask will be listed in the parameters section, which may have multiple tabs. Each mask parameter input description is formatted as: "Name_M – Definition [units, if applicable] (matrix size, if applicable) (equation format)," where the suffix "_M" denotes a variable defined in the mask. "Equation format" is typically used for table formatting, described later in this section. Check boxes are used in cases where features can be enabled or disabled; variable names for these items have the form "(VariableName)En_M". At the bottom of the parameter dialog is a "Help" button that will provide more information about the block when clicked.

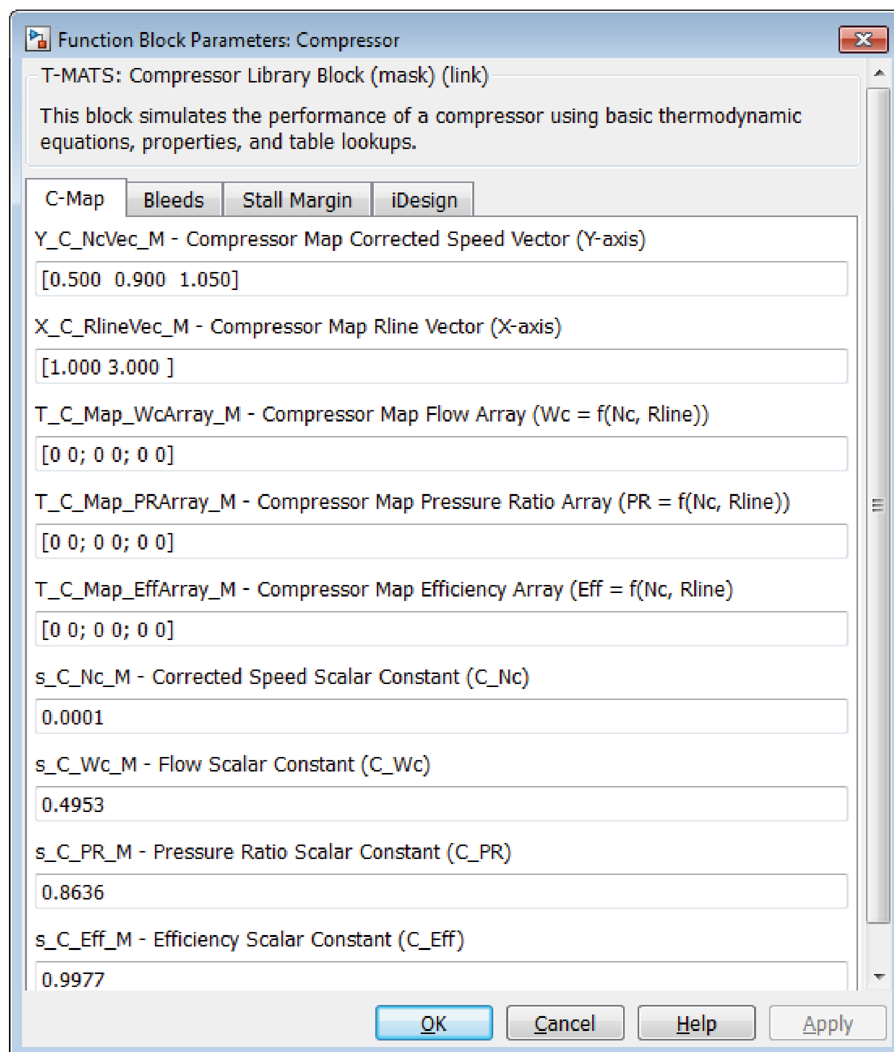


Figure 2.—Sample T-MATS function block parameters mask.

Many T-MATS blocks accept vector inputs, the length of which may determine the number of block inputs, dependent variables, independent variables, or block outputs. If an input description contains the symbol (“(size)x1”), then that input needs to be a vector of length “size” as dictated by the design. In a specific module, the value of “size” needs to be the same for each item. For example, in the compressor block the dimensions of the input for customer bleed are *cbx1* and for fractional bleed are *bx1*, so the length of the customer bleed vectors should all be the same (*cb*) and the length of fractional bleed vectors should be the same (*b*); it is not required that $cb = b$, as demonstrated in Figure 3 where $cb = 3$ and $b = 2$.

Maps and tables in T-MATS are entered as mask variables, with two inputs required for 1-D tables (axis and value breakpoints) and three inputs required for 2-D tables (breakpoints for two axes and for values). The axis breakpoint arrays will be indicated by the prefix “X_” or “Y_” while prefix “T_” indicates the array of table values. Scalars, which change the scale of maps and tables, are prefixed with “s_.” The input descriptions of all tables include a simple equation, surrounded by “(),” that denotes which axis variables are used for that particular table (e.g., in the compressor, “Wc = f (Nc,R-line)” denotes that an element of table *Wc* can be expressed as a function of axis variables *Nc* and *R-line*, which are defined right above it). Errors will be encountered if single element array (1x1) is entered for tables; instead a line or plane of the same number, for 1-D or 2-D tables respectively, should be entered to create a table with the same value. For example, a 2-D table with *x*-axis = [0 1], *y*-axis = [0 1], and table values = [5 5; 5 5], creates a plane of value 5.

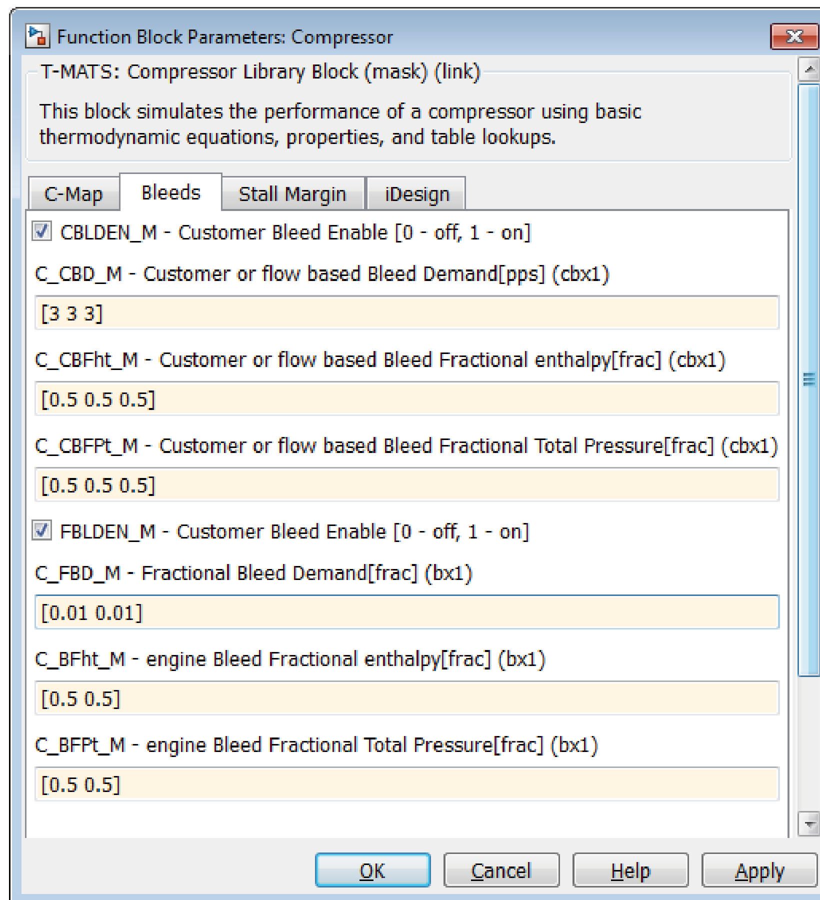


Figure 3.—Sample T-MATS function block parameters mask with vectored inputs and checkboxes.

4.2.4 S-functions

Many T-MATS function blocks make use of S-functions, which allow code to be developed in the C programming language and then be used by Simulink. (See Section 5.2 for more details about S-functions operation and Appendix A for a simple code example.) It is important to have a basic understanding of S-functions so the code may be tuned for each user's need. The C code is compiled for use in Simulink via the MATLAB *mex* command.

4.3 Solver Setup

The T-MATS solver block sets are based around two major components: an iterative solver and a Jacobian calculator. The iterative solver makes use of the Newton-Raphson method to step a plant toward solution and is described mathematically in Equation (1):

$$x(n+1) = x(n) - \frac{f\big(x(n)\big)}{f'\big(x(n)\big)} \qquad \text{where,} \qquad f'(x(n)) = Jacobian \quad (1)$$

The Jacobian is a linear map between plant inputs and outputs. It is defined by perturbing each plant input from an initial condition (*x*) to find the effect on the plant outputs (*f*). A more precise mathematical description of the Jacobian can be seen in Equation (2).

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \quad (2)$$

T-MATS solver blocks work in two major steps. In the first step, the Jacobian calculator calculates a linear map of the plant. Starting at the initial condition, the Jacobian calculator takes turns perturbing each input slightly, recording the results, and calculating the slope of each output as a function of each input. Once this is completed for each input, the Jacobian is built, inverted, and sent to the Newton-Raphson block. In the second step, the Newton-Raphson solver steps toward a solution using the Jacobian developed in the first step. This solver method makes the assumption that the plant is locally linear, so it is possible that the solver will not converge on a solution when the system is (highly) nonlinear. The help files for the blocks inside the solver (TMATS: `Newton Raphson Solver` and TMATS: `Jacobian Calculator`) include more information about the solver and how it works.

4.4 Turbomachinery Plant Setup

The "Turbomachinery" sub-library of the T-MATS Simulink library contains the generic blocks that may be combined to create gas turbines. The most basic definition of a gas turbine is a compressor in series with a burner, a turbine, and a shaft to connect them; more complex gas turbines may have multiple sets of compressors and turbines, as well as different air bypass setups. A simple example of a working gas turbine, used throughout this document, is the turbojet, which consists of a compressor, a burner, a turbine, and a nozzle in series (as shown in Figure 4). Descriptions of the plant setup in this section assumes the reader has some understanding of turbomachinery; if this is not the case, the reader should refer to the sources listed in Section 1.0.

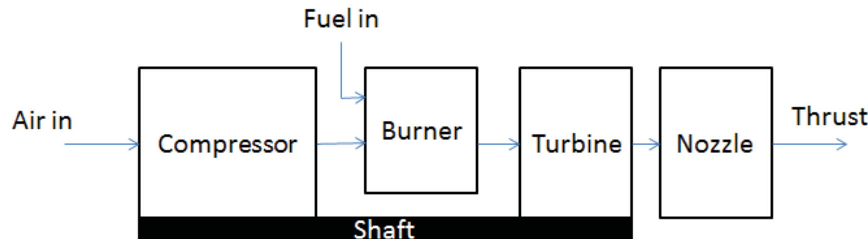


Figure 4.—Sample turbojet block diagram.

4.4.1 Independent and dependent variables

In a turbomachinery system, flow from station to station is typically unknown and must be solved for. T-MATS solver blocks can be used to drive the flow errors (the difference between input flow and internally calculated flow) to zero by changing the location of the operating point on compressor and turbine maps. When setting up the simulation, the number of independent variables must match the number of dependent variables. Variable definitions for the dynamic gas turbine example (a simple turbojet) can be seen in Table 3.

TABLE 3.—DYNAMIC TURBOFAN MODEL INDEPENDENT AND DEPENDENT VARIABLES

Dependent variables	Independent variables
Nozzle flow error	Flow into engine
Turbine flow error	Turbine PR-map
Compressor flow error	Compressor R-line

(Note: See the color coding Section, 4.2.1, for an easy way to find typical independent and dependent variables.)

It should be noted that, although certain inputs may be known in some simulations, setting the known independent variables to constants, and removing dependent variables from the solver, should not be done because minor changes in the vector of independent variables may drive convergence to a map point that does not exist, which will result in the simulation crashing. It is more prudent to simply use the known value(s) in the initial conditions of the independent variables and allow the solver to use the dependent variables to solve for it. If the initial conditions are indeed the steady-state solutions, the final value of the independent variables will remain close to, or the same as, the initial values. If precise input/output interdependencies are required, they can be generated using the TMATS: `Jacobian Calculator` block in the “Numerical Methods” sub-library.

4.4.2 Maps

The TMATS: `Compressor` block uses maps to compute efficiency, pressure ratio, and corrected flow for the compressor, each as a function of corrected compressor speed (*Nc*) and R-line (typically solved for by the iterative solver) as shown in Figure 5 and Figure 6. Because of the complication of creating a large multidimensional table in MATLAB, the compressor maps in Figure 5 and Figure 6 have been broken down into three different tables for T-MATS, which solve for corrected flow (*Wc*), pressure ratio (*PR*), and efficiency (*Eff*) separately. In addition, scalars have been provided to modify the inputs and outputs of this interpolation to modify performance map characteristics. The scaling of the compressor module’s maps is summarized below in Table 4. It should be noted that, for each turbomachinery component in T-MATS, speed and flow are corrected as in Equation (3), where *Nmech* is mechanical shaft speed, *Tin* is total temperature at the block input, *Tstd* is standard day temperature (518.67 °R), *Pin* is total pressure at the block input, *Pstd* is standard day pressure (14.7 psia), and *Win* is flow at the block input.

$$Correct\ Speed\ (Nc) = \frac{Nmech}{\sqrt{\frac{Tin}{Tstd}}}$$

$$Correct\ Flow\ (Wc) = Win\left(\frac{\sqrt{\frac{Tin}{Tstd}}}{\frac{Pin}{Pstd}}\right) \tag{3}$$

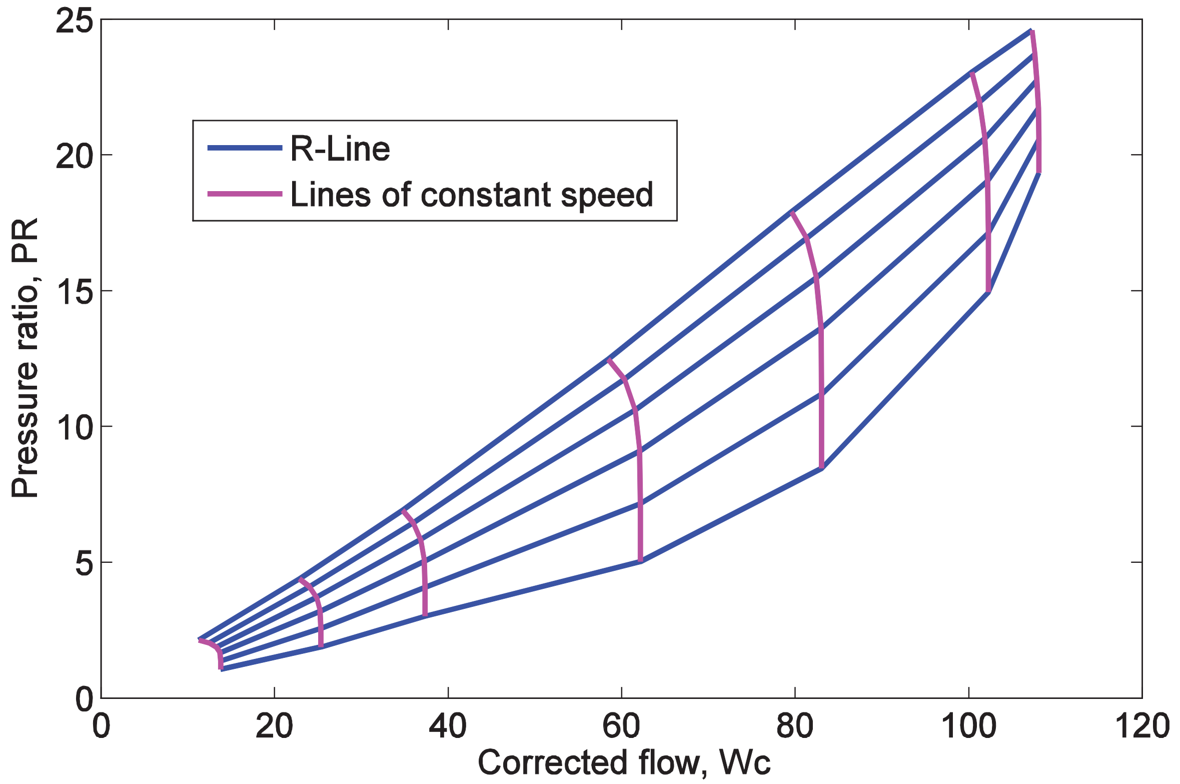


Figure 5.—Sample compressor map (pressure ratio vs. corrected flow).

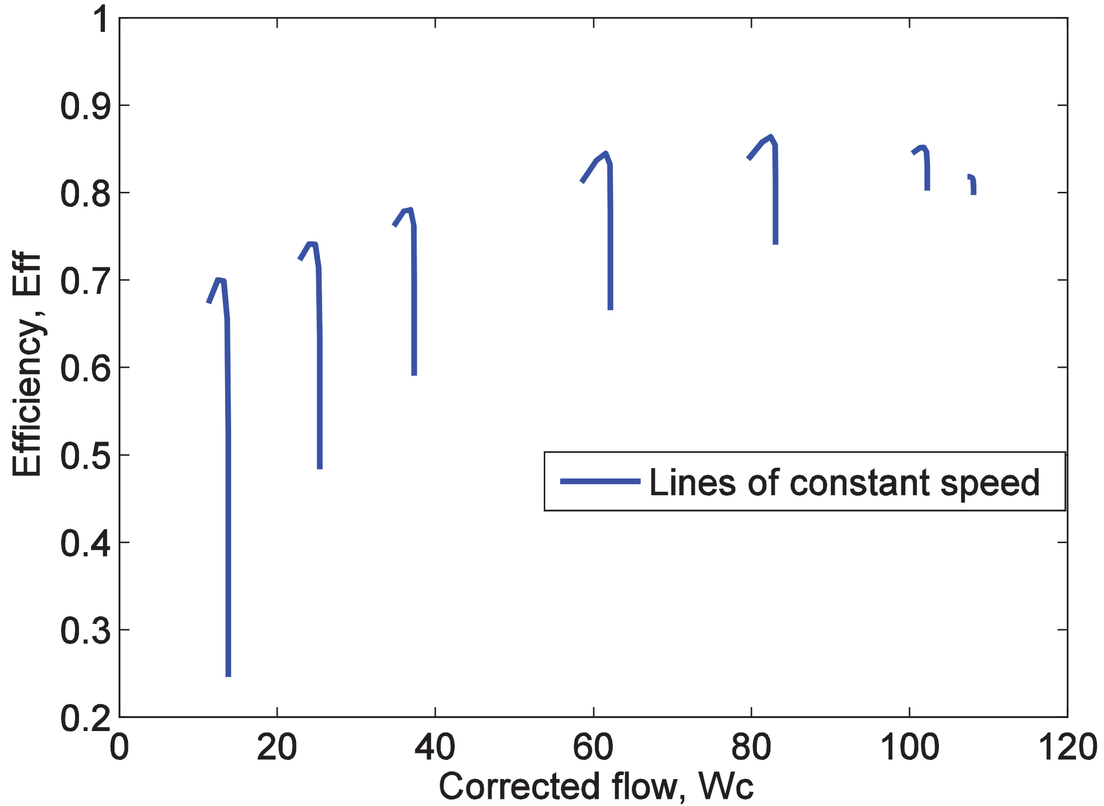


Figure 6.—Sample compressor map (efficiency vs. corrected flow).

TABLE 4.—COMPRESSOR MAP EQUATIONS

Compressor map variable	Map equations
Scaled compressor speed	*Nc-map* = *Nc* * *C_Nc*
Pressure ratio	*PR* = tablelookup(*R-line*, *Nc-map*) * *C_PR*
Corrected flow	*Wc* = tablelookup (*R-line*, *Nc-map*) * *C_Wc*
Efficiency	*Eff* = tablelookup (*R-line*,*Nc-map*) * *C_Eff*

(Note: Table definition variables are not listed, however it should be noted that each 2-D map requires *x*-axis, *y*-axis and table values definition.)

The TMATS: `Turbine` block uses maps to compute the efficiency and corrected flow for the turbine. These two variables are taken as functions of corrected turbine speed (*Nc*) and PR-map value (typically solved for by the iterative solver), as shown in Figure 7, and T-MATS uses two tables to represent the turbine map: one solves for efficiency (*Eff*) and the other for corrected flow (*Wc*). As with the compressor map definition, scalars are provided to allow for modification of performance map characteristics. Scaling of the turbine block's maps has been summarized in Table 5.

.

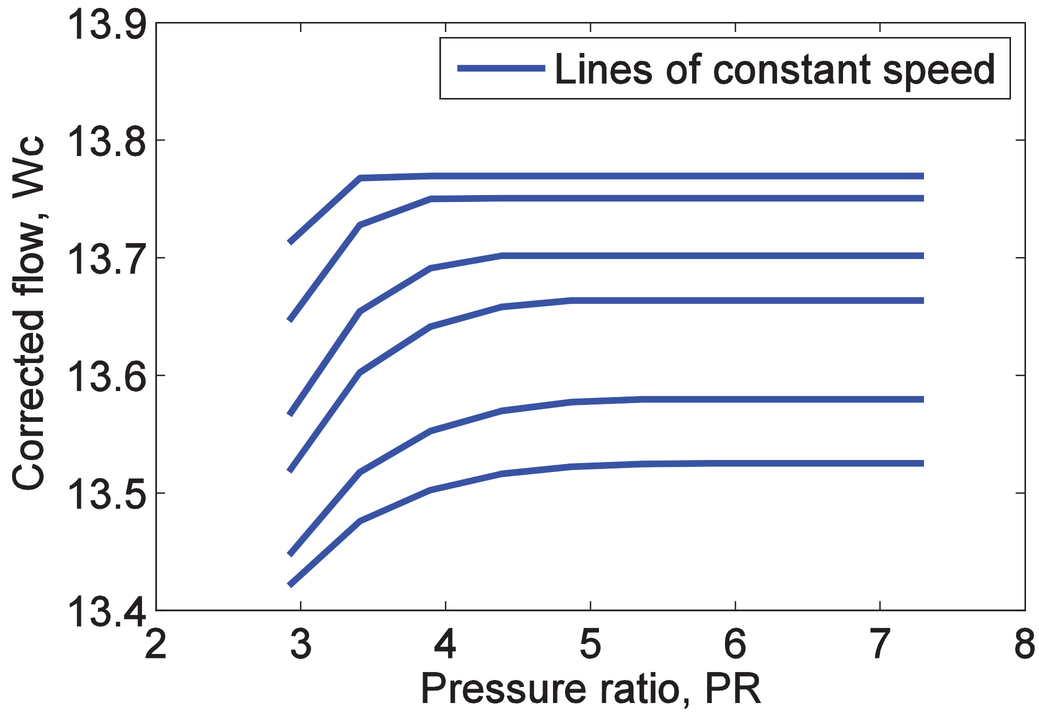


Figure 7.—Sample turbine map (corrected flow vs. pressure ratio).

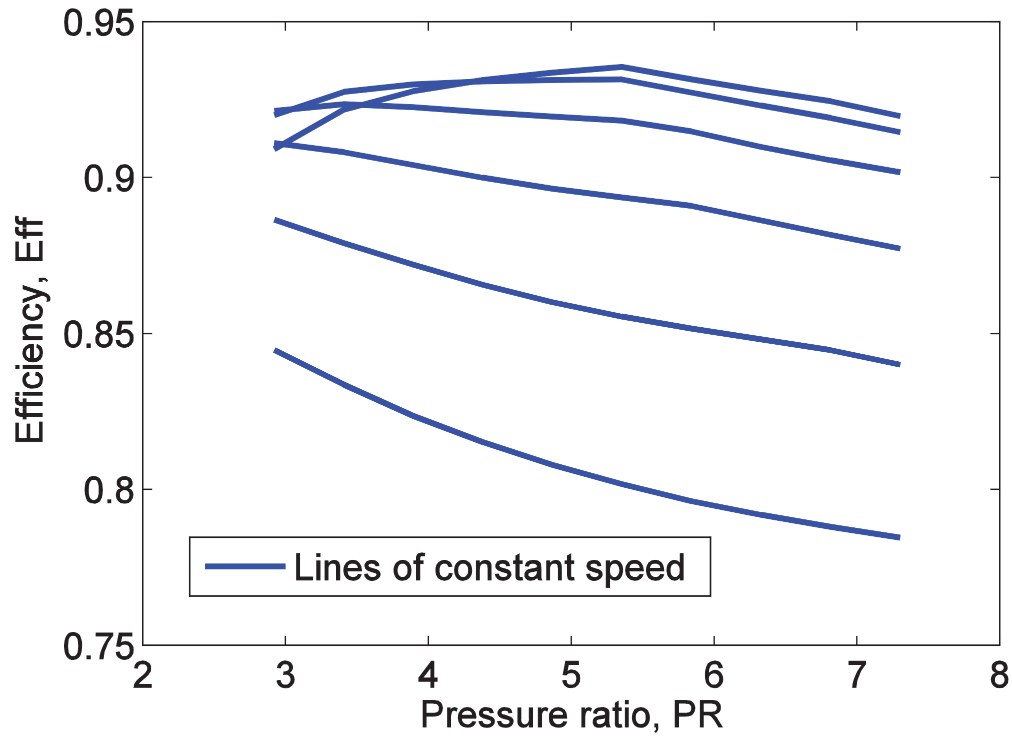


Figure 8.—Sample turbine map (efficiency vs. pressure ratio).

TABLE 5.—COMPRESSOR MAP EQUATIONS

Turbine map look up	Map equations
Scaled turbine speed	*Nc-map* = *Nc* * *C_Nc*
Scaled pressure ratio	*PR-map* = *PRinput* * *C_PR*
Corrected flow	*Wc* = tablelookup (*PR-map*, *Nc-map*) * *C_Wc*
Efficiency	*Eff* = tablelookup (*PR-map*, *Nc-map*) * *C_Eff*

(Note: Table definition variables are not listed, however it should be noted that each 2-D map requires *x*-axis, *y*-axis and table values definition. Also, turbine-defined parameters are independent of compressor-defined parameters.)

T-MATS has been designed to be as flexible as possible; however compressor and turbine maps can be defined in various ways. In cases where the maps are not compatible and cannot be adjusted for use with the T-MATS block sets, new modules must be created to be compatible with the map version that is desired.

One case where a map can be converted for use in a T-MATS module is when a compressor EM-line is used. An EM-line is an R-line that has been forced to take the form of a linear equation. The compressor module for T-MATS is designed to use tables for the map, not equations, but a translation from one map style to the other is possible by creating an R-line from the EM-line through calculation of various R-line points using the EM-line equation. Take, for example, the case where corrected flow (*Wc*) is a table lookup based on compressor speed (*Nc*) and EM-line, and the compressor pressure ratio and efficiency are defined as in Equation (4).

$$PR = 5 + EM\text{-}line\ (Wc + 100)$$
$$Eff = \text{tablelookup}(Nc, EM\text{-}line)(PR^{0.1} - 1) \quad (4)$$

To convert this map to an R-line based map, pairs of *Nc* and EM-line values may be chosen for calculation of *PR* and *Eff* tables based solely on *Nc* and EM-line (EM-line values can essentially be used in place of R-line values). The table for *Wc* is already in an acceptable form, so translation is not required. In this example, all T-MATS scalars would be equal to 1 as they do not appear in the compressor map definition.

4.4.3 Compressor variable geometries (IGVs and VSVs)

Variable geometries (VGs) within the compressor perform essential functions in many modern large engines. Inlet guide vanes (IGVs) and variable stator vanes (VSVs) are two such geometries that may need to be taken into account when developing a gas turbine engine model in T-MATS. In the TMATS: `Compressor` block VG positions must be taken into account in the compressor maps. At this time, T-MATS does not have the ability to produce off-nominal transients of the VGs; however this feature is planned for future releases.

4.4.4 Bleeds and air bypass

High pressure air generated by turbomachinery is a versatile commodity with a wide range of uses. In T-MATS, the removal of air via “bleeds” or air bypasses can be modeled in a few ways: inter-stage bleeds, such as customer bleeds and fractional bleeds, may be enabled for the TMATS: `Compressor` module and the T-MATS: `Splitter` block allows for the creation of bypass flow. The customer bleed is a constant demand for flow while fractional bleed is a fraction of the total flow through the compressor.

Typically, customer bleed is air demanded for use by a system outside of the engine (e.g., to cool the cabin, run power generators, etc.) and fractional bleed is used as cooling flow in turbine components. Bleed flow outputs are vectors of flow path variables (color-coded blue as described in Table 2) combined in series and contain all information required for general processing. As currently written, twenty bleed flows of each bleed type may be requested in a block. Each bleed flow may be “pulled off” the bleed vector output and routed into the intended/appropriate input, such as cooling flow for a turbine.

Main air bypasses in the model may be created using the TMATS: `Splitter` block, which typically work in conjunction with a nozzle block. See Section 4.4.5 for more details about how to use a splitter to create a high bypass engine model.

4.4.5 Turbofan modeling options

Modeling a turbofan engine may require the creation of a fan module and the adoption of a system architecture that partitions flow into a main flow path and a bypass flow path. This architecture can be set up in several ways, each of which makes use of compressor, splitter, and nozzle T-MATS modules. In T-MATS, a fan may be modeled as a one-stage compressor, or the two flow paths in a high bypass engine (commonly called the tip and hub flows) may instead be modeled using one of the three flow-splitting modeling architectures in Figure 9:

Option 1. Split the main flow path before the fan, modeling the tip and hub flows separately with two compressor blocks.
Option 2. Dedicate separate input flows to the tip and hub, modeling each with separate compressor blocks.
Option 3. Split the main flow path after the fan, modeling the entire fan with one compressor block (tip and hub flows are combined).

The architecture chosen for a model will mainly be determined by the available maps, and the options in Figure 9 are by no means the only options for modeling tip and hub flow. It should be mentioned that the inclusion of a bypass flow creates additional independent and dependent variables that must be added to the solver. The additional independent variables are the input flows, R-lines, and/or splitter bypass ratios for the added components, while the additional dependent variables are the flow errors associated with these same components.

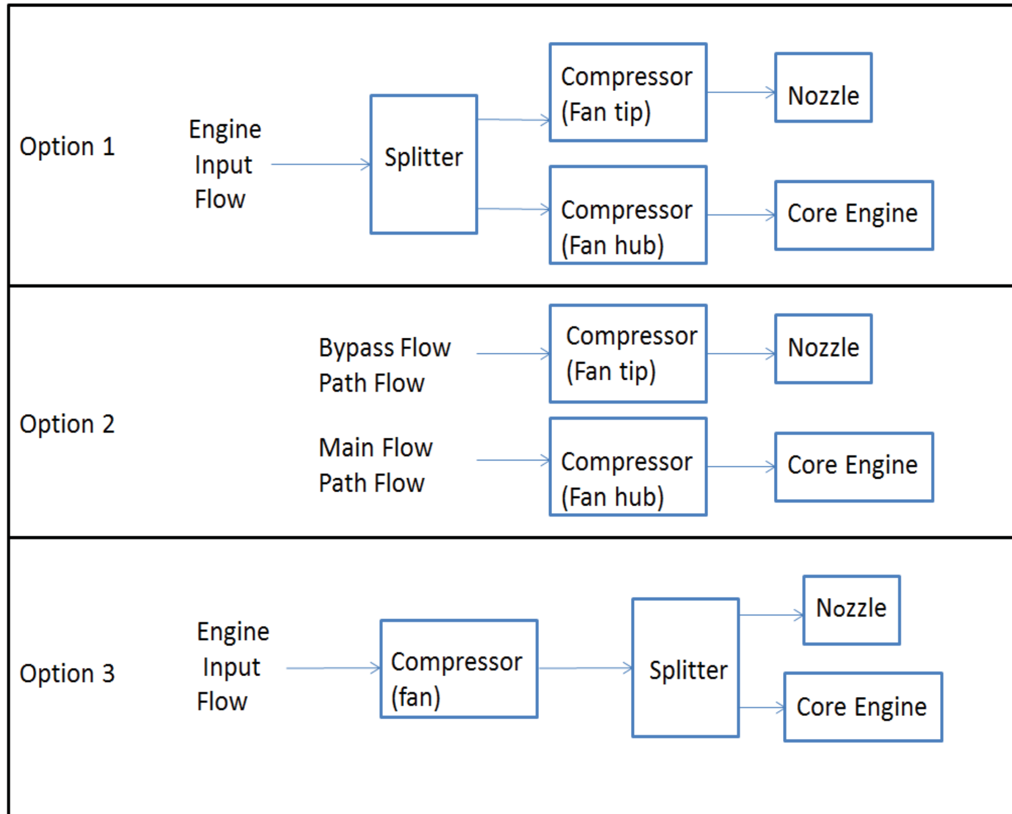


Figure 9.—Sample fan model creation schemes for a gas turbine with bypass.

4.4.6 iDesign

Certain T-MATS turbomachinery blocks have the capability of calculating engine map scalars, based on design or steady-state input values, through the process called iDesign. This process may be turned on through each block individually, by checking the box next to iDesign in the mask, or may be turned on for all the blocks in a model using the iDesign_on tool in the T-MATS Tools menu (Diagram->TMATS Tools). T-MATS blocks that contain maps (compressor and turbine modules) require certain design or steady-state variables to be known and entered into their masks in order for iDesign to work; the help menu for each of these blocks should be consulted to determine what is required. Once this information has been defined, the simulation can be run. Typically, an iDesign run involves simulation of a steady-state model to allow the solver to converge to its design point (or a suitable steady-state value). Once complete, the scalar for each block that contains maps can be found in the **_Data* output bus. These values should be recorded and loaded into the block's mask as the new scalar values. Simulation may then be run as normal after iDesign is turned off (via the iDesign_Off option in the T-MATS Tools menu). Be aware that running T-MATS dynamically with iDesign enabled will produce unpredictable results that should not be used. Also note that, in cases where the maps have not been scaled, the values calculated by iDesign should be near 1. Otherwise, in cases where the maps have been scaled (as in *Example_Gas_Turbine_SS*) the values calculated by iDesign may vary widely from 1.

5.0 Customization

Although T-MATS blocks are designed to be as general as possible, a user may need to customize an existing function within T-MATS, or create an entirely new function, to meet the needs of a specific application. This section will go over how to add new blocks to the library and how to change existing blocks to meet simulation needs.

The library file structure, in Table 6, reflects how T-MATS blocks are displayed in the Library Browser.

TABLE 6.—T-MATS LIBRARY FILE STRUCTURE

File	Map equations
slblocks.m	MATLAB script. Highest Level file, contains library definition for the Simulink Browser Library
TMATS_Lib.slx	Library file, contains subroutines that act as folder links for libraries
TMATS_FolderName.slx	Library file, contains library links to the various T-MATS blocks
Lib_FolderName_BlockName_TMATS.slx	Library file, contains individual T-MATS blocks

5.1 Library Blocks

The T-MATS library is essentially a grouping of useful Simulink blocks that can be connected together to create complex thermodynamic systems. By adding new library blocks, it is possible to expand and customize the simulation environment to meet the needs of the simulation. A new block can be added to a T-MATS library by performing the following steps:

1) Navigate to the \TMATS_Library folder in your T-MATS installation directory.
2) Open the Simulink Library Browser and create a new library by clicking File -> New -> Library (or open and rename an existing T-MATS library file with a file name to reflect the contents of the new library).
3) Create a block in the library file, making sure that the highest view in the hierarchy contains only a subsystem. The library file must be saved before it can be linked to a T-MATS library.
4) Open the file of the T-MATS sub-library into which the new block will be linked (e.g., *TMATS_Solver*) and drag the new block into that window to create the link. Note that Simulink may prompt the user to unlock the library before it can be edited. This can be done by clicking Diagram-> Unlock Library.
5) After refreshing the Simulink Library Browser (F5), the new block will appear in the window.

5.1.1 Creating new Simulink Library Browser folders

Simulink Library Browser folders are an easy and convenient way to organize blocks contained in a library. The file “slblocks.m” is used to add libraries to the highest level of the Simulink Library Browser’s hierarchy. For T-MATS, this file links the high-level Library block, *TMATS_Lib* (which contains all the block “folders” for T-MATS), to the Simulink Library Browser. When adding new blocks to T-MATS that do not fit into the current library organization, it is suggested to add another folder to the library by augmenting a new “folder” to the *TMATS_Lib.slx* library file following these steps:

1) Open *TMATS_Lib.slx*, located at TMATS\TMATS_Library.
2) Create a subsystem with no inputs or outputs, giving it an appropriate name. Note that the library will need to be unlocked (following the prompt from Simulink) before it can be edited.
3) Right click the block and select "Properties" from the menu. Under the "Callbacks" tab, select OpenFcn and add the name of the library file containing the "folder" to the contents area (for example, *TMATS_Turbo*).

5.2 S-functions

In T-MATS, many blocks utilize Simulink S-functions. S-functions allow for the integration of Simulink with code written in another programing language (in this case, C or C++), imparting a number of advantages over using traditional Simulink blocks. One advantage is speed: while many Simulink blocks and MATLAB functions can run quite fast, compiled C code typically runs much faster. A second advantage is that using S-functions allows for the use of legacy code, which may not be written in Simulink/MATLAB, in simulation development. (Legacy code, however, does require some modification to match the special formatting requirements of S-functions.) It should be noted that in order to use C code in a simulation, a MATLAB-compatible compiler must be installed and the code must be compiled (using the MATLAB *mex* function) to produce a MEX file.

S-functions calls are created in Simulink by placing in the model the `S-function` block located within the Simulink -> "User-Defined Functions" sub-library in the Simulink Library Browser. By default, S-function code in T-MATS has been written in C. Starting with an example S-function as a template (such as the marked up function in Appendix A), modifications are generally only needed to the definitions at the top of the file, and to the `mdlInitialSizes` and `mdlOutputs` functions, to create any functionality. A quick reference for the most common MATLAB/Simulink S-function programming functions is listed below:

1) `ssGetSFcnParam`: get a parameter from the dialog box.
2) `mdlInitializeSizes`: initialize the size of the input and output ports
3) `mdlInitializeSampleTimes`: initialize code sampling characteristics
4) `mdlOutputs`: includes the main body of code and defines the outputs
5) `mdlTerminate`: ends the S-function, includes code that executes at the end of the simulation

For more information on S-function block setup, creation of MEX files, or functions prefixed with an *mdl* or *ss*, see the MATLAB help functions. Additional T-MATS code that may be used as examples can be found in the TMATS\TMATS_Library\MEX\C_code folder.

5.3 Tool Creation

T-MATS makes use of internal MATLAB menu functions, which act to call .m scripts each time the menu item is selected. The setup of these menu functions is handled by the file *Menu_customization_TMATS.m*, located within the TMATS\Tools folder. Once T-MATS is installed, this script is called each time MATLAB is opened to create the items in the T-MATS tools menu using the `sl_customization_manager` command.

6.0 Troubleshooting T-MATS

6.1 Convergence "Errors"

When running T-MATS modules in a loop with solver blocks, the simulation will first act to converge the plant. In a simulation with new compressor and turbine maps, it is common for discrepancies in parameters to be present. These discrepancies are the equivalent of placing the wrong hardware on the gas turbine (e.g., a gas turbine with a compressor that is too small) and will cause problems with plant convergence. Failures in convergence often result in Simulink notification of errors in blocks that seemingly have nothing to do with the solver, such as integrators or state-space blocks, which makes troubleshooting the error more difficult. This is why, if an error occurs when running a model, the value of dependent variables (*f(x)*) routed to the solver should be checked first. If any of the values are inf, NaN, or outside the specified iteration condition limits for the solver, the simulation is not converging. It should be noted that in some cases T-MATS blocks will issue errors. The reason for these errors will be justified in the error message and they will not be reviewed in this document.

Troubleshooting a convergence issue can be very difficult. Outlined here are a few ways to troubleshoot a suspect simulation:

1. **Double-check block inputs:** Check for errors in the maps and constants used in the simulation.
2. **Check solver inputs:** Improper independent variable initial conditions, perturbation size, etc. may cause the solver to get stuck in a local minimum that does not result in system convergence.
3. **Check independent/dependent variable matching:** Verify that all system independent variables and dependent variables are accounted for in the simulation/solver.
4. **Component testing:** If data is available (from another example of the system) at the input and output of each component (station), the simulation can be broken down to the component-level, feeding the inputs observed from the other model to the inputs of each block. This can help isolate a specific block that is not modeled correctly.
5. **Steady-state testing:** If the system is intended to be run dynamically, try running it in steady-state first to identify if a problem with the solver parameters is contributing to the convergence problem.
6. **Run without solver:** Take out all the solvers and run the simulation with constant input to each independent variable; if there is another example of the system that the model can be compared against, doing so can help find errors.
7. **Analyze the Jacobian:** It may be useful to look at the Jacobian (*J*), which shows the relationship between independent variables and dependent variables in the system, as the solver tries to converge to a solution. This value is located within *S_Data* bus output from the solver blocks and can be displayed using the Simlunk: `Display` sink block. Given an understanding of which independent variables should affect which dependent variables, the values of the Jacobian may be used to determine a possible source of error in the model.
8. **iDesign:** The iDesign feature, which can be turned on for each block, forces the simulation to converge; this feature may be used to help determine how "close" to convergence the simulation is, with the caveat that unrealistic scalars may be produced (e.g., efficiency of a component might be greater than 1). See Section 0 for more information on iDesign.

6.2 Crashing

There may be times when a T-MATS simulation will crash and issue a system error, causing MATLAB to close ungracefully (Figure 10). These internal errors typically arise due to errors in the execution of the compiled C code (generally when the C code attempts to access memory locations that it doesn't have access to). This can occur when:

- Attempting to access elements of arrays outside of the memory range (e.g., element 21 of a 15 element array).
- Using pointers incorrectly.
- Passing arrays to other functions by reference.

One such error has been observed when a length-1 vector or array is provided for a mask parameter used for table lookup. If the simulation crashes, one of the first steps should be to check that the mask parameters for each T-MATS module have proper length. It should be noted this is just one possible reason for the crash.

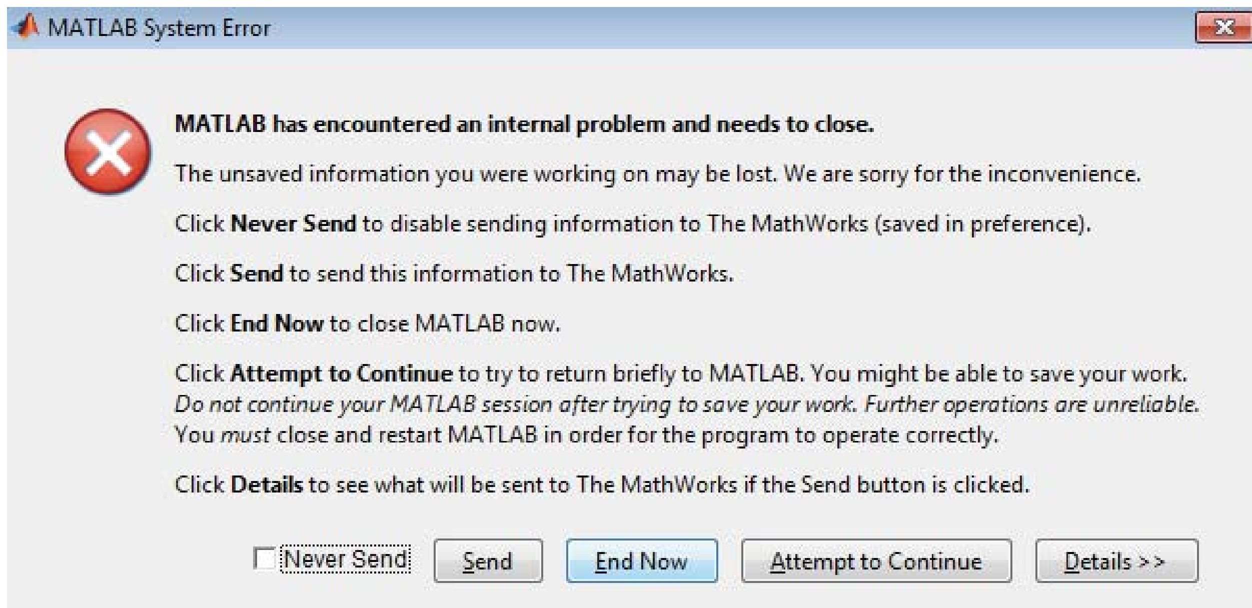


Figure 10.—Sample MATLAB internal problem message.

7.0 T-MATS Tutorials

The T-MATS software package is a Simulink library that can be used to model any thermodynamic system. In order to help the user better understand how the various blocks in the library can be assembled, the software package contains three examples. This tutorial will describe their development and highlight important points that the user should keep in mind when developing simulations. As a user works through the tutorial, it may be useful to have the corresponding example from the T-MATS library loaded for comparison.

7.1 T-MATS Example 1: Newton-Raphson Equation Solver

This section will describe how to combine T-MATS blocks to create a simple equation solver. This solver is included in the examples section of T-MATS (*NewtonRaphon_Equation_Solver.slx*).

1) **Create plant**
   In this example, the plant is fully defined, is dependent only on a set of inputs, has 3 independent variables and 3 dependent variables, and is in the form of polynomials (created with Simulink: `Fcn` blocks). For quick reference, the three functions used in the T-MATS example have been reproduced in Equation (5).

$$\begin{aligned} f_1(u) &= 2 * u(1) - 5 * u(2) + u(3) + 5 \\ f_2(u) &= u(1)^3 + 2 * u(2) \\ f_3(u) &= u(1) + u(2)^2 - 2 * u(3) - 3 \end{aligned} \tag{5}$$

2) **Place the solver**
   In this case, the system is not time-dependent so the steady-state solver block (TMATS: `SS NR Solver w JacobianCalc`) will be used.
3) **Connect the blocks**
   The outputs of the function blocks are dependent only on the inputs to the function blocks. The T-MATS solver block has been designed to accept a vector input and produce a vector output, requiring that the outputs of the function blocks be multiplexed together, and then wired to the input of the solver block, $f(x)$. The output of the solver, X, can be routed directly to the input of each function block, as those blocks accept a vector input (see the diagram later in this section, where displays were added to enable the user to observe results).
4) **Set up the solver**
   Double click the solver. The mask parameters need to be tuned per the application. The length of the vectors *SJac_Per_M* and *SNR_IC_M* must both be equal to the number of independent variables and dependent variables in the system (in this case, 3). See the example model *NewtonRaphson_Equation_Solver.slx* to view the mask parameters for this particular case.
5) **Configure the simulation**
   To avoid encountering errors when running the simulation, the solver options should be set up prior to doing so through the "Configuration Parameters" window, accessed from the "Simulation" option on the Simulink menu bar. Navigate to the "Solver" pane and select the fixed-step, discrete solver as shown in Figure 11.

6) **Run the simulation**
At this point the system can be run. Click the green play button at the top of the Simulink model. If the simulation solver options have been properly set up, the simulation will run successfully.

7) **Execution is complete, interpret results**
The displays will be populated with the solution to the equations (see Figure 12). The element *Converged* in the S_*Data* output vector shows whether the model has converged. This element will be 1 if the convergence criteria have been met, but will be 0 if the solver inputs, *f(x)*, do not approach zero.

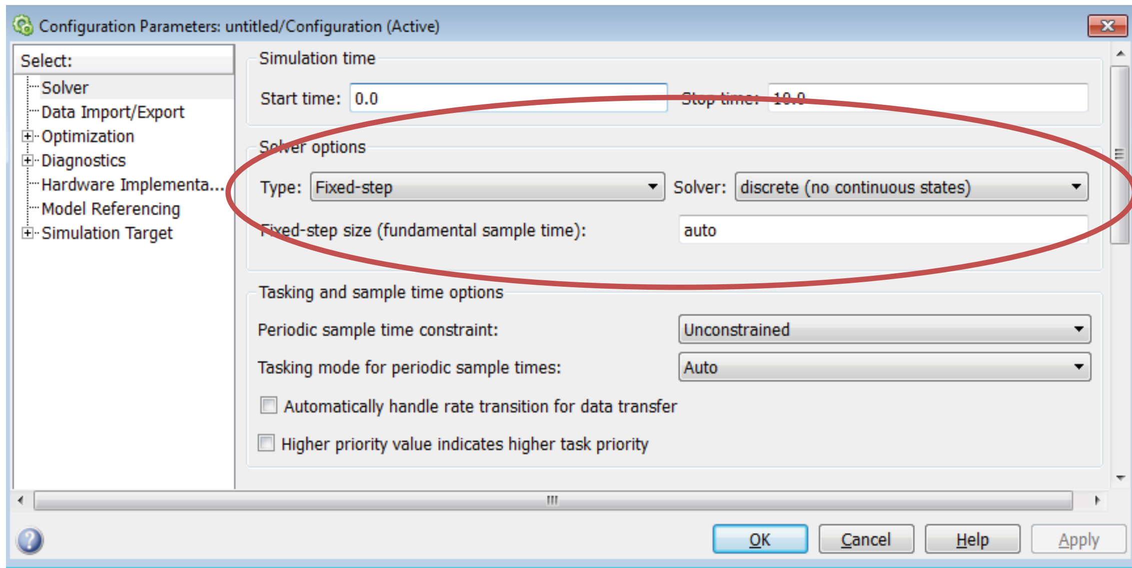


Figure 11.—Model configuration parameters setup.

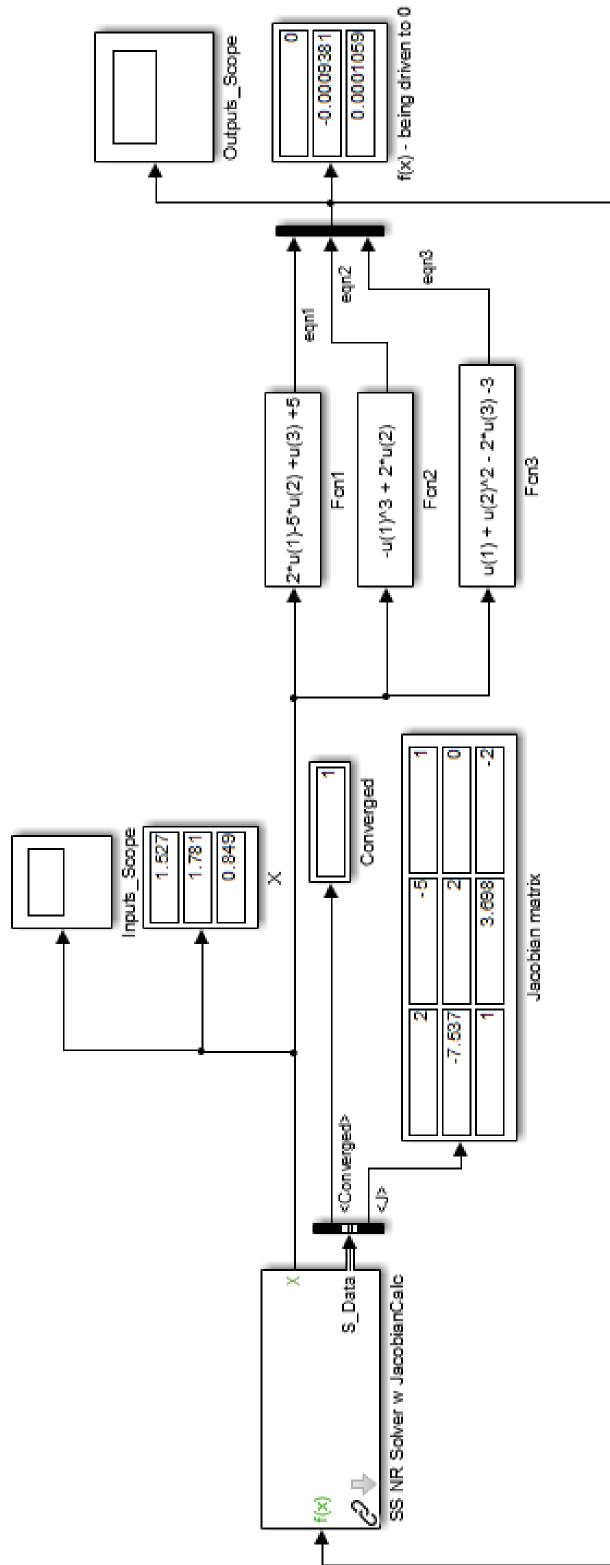


Figure 12.—Run equation solver.

7.2 T-MATS Example 2: Example_GasTurbine_SS

This section will describe how to combine T-MATS blocks to create the simple steady-state gas turbine example included with T-MATS (*GasTurbine_SS_Template.slx*). Note that the example model makes use of the Block Link Setup tool to produce a clean-looking model by replacing lines with Simulink: `GoTo` and Simulink: `From` blocks.

7.2.1 Creating the gas turbine plant

1) **Gather the blocks**
Place TMATS: `Shaft`, `Compressor`, `Burner`, `Turbine`, `Duct`, and `Nozzle` blocks into a model. These are located in the “Turbomachinery” sub-library of T-MATS.
2) **Connect the core gas path**
Line up the blocks in the same order as they are listed in step 1 and connect each *GasPthChar* output to the *GasPathChar* input of the next block. (The shaft is not part of this step.)
3) **Connect the bleed gas path**
Although this example does not use customer bleed or cooling flow, connecting the *FBldsCharOut* output from the compressor with the *CoolingFlwCharIn* input to the turbine will allow for these features to be added later, if necessary.
4) **Route un-used outputs to terminator blocks**
The outputs listed in Table 7 are not used in this tutorial and should be wired to a Simulink: `Terminator` block.

TABLE 7.—STEADY-STATE TURBOFAN ENGINE EXAMPLE, UNUSED OUTPUTS

Block	Unused output
Compressor	*CustBldsCharOut*
	C_Data
Burner	B_Data
Turbine	*T_Data*
Nozzle	*WOut*
	Fg
	N_Data

Note that using the Block Link Setup tool on each block will automatically place the Simulink: `Terminator` blocks at these outputs.

5) **Set up the shaft component**
Connect *NmechOut* (from the shaft) to the *Nmech* input on both the compressor and turbine. Multiplex the *TrqOut* outputs from the compressor and turbine and connect this signal to the *Torque* input on the shaft.

6) **Set up the input flow**
The TMATS: `Ambient` block converts environmental characteristics to gas turbine thermodynamic characteristics; place this block in the model and connect the *GasPthChar* output to the *GasPthChar* input of the TMATS: `Compressor` block. The ambient pressure (*Pamb*) output of the TMATS: `Ambient` block needs to be routed to the *Pamb* input of the TMATS: `Nozzle` while the *A_Data* output is unused and can be routed to a Simulink: `Terminator` block.
Set inputs for the environmental conditions (TMATS: `Ambient` block) and fuel flow (TMATS: `Burner` block) to constants, since the simulation will solve for steady-state.

7) **Define the mask parameters**
Constants and maps for each component should be updated per the user's particular application. See the example model for values usable for this particular case. (Note that the actual values for each parameter are in the setup file, which creates the variables specified for the mask parameters in the Simulink model file.) The completed gas turbine plant model should resemble Figure 13, where Simulink: `Display` blocks have been added at the output of each component.

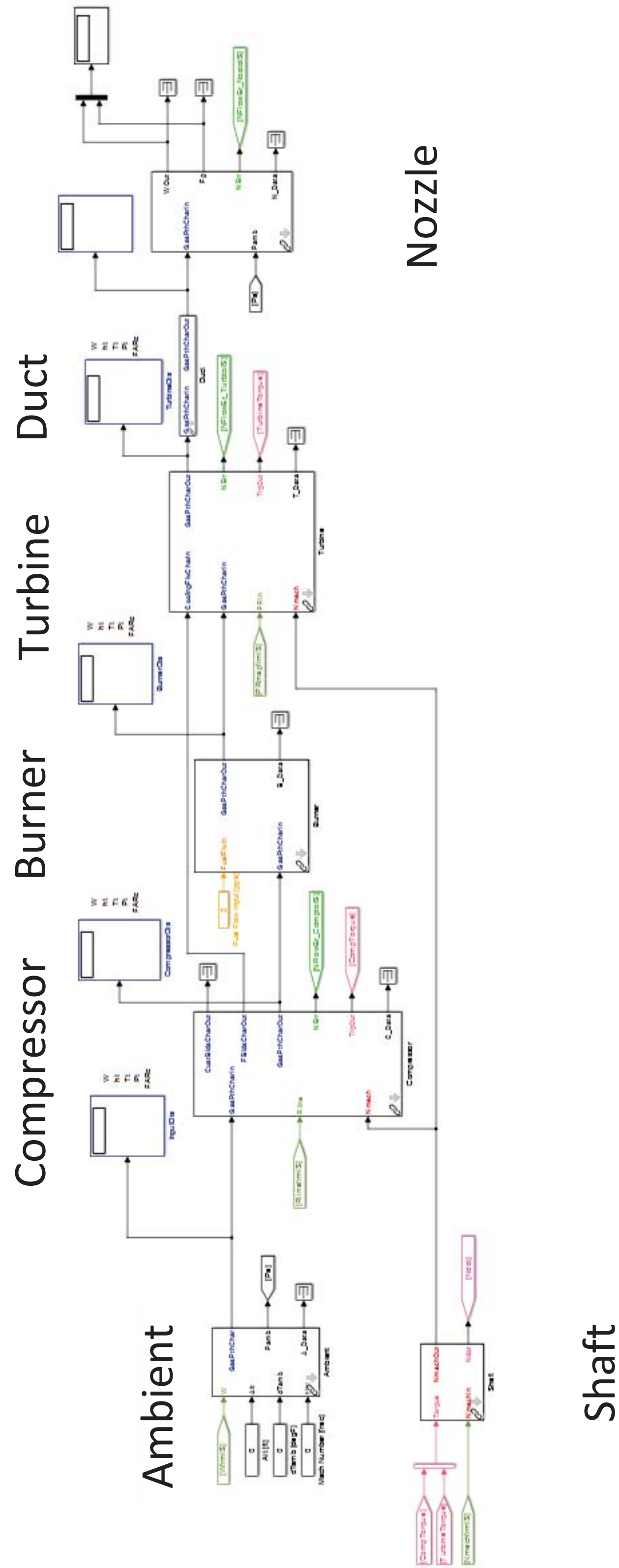


Figure 13.—Gas turbine plant.

7.2.2 Plant solver integration and plant inputs

1) **Create the solver**
   Place the TMATS: `SS NR Solver w Jacobian Calc` block in the model. Multiplex all dependent variables (component outputs color coded in green) and *Ndot* (from the shaft) and route the signal to the *f(x)* input of the solver block. De-multiplex the output of the solver block, *X*, and route each independent variable to one of the dark green inputs of the component blocks; one of the independent variables should be routed to the *NMech* input of the shaft block. In this example there are four independent variables and four dependent variables, as shown in Figure 14, where display blocks have been added to the solver input and output to aid in determining if the solver has converged to a solution.
2) **Set up the solver parameters**
   Once the signals have been connected, the solver mask parameters should be defined; these parameters include the initial conditions, the termination conditions, and other values that affect the accuracy of the solver. See the example for the values used for this model.
3) **Configure the solver**
   Open the simulation configuration window (Simulation -> Model Configuration Parameters on the Simulink menu bar) and select a fixed-step discrete solver, specifying the start and stop times as well as the fixed step size. (See the tutorial in Section 7.1 for more information on setting up the solver for simulation.)
4) **Running the simulation**
   Once the engine model has been created and connected to the solver, the simulation can be run by pressing the run button located in the menu bar. After completing the run, the display blocks should be checked for convergence. It can be seen in Table 8 that the independent and dependent variables in example 2 are being driven to convergence (that is, the flow errors approach zero).

TABLE 8.—EXAMPLE 2 CONVERGENCE OUTPUTS

Independent variable	Independent request (value)	Dependent variable	Errors (Dependent value)
W (system input)	101.2 pps	Normalized flow error nozzle	6.413e-06 N/A
R-line (compressor)	1.955 N/A	Normalized flow error compressor	6.379e-06 N/A
PRmap (turbine)	3.054 psia/psia	Normalized flow error turbine	4.884e-06 N/A
Nmech (shaft)	9980 rpm	Ndot (shaft)	0.007742 rpm/sec

Because the steady-state gas turbine example requires definition of a large number of block parameters, the example was developed to utilize a setup script (the filename has the suffix "*_setup_everything*"). When run, this script creates a path to the SimSetup folder and runs all the files in that folder, which create the variable *MWS* in the workspace. This variable is a MATLAB data structure that contains many of the variables used in the example, which can be accessed with the syntax: *MWS.Component.Variable* (for example, *MWS.HPC.s_Wc* can be used to access the scalar for the *Wc* map in the HPC). When the simulation is run, Simulink will look for *MWS* in the MATLAB workspace and load the required variables. The example has been configured to automatically remove the temporary paths (created by the setup script) by running a cleanup script when the model file is closed.

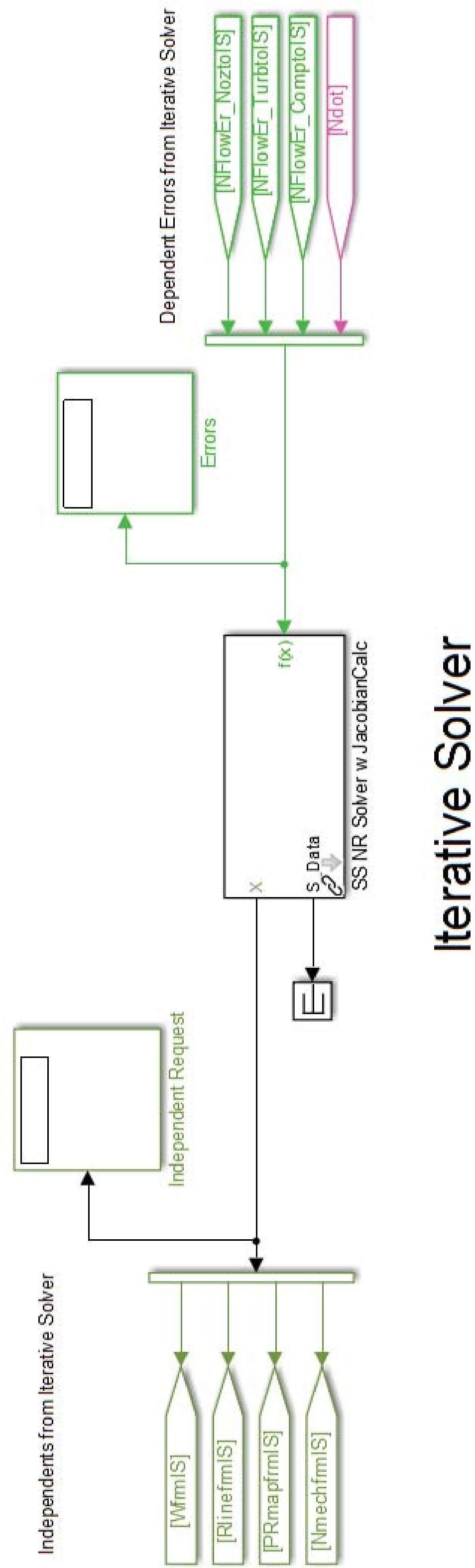


Figure 14.—Steady-state solver setup.

7.3 T-MATS Example 3: Example_GasTurbine_Dyn

Although it has the capability to solve for a steady-state condition, T-MATS is designed to ultimately be used in a dynamic simulation environment. This section will describe how to create a dynamic gas turbine simulation using T-MATS blocks and will also provide demonstration of various MATLAB practices the user may wish to utilize when developing a new simulation. This example is included in the examples section of T-MATS (*GasTurbine_Dyn_Template.slx*).

Before attempting to recreate the dynamic gas turbine simulation example, the user should read through the previous two examples, as certain topics covered in them will be skipped here. In particular, the description of how to structure the model assumes that the engine plant model has already been constructed, following the instructions presented in Section 7.2 for the steady-state gas turbine simulation.

7.3.1 General architecture

Implementing a dynamic solver in T-MATS requires that at least two separate loops be set up. This example uses an "inner" loop, which solves for convergence, and an "outer" loop, which steps through time. The "inner" loop includes the plant and the solver, and is where the independent variables are solved for from the dependent variables. The "outer" loop sets up the dynamic portion of the simulation and represents evolution of the system, which is converged at each time-step, over time. A hierarchical approach is required when creating such nested loops in Simulink (see Section 4.1 for more details).

7.3.1.1 Creating a dynamic simulation framework

1) **Set up the inner loop**
   Starting with the steady-state model described in the previous section, place a Simulink: `While Iterator Subsystem` in the model. Enter the subsystem and double-click the Simulink: `While Iterator` block to open the parameter window. Make the following changes: set the maximum number of iterations to -1, change the loop type to "do-while", and check the box next to the "Show iteration number port" option.
2) **Prepare the steady-state model for dynamic simulation**
   Delete the default input and output ports in the Simulink: `While Iterator Subsystem`. Copy and paste the entire steady-state model (plant and solver blocks) into the subsystem. It may be helpful at this step to place the plant in its own subsystem within the while-iterator subsystem, as shown in Figure 15, where the low-level gas turbine components have been placed in the InnerLoopPlant subsystem.
3) **Add the dynamic solver**
   Replace the TMATS: `SS NR Solver w JacobianCalc` block with the TMATS: `Iterative NR Solver w JacobianCalc` block and connect the *do_while_Condition* output of the T-MATS solver to the *cond* input of the Simulink: `While Iterator` block, and the output of the Simulink: `While Iterator` block to the Iterations input of the T-MATS solver block, as shown in Figure 15.

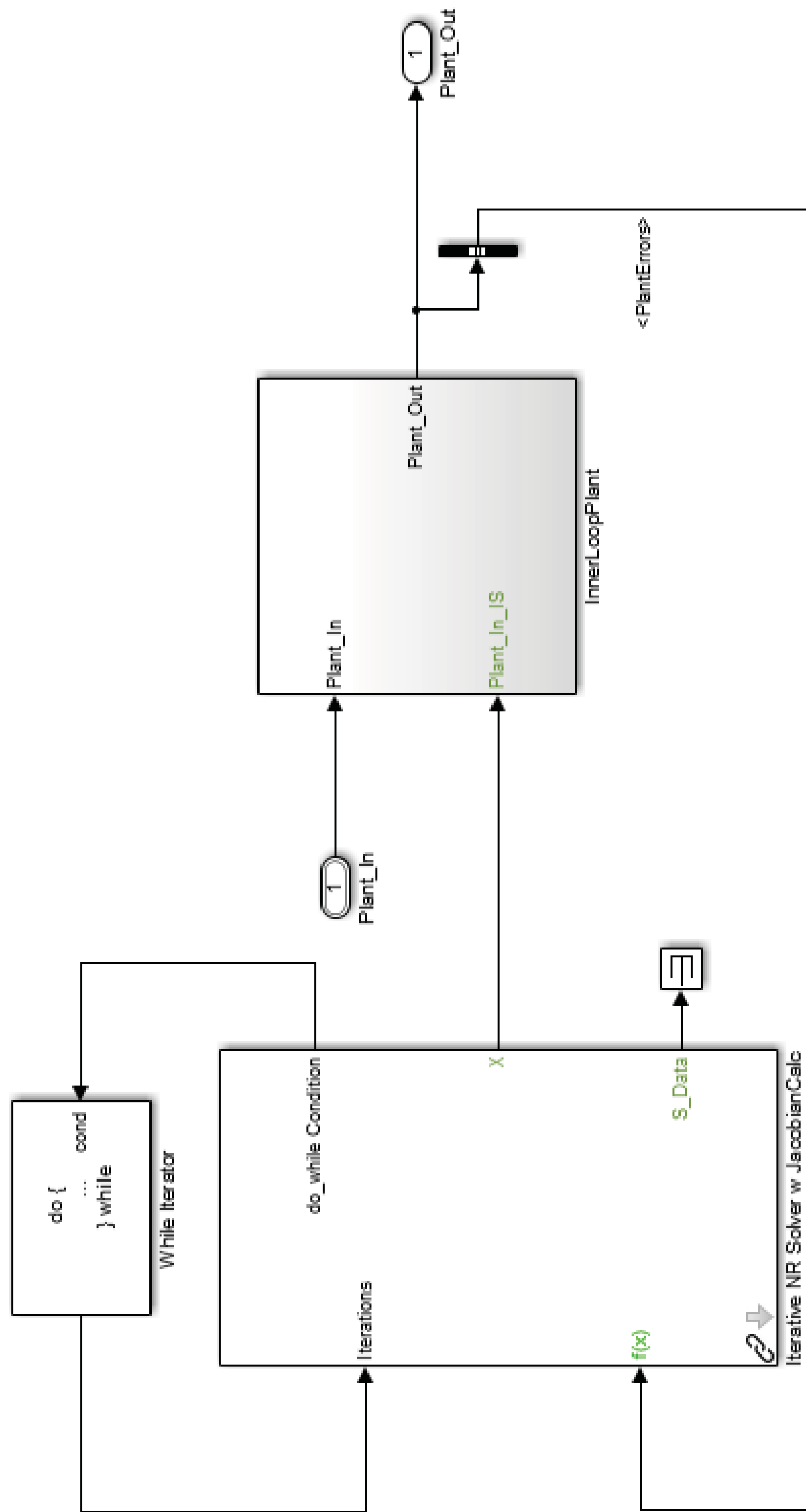


Figure 15.—`Iterative NR Solver w JacobianCalc` wiring.

4) Set up the solver

In the dynamic simulation, shaft speed is not one of the solver independent variables; reduce the order of the system by removing *Ndot* from the input to the solver block, and *NMech* from the output of the block, at the same time routing *Ndot* to the outer loop of the model and creating an input (for *NMech*) from the outer loop to the plant model. (A bus selector may be used for routing signals; see Section 7.3.4.1 for more information on this modification.)

5) Setup the outer loop

Outside of the while iterator subsystem, route *Ndot* to the input of the TMATS: `OuterLoop integrator` block from the "Solver" sub-library (a bus selector may be used) and route the output of the integrator block to the NMech input of the plant (in the inner loop).

To speed up the ability to modify the model when running multiple simulations, the environmental inputs (to the TMATS: `Ambient` block) and the fuel flow input (to the TMATS: `Burner` block), may similarly be routed from the outer loop to the inner loop. Bus creators may be used to combine these signals with *NMech*, as shown in Figure 16, to produce a cleaner-looking model.

6) Define the plant inputs

Environmental inputs and fuel flow may be defined by connecting TMATS: `Model Source` blocks (from the "Solver" sub-library) to the respective inputs. The mask parameters for these blocks include a time vector and a vector that defines the values the signal takes at each time; the simulation profile can be defined through these inputs.

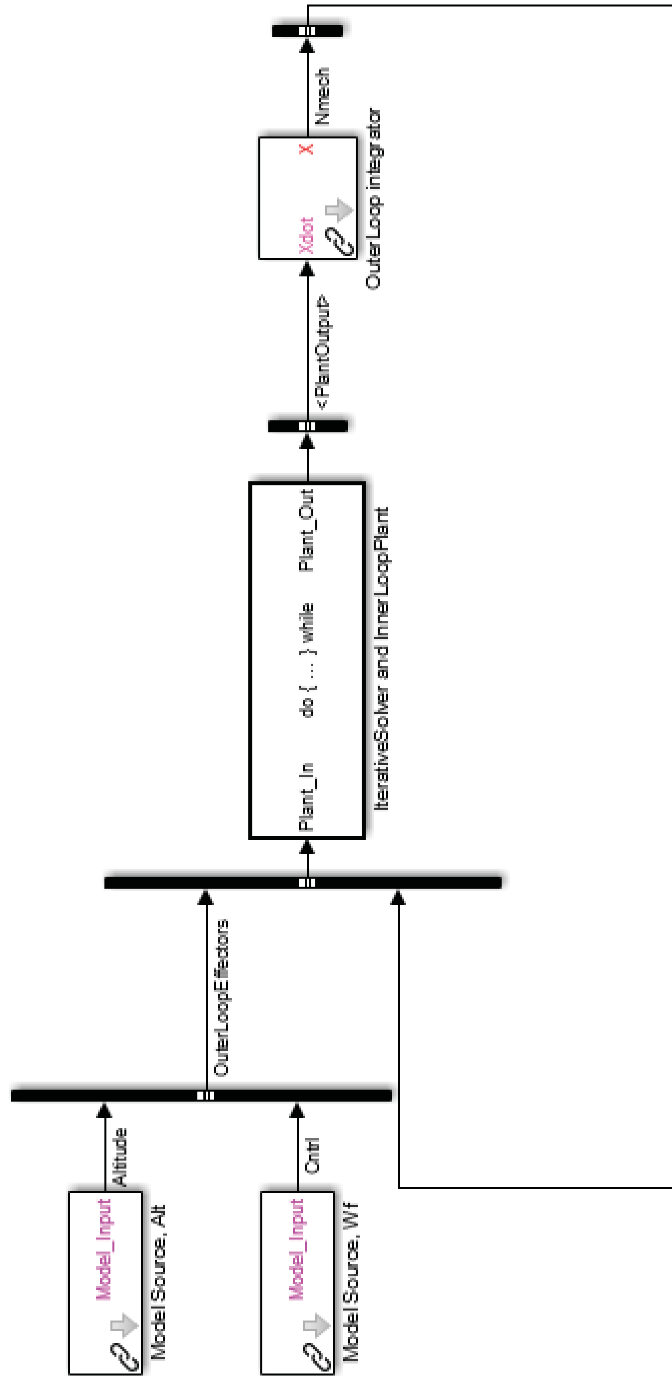


Figure 16.—Dynamic gas turbine simple “outer” loop.

7.3.2 Adding control to the simulation

This section will describe how to add a simple feedback controller to the T-MATS simulation that uses shaft speed as the feedback signal.

1) **Prepare the open-loop model for a controller**
   Replace the fuel flow TMATS: `Model Source` block with a TMATS: `Model Source` block for the shaft speed and place a TMATS: `Simple PI controller` block (from the "Effectors and Controls" sub-library) in the outer loop of the model
2) **Connect the controller**
   Wire *Nmech*, the output of the outer loop integrator block, to the *Input_sensed* input of the PI controller (this may require adding a Simulink: `Zero-Order Hold` block to sample *Nmech*). Also wire the output of the TMATS: `Model Source` block for shaft speed to the *Input_dmd* input of the PI controller, as shown in Figure 17. The output of the PI controller block should be wired to the fuel flow input of the burner block in the inner loop (i.e., inside the subsystem with the Simulink: `While Iterator` block).
3) **Tune the controller**
   The gains of the controller (defined in the mask parameters) may be tuned until the desired response is obtained. Note that although the PI controller may function adequately in a simple gas turbine, it does not represent a realistic gas turbine controller and will not have the ability to account for multi-variable constraints without significant modification.

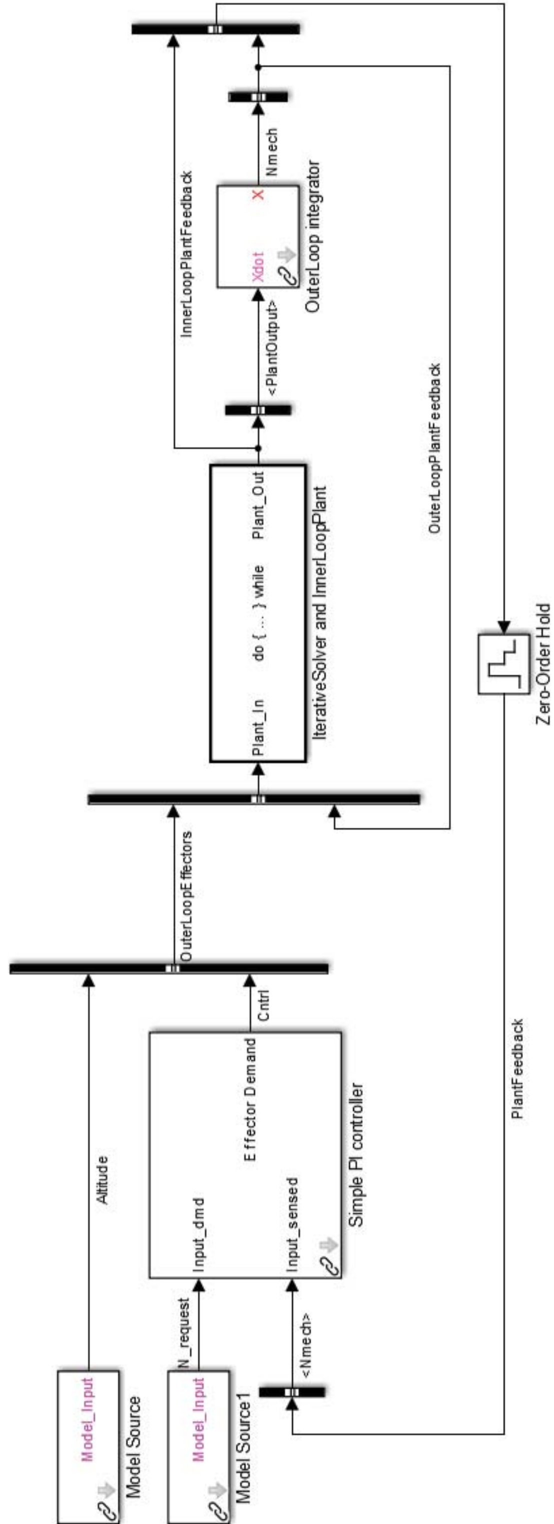


Figure 17.—Dynamic gas turbine with controller.

7.3.3 Plant interface with outer loop systems

In a real system, engine effectors and internal states need to be accessed with control system hardware. The T-MATS package comes with generic controller interface blocks, such as the first-order sensor and actuator blocks in the "Effectors and Controls" sub-library, which may be used to simulate the dynamics introduced with added hardware. These blocks may be added to the "outer" loop of the model to increase simulation fidelity.

7.3.4 Advanced simulation structure and formatting

The dynamic gas turbine example employs a few optional structural and formatting techniques that help to organize and partition the simulation. These options are useful when developing a complicated model or when several developers are collaborating to create the model. The techniques described in this section are based on MATLAB/Simulink blocks and are independent of the T-MATS package.

7.3.4.1 Busses

Busses are a useful way to route signals through a Simulink model while keeping the diagram clean.

Creating Busses:

1) Place a Simulink: `Bus Creator` block in the model. Any signal fed into that bus creator will be on the bus and associated with the name of the wire fed into the bus creator. (Note that this is the only step required if a bus stays within a single model.)
2) The Bus Editor (found in the Edit menu) may be used to create bus elements. A bus that has been created in the Simulink diagram should also be created in the Bus Editor, being careful to organize the elements in the same order as they appear in the Simulink diagram.
3) After creating the bus, in the Bus Editor, select "Export to File" in the File menu of the editor. This saves the bus to a file that must be loaded before opening the Simulink model. (For example, the *setup_everything.m* script for the GasTurbine_Dyn_Template example loads all the necessary bus elements from the folder Bus)
4) To add any element to the bus later, these steps must be repeated (i.e., open the bus editor, add the new item, then save the item to a file.)

Using Busses:

1) Bus elements may be routed off the bus through the use of the Simulink: `Bus Selector` block. Simply wire the bus into the selector, and then use the dialog box to select which bus element(s) to output.

7.3.4.2 Model Calls

Model calls may be used to incorporate external files into a Simulink model. This type of partitioning has three main advantages: it allows multiple users to work on the simulation at the same time (provided they don't need the same file at the same time), it enables cleaner software version control, and it enables model reuse.

Setting up a model call requires placing a Simulink: `Model` block, from the "Ports and Subsystems" sub-library, in the model. Once the block has been placed, the file name of the model to be called should be specified in the dialog box opened by double-clicking the block. In the gas turbine example with the

dynamics solver, calls were made to external plant and control models. The simulation may use Simulink: `Model` blocks with busses, which requires that the bus object creation method be followed.

Model Calls can also be used to simulate the plant in multiple simulation environments. For example, a steady–state simulation environment and a dynamic simulation environment could be created that both reference the same plant. This way, instead of having to modify two models to update the plant, a single file (containing the plant model) may be modified and these changes would be reflected in both simulation environments.

Appendix A

```c
/*                    T-MATS -- Ambient_TMATS.c
 * % **********************************************************************
 * % written by Jeffryes Chapman
 * % NASA Glenn Research Center, Cleveland, OH
 * % March 19, 2013
 * %
 * %  This file converts Altitude and MN to common engine input variables.
 * % **********************************************************************/
#define S_FUNCTION_NAME  Ambient_TMATS
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#include "constants_TMATS.h"
#include <math.h>
```

Function parameter inputs.

```c
#define X_A_AltVec_p(S)     ssGetSFcnParam(S,0)
#define T_A_TsVec_p(S)      ssGetSFcnParam(S,1)
#define T_A_PsVec_p(S)      ssGetSFcnParam(S,2)
#define NPARAMS 3
```

Number of parameter inputs

```c
extern double interp1Ac(double aa[], double bb[], double cc, int ii,int *error);
```

C code function call definition

```c
static void mdlInitializeSizes(SimStruct *S)
{
   int i;
   ssSetNumSFcnParams(S, NPARAMS);  /* Number of expected parameters */
   if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
      /* Return if number of expected != number of actual parameters */
      return;
   }

   for (i = 0; i < NPARAMS; i++)
      ssSetSFcnParamTunable(S, i, 0);

   ssSetNumContStates(S, 0);
   ssSetNumDiscStates(S, 0);

   if (!ssSetNumInputPorts(S, 1)) return;
   ssSetInputPortWidth(S, 0, 3);
   ssSetInputPortRequiredContiguous(S, 0, true);
   ssSetInputPortDirectFeedThrough(S, 0, 1);
```

Number of inputs within the first inport

```c
   if (!ssSetNumOutputPorts(S, 1)) return;
   ssSetOutputPortWidth(S, 0, 5);
```

Number of outputs within the first outport

```c
   ssSetNumSampleTimes(S, 1);
   ssSetNumRWork(S, 0);
   ssSetNumIWork(S, 0);
   ssSetNumPWork(S, 0);
   ssSetNumModes(S, 0);
   ssSetNumNonsampledZCs(S, 0);
```

```
   ssSetOptions(S,
          SS_OPTION_WORKS_WITH_CODE_REUSE |
          SS_OPTION_EXCEPTION_FREE_CODE |
          SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
   ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
   ssSetOffsetTime(S, 0, 0.0);
   ssSetModelReferenceSampleTimeDefaultInheritance(S);
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
   /*-------- vector & array data -------*/
   const real_T *X_A_AltVec       = mxGetPr(X_A_AltVec_p(S));
   const real_T *T_A_TsVec        = mxGetPr(T_A_TsVec_p(S));
   const real_T *T_A_PsVec        = mxGetPr(T_A_PsVec_p(S));

   /*------get dimensions of parameter arrays-------*/
   const int_T A   = mxGetNumberOfElements(X_A_AltVec_p(S));

   /*---------Define Inputs--------*/
   const real_T *u  = (const real_T*) ssGetInputPortSignal(S,0);

   double AltIn     = u[0];    /* Altitude(ft)    */
   double dTempIn   = u[1];    /* delta Temperature [degF]     */
   double MNIn      = u[2];    /* Mach Number (frac)  */

   real_T *y  = (real_T *)ssGetOutputPortRealSignal(S,0);  /* Output Array */

   /*--------Define Constants-------*/
   double PsOut, TsOut, TtOut, PtOut, VengOut, TsStDayOut, Vsound;

   int interpErr = 0;

   /*  Static Temperature */
   TsStDayOut = interp1Ac(X_A_AltVec,T_A_TsVec,AltIn,A,&interpErr);
   TsOut = TsStDayOut + dTempIn;

   /* Static Pressure*/
   PsOut = interp1Ac(X_A_AltVec,T_A_PsVec,AltIn,A,&interpErr);

   /*------ Total Temperature ---------*/
   TtOut = TsOut * (1+MNIn*MNIn*(C_GAMMA-1)/2);

   /*------ Total Pressure ---------*/
   PtOut = PsOut/(pow((TsOut/TtOut),(C_GAMMA/(C_GAMMA-1))));

   /*---- Engine Velocity ---------*/
   Vsound = C_MACH1 * sqrt(TsOut);
   VengOut = Vsound * MNIn;

   /*------Assign output values------------*/
   y[0] = TtOut;      /* Total Temperature [degR] */
```

Begin C code for S-function here

Assign “u,” inputs from Simulink

Set variables from

Assign, “y” outputs to Simulink

Set “y,” outputs to Simulink. Continued on next page

```
    y[1] = PtOut;      /* Total Pressure [psia] */
    y[2] = PsOut;      /* Static Pressure [psia] */
    y[3] = TsOut;      /* Static Pressure [degR] */
    y[4] = VengOut;    /* Engine Velocity [ft/sec] */

}

static void mdlTerminate(SimStruct *S)
{
}

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
/*===================*/
```