NASA/TM—2016-218922

An Introduction to Transient Engine Applications Using the Numerical Propulsion System Simulation (NPSS) and MATLAB®

Jeffrey C. Chin, Jeffrey T. Csank, William J. Haller, and Jonathan A. Seidel
Glenn Research Center, Cleveland, Ohio

January 2016

NASA/TM—2016-218922

An Introduction to Transient Engine Applications Using the Numerical Propulsion System Simulation (NPSS) and MATLAB®

Jeffrey C. Chin, Jeffrey T. Csank, William J. Haller, and Jonathan A. Seidel
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

January 2016

Abstract

This document outlines methodologies designed to improve the interface between the Numerical Propulsion System Simulation framework and various control and dynamic analyses developed in the Matlab and Simulink environment. Although NPSS is most commonly used for steady-state modeling, this paper is intended to supplement the relatively sparse documentation on it's transient analysis functionality. Matlab has become an extremely popular engineering environment, and better methodologies are necessary to develop tools that leverage the benefits of these disparate frameworks.

Contents

List of Figures

List of Tables

1 Introduction

Transient analysis is not a new feature of the Numerical Propulsion System Simulation (NPSS), but transient considerations are becoming more pertinent as multidisciplinary trade-offs begin to play a larger role in advanced engine designs. This paper serves to supplement the relatively sparse documentation on transient modeling and cover the budding convergence between NPSS and Matlab based modeling toolsets. The following sections explore various design patterns to rapidly develop transient models. Each approach starts with a base model built with NPSS, and assumes the reader already has a basic understanding of how to construct a steady-state model. The second half of the paper focuses on further enhancements required to subsequently interface NPSS with Matlab codes. The first method being the simplest and most straightforward but performance constrained, and the last being the most abstract. These methods aren't mutually exclusive and the specific implementation details could vary greatly based on the designer's discretion. Basic recommendations are provided to organize model logic in a format that most easily amenable to integration with existing Matlab control toolsets.

2 Running NPSS Transiently

This paper assumes a basic understanding of the concepts required for steady-state engine modeling within NPSS. Foundational concepts are best introduced in the NPSS user's guide [1] and other introductory resources. [2] Transient simulations represent the time-varying behavior of a system by finding a series of solutions at discrete time steps over a desired time interval. At a high level, the NPSS solver drives models to a converged state by simultaneously solving a system of equations comprised of thermodynamic and user-defined constraints. For most gas-turbine engine problems, the model cannot be solved by explicitly marching from the front to the back of the engine cycle. Instead of sequentially solving for each engine component, an initial guess at the entire model solution is iteratively (implicitly) refined until all constraints are satisfied. For transient problems, NPSS solves these systems of equations largely in the same way it handles steady-state solutions. However for transient systems, certain engine states have "memory" and are driven to a time-dependent integrated state, rather than simply driving all imbalances to zero (steady-state). This extra integration step and additional user-defined steps are outlined in the following sections. At the highest level, a transient model minimally requires the following considerations beyond steady-state modeling design:

1. Configuring transient solver parameters: solutionMode, time boundary, time step, tolerances, termination criteria

2. Defining time-varying input/output variables using functions or interpolation tables, which are are subsequently connected to the solver via independents, dependents, and constraints

3. Defining transient engine components and initial conditions for transient-specific properties

4. Configuring output viewers

These steps are outlined with example code below.

2.1 Solver and Integration Methods

NPSS supports multiple integration types that are outlined in the NPSS users guide. [1, chap. 7.1] Table 1 summarizes the available methods, with a first-order differential equation for spool speed used as an example:

Table 1: NPSS Integration Methods

Method	Type	Solving for N_{t2}, given $N_{t2} - N_{t1} = \int_{t1}^{t2} \frac{T_{net}}{I}\,\mathrm{d}t$
Euler	Explicit	$N_{t2} = N_{t1} + \left.\frac{T_{net}}{I}\right\|_{t1} (t2 - t1)$
Trapezoidal	Implicit	$N_{t2} = N_{t1} + \frac{1}{2}\left(\left.\frac{T_{net}}{I}\right\|_{t1} + \left.\frac{T_{net}}{I}\right\|_{t2}\right)(t2 - t1)$
1st Order Gear	Implicit	$N_{t2} = N_{t1} + \left.\frac{\mathrm{d}N}{\mathrm{d}t}\right\|_{t2} (t2 - t1)$
2nd Order Gear	Implicit	$N_{t2} = N_{t1} + \left(\left.\frac{1}{3}\frac{\mathrm{d}N}{\mathrm{d}t}\right\|_{t1} + \left.\frac{2}{3}\frac{\mathrm{d}N}{\mathrm{d}t}\right\|_{t2}\right)(t2 - t1)$

Steady-state iteration may be required for any of these methods, and the choice between explicit and implicit types comes down to accuracy vs time. Explicit methods assume the integrand is constant over the specified time interval, therefore integration is only performed once per time step. Implicit methods perform a sub-iteration (independent from time) until the predicted state value agrees with the corrected value within a specified tolerance. NPSS also allows the user to define custom integration methods using the `Integrator` class. [1, chap. 15.2] The following code shows how a model can be initialized for either an implicit or explicit transient run in NPSS.

```
setOption( "solutionMode", "TRANSIENT" );
Transient.integrationType = "TRAPEZOIDAL"; //Default Gear 1st order
transient.setup(); //run if changing to (or from) Euler method
initializeHistory(); //run if initial conditions
        //differ from most recent transient run
```

The top level transient solver is of type `TransientExecutive`, and is named `transient` by default. This variable is analogous to the top-level steady-state "`solver`" object. The `transient` object is responsible for setting integration solver properties including the simulation start, step and stop parameters. [1, chap. 7.5] [1, chap. 15.1.8] All attributes have a default value except `transient.stopTime`, which must be supplied by the user. Over the course of a run, the `TransientExecutive` may update these attributes or even overwrite values set by the user. The user can also preemptively stop a simulation using the `quiescence()` and `terminateCondition()` functions available in the `TransientExecutive`. The code below demonstrates how to set time step settings.

```
transient { //set as a group
        timeStepMethod = "ADAPTIVE";
        baseTimeStep= 0.10;
        dxTransLimit = 0.05;
        maxTimeStep = 0.20;
        minTimeStep = 0.01;
        stopTime= 3.60;
} //or set individually: transient.stopTime = 3.6;
```

Before running transiently, the user must first ensure that each engine component with transient specific attributes is properly initialized. Common components with special transient properties include, shafts, springs, control volumes, heat exchangers, walls and thermal masses. The following code snippet shows how the special `inertia` property may be initialized for a shaft component:

```
Element Shaft HP_Shaft  {

        ShaftInputPort HPC, HPT; //list connected turbomachinery
        HPX = 100.0; //Horsepower extracted from the shaft
        Nmech = 10000; //mechanical speed
        inertia = 2; //shaft inertia  **required for transient**
        //dNqdt = Derivative of speed with respect to time (acceleration)
        real dNcqdt; //user defined variable (corrected shaft acceleration)
        void postexecute(){
                dNcqdt = HP_Shaft.dNqdt / (HPC.Fl_I.Pt / Ambient.Fl_O.Ps);
        }

}
```

In steady-state mode the solver will vary the shaft's mechanical speed to balance the input turbine torques (τ_{turb}) with the output compressor port torques (τ_{comp}). In this situation, steady state refers to:

$$\tau_{comp} - \tau_{turb} = F_{net} = ma = 0 \tag{1}$$

In transient mode there is no guarantee that compressor torques match the turbine torque. In fact, this time-dependent imbalance is often the defining transient being modeled. To model the effects of changing engine "momentum", the solver guesses are varied until the shaft speeds match the speed calculated by integrating the acceleration derived from the net engine torque.

2.2 Transient Dependents and Constraints

Transient calculations aren't limited to integrated engine state variables; in fact, any variable can be configured to change as a function of time. Time-varying inputs, dependents, and constraints must be defined and evaluated at every time step during a simulation, and therefore cannot simply be assigned a constant value before executing the `run()` command as they are in steady-state simulations. Time dependent variables can be calculated explicitly with piecewise functions for every time step or scheduled using built-in table and interpolation routines. The code snippets below show both of these methods respectively, for defining fuel flow as shown in figure 1.

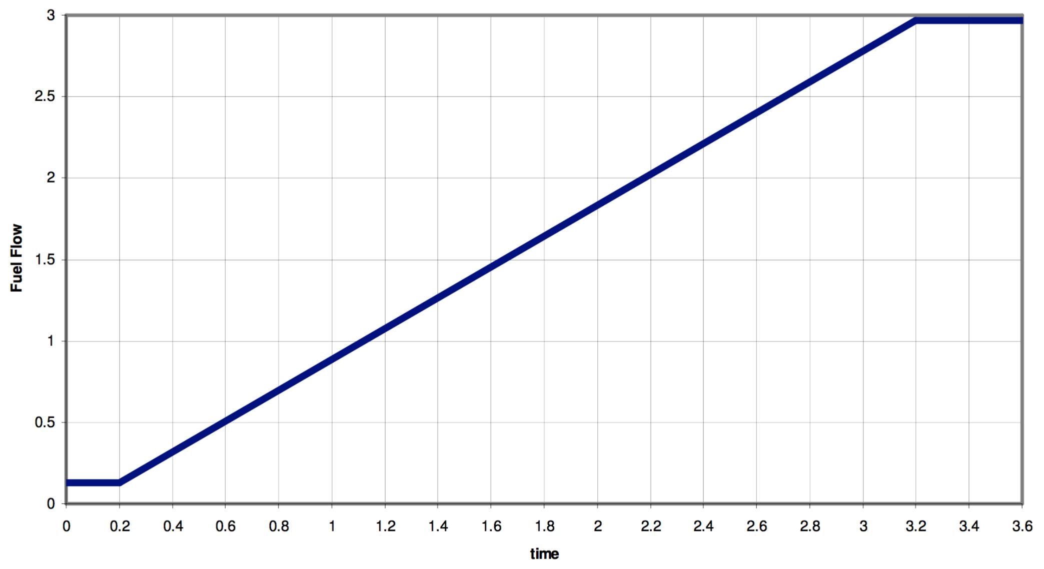


Figure 1: Example fuel ramp used to drive a transient run

```
real Ramp( real time, real tstart, real duration, real Y1, real Y2 ) {
  real fuel, slope;
  if (duration < 0) { duration = 0;}
  if (duration) {slope = (Y2 - Y1) / duration;}
  if (time <= tstart) { fuel = Y1; }
  else if (time > tstart  &&  time < tstart+duration) {
    fuel = Y1 + (slope * (time-tstart));
  }
  else { fuel = Y2; }
  return fuel;
}
trans_Condition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
```

Or the same function could be built using a table.

```
real Ramp(real time, real tstart, real duration, real Y1, real Y2) {
        real fuel;
        Table TB_time(real time) {
                time.interp = "linear";
                time.extrap = "none"; //edge val used if extrap
                time= { 0.00, tstart, tstart+duration}
                fuel = { Y1, Y1, Y2}
    }
        return TB_time(time);
}
trans_Condition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
```

The last line of each of these example functions tells NPSS how to evaluate the calculated fuel flow input. These dynamic variables can evaluated by creating a set of independent/dependent equations that are either defined in-line as shown above, or grouped together as show below. The name `trans_Condition` has no special meaning to the framework, it's an arbitrary name chosen for this example. The transient variable function is set as the right hand side (`rhs`) of a dependent equation, with the left hand side (`lhs`) set to fuel flow. A second variable must be defined as the independent variable and is varied by the solver until the dependent equation is satisfied. The independent variable and the `eq_lhs` dependent variable don't necessarily have to be the same as shown in this example. However, the independent variable must be able to influence the value of the dependent `eq_lhs`.

```
//------------------------------------
// Solver Variable Definition
//------------------------------------
Independent fuel_indep {
    varName = "Burner.Wfuel";
}
Dependent trans_Condition {
    eq_lhs = "Burner.Wfuel";
    eq_rhs = "Ramp( time, 0.2, 3, 0.1, 2.9 )";
            //Ramp(time, tstart, duration, Wstart, Wfinish)
}
//------------------------------------
// Transient Solver Setup
//------------------------------------
solver.addIndependent("fuel_indep");
solver.addDependent("trans_Condition");
```

Multiple dependent equations can be paired with a single independent variable in the form of constraints. This method is appropriate when the user intends to drive an output variable transiently, such as thrust, while ensuring no engine constraints are violated. This behavior can be beneficial for simulating engine limiters in a controller, and managing competing constraints on several variables.

```
//------------------------------------
// Constraint Definition
//------------------------------------
Dependent Tt4_Max_Limit {
    eq_lhs = "Burner.Fl_O.Tt";
    eq_rhs = "3550";
}
Dependent ThrustTarget{
    eq_lhs = "PERF.Fn";
    eq_rhs = "Ramp( time, 0.2, 5, minThrust, maxThrust )";
}
//------------------------------------
// Transient Solver Setup
//------------------------------------
```

```
    //add additional constraint(Name,Min/Max,Priority,Slope);
    ThrustTarget.addConstraint("Tt4_Max_Limit","MAX",1,1);
    //after adding any constraints, add the equation pair to the solver
    solver.addIndependent( "Burner.Wfuel" );
    solver.addDependent( "ThrustTarget" );
```

Although the constraint is defined identically to a dependent variable, it is applied to a pre-existing independent variable. The example shown above varies fuel flow to reach a specified thrust target, but only as long as a temperature constraint isn't violated. As long as this constraint is activated, fuel flow will follow the temperature limit and ignore the thrust target. Since numerous constraints can be applied to a dependent variable, optional arguments can be supplied to specify if a variable is a minimum or maximum limit. The optional third argument of the `addConstraint` method, labeled as 'priority' in the code snippet above, determines which limit to ignore if competing min and max limits are violated. In rare cases, the change in error is negatively correlated to changes in the independent variable, leading to the solver to get locked into alternating limits. One such case would be varying fuel flow to reach a target thrust, with an additional constraint on minimum and maximum low pressure compressor R-lines. Flipping the 'MIN' or 'MAX' and the sign of the 'slope' arguments can be used to resolve this numerical instability.

A further abstracted method of setting user defined variables can be implemented using the NPSS supplied `solversequence()` method. This allows users to simply append a function to the beginning or end of every time-step calculation loop and evaluate any variable to a specified value.

```
//-------------------------------------
// Solver Variable Definition
//-------------------------------------
Independent fuel_indep {
    varName = "Burner.Wfuel";
}
Dependent trans_Condition {
    eq_lhs = "Burner.Wfuel";
    eq_rhs = "RampOutput";  //needs to be recalculated every time step
}
//-------------------------------------
// Transient Solver Setup
//-------------------------------------
solver.addIndependent("fuel_indep");
solver.addDependent("trans_Condition");

solver.presolverSequence("fuelCalc"); //backwards compatible alias
//solver.preExecutionSequence("fuelCalc"); //equivalent statement
//-------------------------------------
// fuelCalc function definition
//-------------------------------------
void fuelCalc(){ //this is run before every time step
        real Wf1 = 0.1;
        real Wf2 = 2.9;
        real RampOutput = Ramp( time, 0.2, 3.0, Wf1, Wf2 ); // eq_rhs
}
```

This code block performs a functionally equivalent operation to the first example snippet of this section. Although the computational differences are minor, this provides the user with another option for organizing logic. From an organizational standpoint, it may be desirable to fully separate concerns by keeping certain logic separate from the model itself. Example code demonstrating all of these methods can be found in appendix A.1.

2.3 Transient Output

In order to capture the engine state after convergence of every time step, standard viewers or custom functions can be invoked for transient cases using the `solver.postExecutionSequence` method. This method takes an array of strings that correspond to viewer objects or function names. If a PageViewer or DataViewer is called, it will automatically invoke the `display()` method of these objects, if a CaseViewer is called, only the `update()` method will be invoked. If an implicit integration method is used, the viewer will only update after each sub-iteration is fully converged.

```
//------------------------------------
// Transient Row Viewer
//------------------------------------
OutFileStream transientStream { filename = "tout"; }
DataViewer CaseRowViewer transientTrace {
        titleBody = " ";
        titleVars = { };
        variableList = {
                "time: ??.???? ",
                "Amb0.W : ???.?? = Air Flow",
                "HPC.Fl_O.Pt : ????.?? = Pt3",
                "HPC.Fl_O.Tt : ????.?? = Tt3",
                "LPshaft.Nmech : ??????.?? = LPspeed",
        };
        pageWidth = 132;
        pageHeight = 0.;
        outStreamHandle = "transientStream";
}
//------------------------------------
// Call to Viewer
//------------------------------------
solver.postExecutionSequence = { "transientTrace" }; //CaseViewer function
  //^executes after solver convergence at each time step
run();
transientTrace.display(); //write viewer variables to it's output file
```

Additional viewer reference material can be found in [1, chap. 7.2.2, 12, 15.3.1]

2.4 Transient Run Files

It's generally recommended to organize each of these steps in separate files or folders to better manage complexity. Each aspect is then generally orchestrated in a run file containing numerous simulations chained together. After on-design steady-state engine sizing occurs, engine state boundaries can be established by running multiple off-design power settings until key engine constraints become active. Generating large tables of equilibrated or 'trimmed' engine points throughout the flight envelopes ensures that transients can be run from any starting condition without starting imbalances. In the following example, a steady-state case is first run to initialize the state of the engine and to determine fuel flow bounds on a transient input driver based on user-provided thrust targets. A transient case is then run from t=0 to t=0.6, paused, then resumed to t=3.6. Finally, a new transient run clears the previously integrated engine states and runs a fresh case from t=0 to t=2.4.

```
//------------------------------------
// Run Two Steady State Cases
//------------------------------------
setOption( "solutionMode", "STEADY_STATE" );
PERF.FnTarget = MinThrustTarget;
run();               // Run once
Wf1 = Burner.Wfuel // Save calculated fuel flow
PERF.FnTarget = MaxThrustTarget;
run();               // Run again
Wf2 = Burner.Wfuel // Save calculated fuel flow

//------------------------------------
//Run Two Transient Cases
//------------------------------------
trans_Condition.eq_rhs = "Ramp( time, 0.2, 3.0, Wf1, Wf2 )";
setOption( "solutionMode", "TRANSIENT" );
initializeHistory();       // initialize variable history
transient.baseTimeStep = 0.10;
transient.stopTime = 0.60;
run();                                                  // t = 0 -> 0.6
transient.stopTime = 3.60; // extend same transient run
run();                                                  // t = 0.6 -> 3.6

transient.clear();
trans_Condition.eq_rhs = "Ramp( time, 0.2, 5.0, Wf1, Wf2 )";
solver.forceNewJacobian = TRUE;
transient.setup();
time = 0.;
transient.stopTime = 2.40;
run();                // new transient case, t = 0 -> 2.4
```

2.5 Advanced Dynamics Modeling

Users can find additional transient solver options in the NPSS user's guide [1, chap. 18] for simulating more advanced engine dynamics such as custom first order-lags, adaptive time-stepping, custom integrators or predictor calculations.

While possible to implement in NPSS, many advanced modeling processes can benefit from specialized tool-sets in other development environments such as Matlab/Simulink. The following sections outline strategies for interfacing NPSS with Matlab to take advantage of Matlab's large ecosystem of tools.

3 Passing Data Between NPSS and Matlab

There are several different ways to pass information between simulations running in NPSS and Matlab/Simulink. There is no silver bullet approach, each solution fits a different use case.

3.1 Raw File I/O

The most simplistic and straightforward method for passing information is automated input/output (I/O) file passing and batched system calls. File I/O is undesirable from a performance aspect, but can be acceptable if the total number of round-trips between NPSS and Matlab are minimized. This method is suitable for upfront engine initialization calculations, and functions where programming simplicity is more important than computational efficiency.

A single round-trip generally involves the following steps:

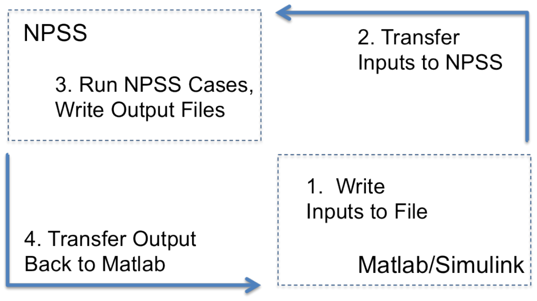

NPSS
3. Run NPSS Cases, Write Output Files
2. Transfer Inputs to NPSS
1. Write Inputs to File
Matlab/Simulink
4. Transfer Output Back to Matlab

Figure 2: Example DataFlow for File I/O

1. Specify run cases, programatically writing the necessary batch and input files $[Matlab]$

2. Transfer cases to the NPSS model directory $[Matlab \rightarrow NPSS]$

3. Execute NPSS Run Cases, saving output variables of interest in a Matlab readable syntax $[NPSS]$

4. Transfer output files back to Matlab where they can be loaded into the working directory $[NPSS \rightarrow Matlab]$

Matlab can dynamically write input/run scripts, which are subsequently copied to the NPSS directory and executed from a system call. The following Matlab snippet generates an input file

```
inputFile = 'NPSS_setpoint.input'; %name of input file to be written
inputPath = '..\path2folder\'; %path to write to

setpointVector = [10000.0, 10952.381, 11904.762, 12857.143, 13809.5238,
                  14761.9048, 15714.2857, 16666.6667, 17619.0476, 18571.4286,
                  19523.8095, 20476.1905, 21428.5714, 22380.9524, 23333.3333,
                  24285.7143, 25238.0952, 26190.4762, 26500.0000];

minS = min(setpointVector);
maxS = max(setpointVector);
fileID = fopen([inputPath inputFile],'w');
fprintf(fileID,'%% setpoints between %10.4f and %10.4f \n',minS,maxS);
dlmwrite([inputPath inputFile],'real Fn_targets[]={','delimiter','','-append')
dlmwrite([inputPath inputFile],setpointVector,'-append')
dlmwrite([inputPath inputFile],'};','-append','delimiter','')
fclose(fileID);
```

which will create the following file, called `NPSS_setpoint.input`

```
% setpoints between 10000.0000 and 26500.0000
real Fn_targets[]={
          10000,10952,11905,12857,13810,
          14762,15714,16667,17619,18571,
          19524,20476,21429,22381,23333,
          24286,25238,26190,26500
};
```

This method works best when adhering to a few heuristics. Firstly, organize an engine repository into sub-folders. Although it's possible to contain everything within a single file or root folder, separating code improves readability and simplifies automated file operations when they can be limited to a smaller subset of files. Table 2 shows one such way to organize files.

Folder Name	Folder Function
Maps	Off-design performance maps and large table data
Inputs	Input files generated from external sources and subsequently fed into NPSS
Outputs	Outputs files generated by NPSS
Run	Files used to orchestrate run cases
View	All viewer functions used to generate outputs
Src	All other model and function source files

Table 2: An opinionated folder scaffolding for NPSS engine repositories

Breaking down folders, files, and functions into smaller pieces also improves code-reuse, which is generally desired for any code-base. This allows many general use functions to act as a common core shared between multiple engine architectures.

Separating logic in this manner allows the programmer to better follow the third recommended heuristic of passing the minimum amount of information necessary between codes. If inputs are isolated in their own file it's much easer for external codes to pass in files containing only the necessary data, while keeping any logic (and therefore potential side-effects) separated.

External codes sometimes require more control than just specifying NPSS input data. If run cases must be dynamically configured during execution, NPSS provides command line arguments [1, chap. 2.1] including preprocessor variables that can be used to invoke optional code. Example code demonstrating these methods can be found in appendix A.1.

It is possible to drive a closed-loop NPSS transient from an external code by passing files between programs after every single time step. However, due to it's inefficiency it is highly advisable to revert to memory-wrapped execution when driving a transient from Matlab.

3.2 Compiled S-function (Memory-wrapped simulations)

A dynamically-linked library (dll) is included with NPSS and can be used within a level 2 S-function in Simulink. The NPSSSfunction.dll requires a full NPSS v1.6.X release distribution, as well a 32-bit(only) Matlab distribution from R2007b through R2010a. Future versions of Matlab disable the operability of dll's, in favor of strictly enforcing the use of custom Matlab Executable MEX files. The `dll` is not compatible with the Mac operating system, or any 64-bit version of Matlab. As of April 2014, NPSS version 2.7.1 VC10 has been recompiled from source as a `MEX` file. This version of the S-function is only compatible between programs supporting the same resolution. So the 64-bit `MEX` is only compatible with with 64-bit NPSS, and has only been tested on Matlab 2012b, and Matlab 2014b. A separately compiled `MEX` file is required for 32-bit Matlab and NPSS distributions. These new wrappers are not backwards compatible to NPSS version 1.6.X The version compatibility is shown in table 3.

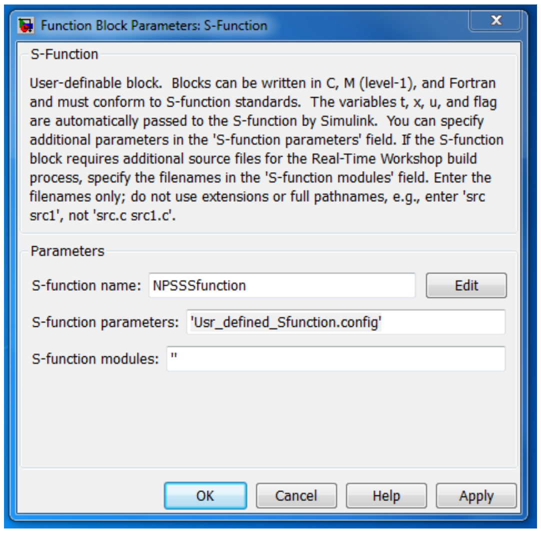
Matlab Version	NPSS v1.6.X	NPSS v2.7.X 32-bit	NPSS v2.7.X 64-bit
32-bit	DLL (R2007b-R2010a)	MEXw32 (R2010a-R2014b)	Not Supported
64-bit	Not Supported	Not Supported	MEXw64 (R2012b-R2014b)

Table 3: S-function compatibility chart

The `MEX` version requires a few extra steps as outlined in the user‘s guide provided in each NPSS distribution. [3] This involves invoking Matlab from the command line, and adding the NPSS distribution bin file to both the computer `PATH` and Matlab path.

The S-function encapsulates NPSS within a Simulink block and can be integrated into a time-varying system using the same conventions as conventional Simulink component blocks. Any standard output `stdout` from NPSS is redirected to the Matlab command window during execution. It's recommended to print any user-relevant error and dialog messages to `stdout` or pipe them to a log file to aid in debugging.

The S-function requires two arguments, as shown in fig. 3. The S-function name refers to the name of the `dll` and the S-function parameter must point to a user-defined configuration file enclosed in single quotes (‘’).



User-definable block. Blocks can be written in C, M (level-1), and Fortran and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for the Real-Time Workshop build process, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

S-function name: NPSSfunction

S-function parameters: 'Usr_defined_Sfunction.config'

S-function modules: ''

Figure 3: S Function Parameter Dialog Box

This configuration bash file includes all necessary paths, global environment variables, and specifies which engine variables to expose to Simulink. In essence, this file defines the which engine variables can be controlled at the boundary of the NPSS S-function black box.

```
commandLine = "-I. -I C:/path/NPSS_dir/bin";
commandLine += " -I C:/path/NPSS_dir/InterpIncludes";
commandLine += " -I C:/path/NPSS_dir/InterpComponents";
commandLine += " -I C:/path/NPSS_dir/DLMComponents/nt";
commandLine += " -I C:/path/NPSS_dir/MetaData";
commandLine += " -I C:/path2/NPSSengine/run";
commandLine += " -I C:/path2/NPSSengine/src";
commandLine += " -I C:/path2/NPSSengine/maps";
commandLine += " C:/path2/NPSSengine/run/engine.run -DFlag1 -DFlag2";

SimulinkInPortMapper inPort1 {
   vars = {  "Burner.Wfuel" }
}

timeStep = 0.02;

SimulinkOutPortMapper outPort1 {
        vars = {
        "LP_Shaft.Nmech", "HP_Shaft.Nmech", "FS_3.Ps", "FS_4.Pt", "FS_9.Tt"
        }
}

SimulinkOutPortMapper outPort2 {
        vars = {
    "Perf.myEPR", "Perf.myFn", "Perf.Wfuel", "HPC.SMN", "HPC.SMW"
    }
}
```

This particular example includes multiple folders on the path for the NPSS distribution itself, as well as paths within the engine folder. The final command line string runs a batch run file and two optional preprocessor flags called `DFlag1` and `DFlag2`. If not explicitly added to the config file, the `engine.run` file must also set the `NPSS_TOP`, `NPSS_DEV_TOP`, `NPSS_CONFIG`, and `NPSS_TOP` environmental variables.

In this example, the config file will create an S-function block with 1 input port, and 2 output ports each containing 5 muxed signals. Signals fed into this input port will continuously update the value of the NPSS defined `Burner.Wfuel` variable during the transient execution. Similarly, all the specified outputs will be fed as output signals of the Simulink block. Multiple variables defined within a single output port are muxed into a single signal bus.

In-depth examples of the S-function can be found in the source code for the Tool for Turbine Engine Closed-loop Transient Analysis program (TTECTrA). [4] The S-function was used extensively for this project under NASA's Advanced Air Transportation Technologies (AATT) Project. The tool is intended to automatically calculate a first-cut approximation of a closed loop controller for any given transient engine model. The program runs multiple internally wrapped NPSS transient scenarios with fuel flow dictated by the Simulink model and uses the returned output to select appropriate controller parameters. Fig 4 shows the NPSS engine response to a fuel ramp controlled by the outer Simulink program.

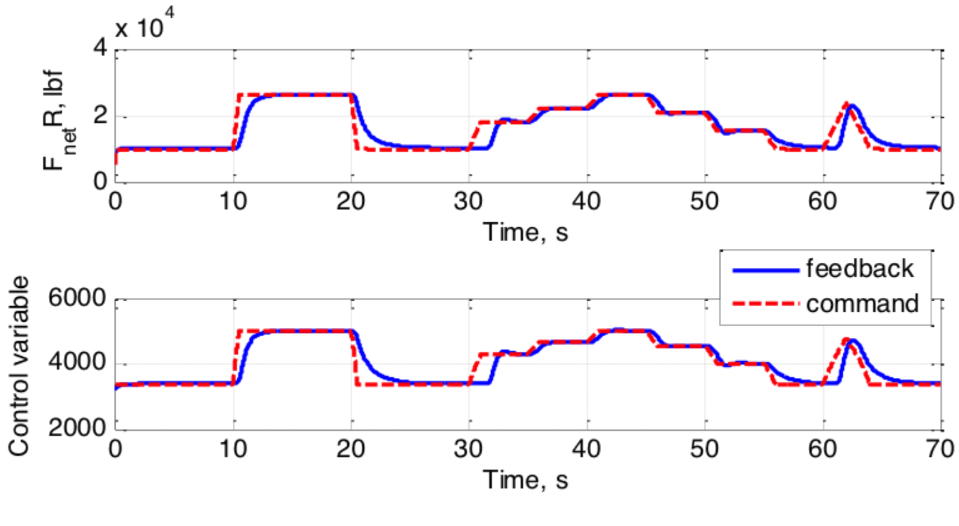


Figure 4: NPSS transient engine response to a Simulink commanded fuel flow

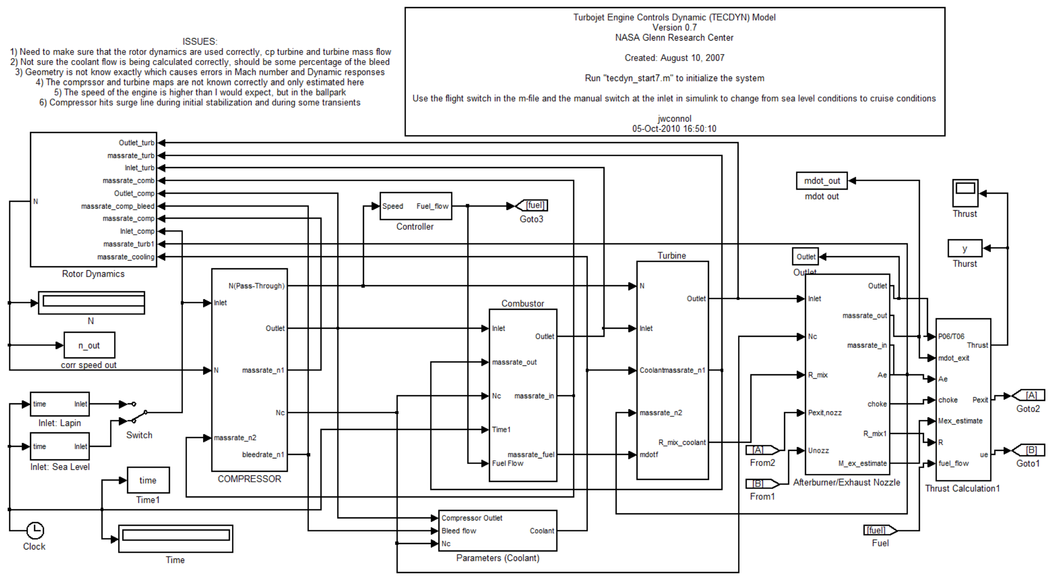
3.3 Source-to-Source Translation

The most abstract method of generating a transient model from NPSS is best described as “Source-to-Source Translation” (SST), which is the process of converting one high level model to another. This pattern is appropriate when developing a transient engine model that may be completely separate, but originally derived, from NPSS.

Alternate thermodynamic cycle analysis tools to NPSS such as TMATS [5] [6], and PyCycle [7] contain the same basic engine components as NPSS. Therefore, assuming an engine model is comprised entirely of thermodynamic components that exist in both tools, it’s possible to automatically convert an engine from one code to another. A source-to-source compiler capable of parsing a NPSS model code could automatically instantiate an equivalent version in an alternate analysis tool.

SST could significantly reduce development time and tedious error-prone manual model translation. Rather than forcing all developers to use the same code-base, this method allows engines to be modeled in each environment’s native language. Following the “write once, run everywhere” paradigm, this process can significantly reduce development time, which is vital for maintaining code flexibility and adaptability across languages.

This methodology is demonstrated with the Supersonic Component Engine Model (SCEM), which is a dynamic engine model developed to better understand the coupling between propulsion and aero-servo-elastic modes of long slender supersonic vehicles. [8] [9] The model is designed to extend a steady-state NPSS model and better capture high bandwidth dynamics associated with large component volumes. When the porting was first attempted by hand, development proved to be challenging largely due to the number of engine components involved. Many of the challenges were associated with human-errors from manually replicating the model in a new language.



ISSUES:
1) Need to make sure that the rotor dynamics are used correctly, cp turbine and turbine mass flow
2) Not sure the coolant flow is being calculated correctly, should be some percentage of the bleed
3) Geometry is not know exactly which causes errors in Mach number and Dynamic responses
4) The comprssor and turbine maps are not known correctly and only estimated here
5) The speed of the engine is higher than I would expect, but in the ballpark
6) Compressor hits surge line during initial stabilization and during some transients

Turbojet Engine Controls Dynamic (TECDYN) Model
Version 0.7
NASA Glenn Research Center

Created: August 10, 2007

Run "tecdyn_start7.m" to initialize the system

Use the flight switch in the m-file and the manual switch at the inlet in simulink to change from sea level conditions to cruise conditions

jwconnol
05-Oct-2010 16:50:10

Figure 5: A legacy version of SCEM built manually

Similar to NPSS, a library of generic SCEM engine elements was composed to handle many combinations and engine architectures. Although the SCEM solver is fundamentally different than the solver used by NPSS, both models were comprised of the same basic thermodynamic elements, linked together in the same order. In order to automate the generation of an analogous NPSS model, both tools followed the same object-oriented nature, rather than the procedural paradigm generally used in Matlab. These custom Matlab classes use object-oriented features that first became available in the R2008a revision of Matlab [10] By aligning the framework paradigms between NPSS and SCEM, it became possible to create a parallel Matlab "classes" for each standard element contained within NPSS.

These Matlab objects parse common attributes within NPSS components, dynamically configuring the analogous Simulink block to match. This automates the conversion of any NPSS model into an analogous Matlab model, greatly reducing the opportunity for human error and simplifying the process of extending the model.

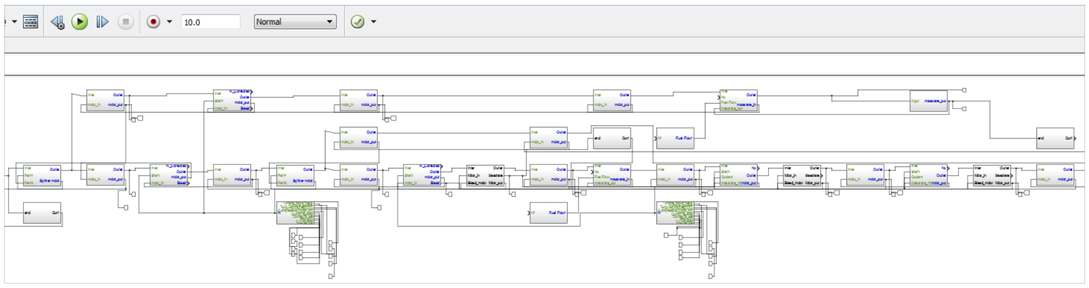


Figure 6: A version of SCEM built automatically via SST

Since NPSS engine models can vary widely in implementation details, a SST is limited to recognizing standard elements, solver variables, and connections. A NPSS developer can also include special attributes within standard components that would be ignored by NPSS, but parsed by the SST transcompiler to provide additional context to the Matlab equivalent engine. The following example code shows a `Matlab_position` variable that could be ignored by NPSS, but parsed by a Matlab counterpart.

```
Element Shaft HP_Shaft  {

        ShaftInputPort HPC, HPT; //list connected turbomachinery
        HPX = 100.0; //Horsepower extracted from the shaft
        Nmech = 10000; //mechanical speed
        inertia = 2; //shaft inertia  **required for transient**

        real Matlab_position = 7; //ignored by NPSS,
                        //but potentially meaningful to a SST parser
}
```

4 Final Remarks

The aforementioned approaches are by no means an exhaustive list, but provide new users with a starting point for developing complex transient engine models where development can be distributed across NPSS and Matlab.

The varying merits, complexities, and pitfalls of each approach indicate that there is no silver-bullet approach to unifying these disparate development tools. The best method is dependent on the problem, and even a hybrid combination of multiple methods can be implemented. As an example, a highly complex NPSS model could be partially transcompiled into a native Matlab model, with any custom components encapsulated within their own isolated S-functions. This same hypothetical model could also perform a one-time initialization involving data passed using basic file I/O. Beyond developing the API, it is also up to the developers of each tool to agree upon the level of control each program is responsible for. Often times tools have overlapping capabilities, so the developers must take extra care to avoid conflicting or competing logic between each tool-set.

Developing standard modeling and message passing practices and approaches are necessary to unify code-bases and development efforts. Investing upfront effort to reduce long-term development time and complexity can ultimately lead to better overall designs.

References

1. The Ohio Aerospace Institute, on behalf of the NPSS Consortium. *NPSS User Guide*, 2010.

2. Scott Jones. An Introduction to Thermodynamic Performance Analysis of Aircraft Gas Turbine Engine Cycles Using the Numerical Propulsion System Simulation Code. Technical Memorandum (NASA/TM2007-214690), 2007.

3. Southwest Research Institute, on behalf of the NPSS Consortium. *NPSS Level 2 S-function Interface to Simulink*, 2015.

4. J. T. Csank and A. M. Zinnecker. *Tool for Turbine Engine Closed-loop Transient Analysis (TTECTrA) Users Guide*, 2014.

5. J. Chapman, T. Lavelle, R. May, J. Litt, and T.H Guo. Propulsion System Simulation Using the Toolbox for the Modeling and Analysis of Thermodynamic Systems (T-MATS). Technical Memorandum (NASA/TM2014-218410), NASA, 2013.

6. J. Chapman, T. Lavelle, J. Litt, and T.H Guo. A Process for the Creation of T-MATS Propulsion System Models from NPSS Data. Technical memorandum, NASA, 2014.

7. J. Gray, E. Hendricks, J. Chin, and T. Hearn. Equilibrium Chemistry Thermodynamics With Analytic Derivatives in OpenMDAO. Technical Memorandum (NASA/TM2016-214690), 2016.

8. G. Kopasakis, J. Connolly, D. Paxson, and P. Ma. Volume Dynamics Propulsion System Modeling for Supersonics Vehicle Research. Technical Memorandum (NASA/TM-2008-215172), NASA, 2008.

9. J. Connolly, G. Kopasakis, and K. Lemon. Turbofan Volume Dynamic Model for Investigations of Aero-Propulso-Servo-Elastic Effects in a Supersonic Commercial Transport. Technical Memorandum (NASA/TM–2010-216069), NASA, 2010.

10. The MathWorks Inc. *MATLAB Object-Oriented Programming*, 2013.

Appendix A

Source Code

A.1 TTECTrA

The entire source code for TTECTrA can be found on github at:
`<https://github.com/nasa/TTECTrA>`
This tool makes extensive use of both file passing and the S-function.