Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1998-12

Development of a graphical numerical simulation for thermoacoustic research

Purdy, Eric W.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/26474

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NPS ARCHIVE
1998.12
PURDY, E.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1998	Master's Thesis

4. TITLE AND SUBTITLE
DEVELOPMENT OF A GRAPHICAL NUMERICAL SIMULATION FOR THERMOACOUSTIC RESEARCH

5. FUNDING NUMBERS

6. AUTHOR(S)
Purdy, Eric W.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis is written to document the design and use of an object-oriented, numerical simulation of thermoacoustic devices. The resulting expert system code entitled "Design Simulation for Thermoacoustic Research", or DSTAR, allows a unique new approach for the rapid design and simulation of thermoacoustic devices. Past approaches to thermoacoustic modeling have involved the use of "disposable" algorithms coded to model one specific device. Conversely, DSTAR uses a Windows™ compliant graphical user interface to construct any given thermoacoustic model at runtime. As a result, the models can be developed quickly and without any revision of the computer code. The approach to simulation involves the solution of a one-dimensional acoustic wave equation concurrently with an energy flow equation from one end of the user-specified device geometry to the other in addition to various lumped acoustical elements. The resulting steady-state solution is displayed in both graphical and textual output. Considerable effort was given to preserving the flexibility and breadth of the possible simulations, in addition to allowing easy modification of the source code for new thermoacoustic components. To demonstrate the utility of the code, a thermoacoustic prime mover was modeled and then optimized for better performance.

14. SUBJECT TERMS	15. NUMBER OF PAGES
thermoacoustic simulation, numerical model, object-oriented	97
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

DEVELOPMENT OF A GRAPHICAL NUMERICAL SIMULATION FOR THERMOACOUSTIC RESEARCH

Eric W. Purdy
Lieutenant, United States Navy
B.A., University of California, Davis, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
December 1998

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

ABSTRACT

This thesis is written to document the design and use of an object-oriented, numerical simulation of thermoacoustic devices. The resulting expert system code entitled "Design Simulation for Thermoacoustic Research", or DSTAR, allows a unique new approach for the rapid design and simulation of thermoacoustic devices. Past approaches to thermoacoustic modeling have involved the use of "disposable" algorithms coded to model one specific device. DSTAR uses a Windows™ compliant graphical user interface to construct any given thermoacoustic model at runtime. As a result, the models can be developed quickly and without any revision of the computer code. The approach to simulation involves the solution of a one-dimensional acoustic wave equation simultaneously with an energy flow equation from one end of the user-specified device geometry to the other in addition to various lumped acoustical elements. The resulting steady-state solution is displayed in both graphical and textual output. Considerable effort was given to preserving the flexibility and breadth of the possible simulations, in addition to allowing easy modification of the source code for new thermoacoustic components. To demonstrate the utility of the code, a thermoacoustic prime mover was modeled and then optimized for better performance.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. THERMOACOUSTIC ENGINE MODELS 1
B. OBJECTIVE 2
C. THESIS ORGANIZATION 2

II. THERMOACOUSTIC HEAT ENGINES 5

A. HEAT ENGINE EFFICIENCY AND THE CARNOT CYCLE 5
B. ACOUSTIC HEAT ENGINES 8
1. Requirements of Operation 8
2. The Acoustic Heat Engine Cycle 10
3. Heat and Work 11
4. The Roit Wave and Energy Flow Equations 13

III. NUMERICAL COMPUTATIONAL METHODS 15

A. FIFTH ORDER ADAPTIVE STEPSIZE RUNGE-KUTTA 15
1. Runge-Kutta Methodology 15
2. Fehlberg RK45 17
B. NEWTON-RHAPHSON METHOD FOR MULTI-DIMENSIONAL ROOT FINDING 18
C. LOWER-UPPER TRIANGULAR MATRIX DECOMPOSITION 21
1. Solving Linear Systems Using LU Decomposition 21
2. The LU Decomposition Algorithm 23

IV. PROGRAM ORGANIZATION AND OPERATION 25

A. DSTAR AND OBJECT ORIENTED PROGRAMMING 25
B. THE DSTAR OBJECT MODEL 26
1. The Core Classes 27
2. Other Classes 30
3. Class Organization 30
C. COMPUTING AN INITIAL VALUE PROBLEM SOLUTION 32
D. COMPUTING A BOUNDARY VALUE PROBLEM SOLUTION 34

V. DSTAR GRAPHICAL USER INTERFACE 39

A. MAIN PROGRAM WINDOW 39
1. Menu Commands 41
2. The Toolbar 42
3. The Multi-Function Tabbed View 43
B. THE OUTPUT WINDOW 50

VI. DSTAR PRACTICAL EXAMPLE: AN ENHANCED HOFLER TUBE 55

VII. CONCLUSION 63

APPENDIX A: EXTENDING DSTAR 65

A. ADDING PROGRAM FUNCTIONALITY 65
B. AN EXAMPLE THERMOACOUSTIC CLASS 68
C. ADDING THE NEW CLASS TO THE USER INTERFACE 72

APPENDIX B: SYMBOLS AND EQUATIONS 75

LIST OF SYMBOLS 75
THERMOACOUSTIC EQUATIONS FOR IDEAL GASES 75
LUMPED ELEMENTS 80
LIST OF REFERENCES 83
INITIAL DISTRIBUTION LIST 85

ACKNOWLEDGEMENT

The author would first like to thank the US Navy and Naval Postgraduate School for giving him the opportunity of a lifetime. Few other institutions would have the courage and patience to give someone with relatively little technical knowledge the chance to pursue graduate studies in physics.

Secondly, I would like to thank Tom Hofler for both introducing me to the world of thermoacoustics and allowing me great latitude in the development of this project. His patience and amicable nature made working with him a truly enjoyable experience.

Lastly, and most importantly, I would like to thank my wife Belinda for putting up with the many hours I spent on the project. Without her love and support, this project could not have happened.

I. INTRODUCTION

Thermoacoustic devices come in the form of prime movers and heat pumps. A prime mover is a heat engine in which heat-flow from a high temperature reservoir to a low-temperature sink generates sound. In a heat pump, or refrigerator, the opposite occurs. Here the addition of acoustic power moves heat from a low temperature reservoir to a high temperature sink. The cycles of these acoustic heat engines can be accomplished with few or no moving parts. As such, these devices are inherently simple and reliable.

A. THERMOACOUSTIC ENGINE MODELS

The history of thermoacoustics dates back to the eighteenth century but the analysis and design of thermoacoustic heat engines is a relatively new science. Since the establishment of the theoretical foundation for thermoacoustics by Nikolaus Rott and his coworkers, physicists have had the means to model the acoustic heat engine. Computer codes have been developed to aid in the design and simulation of these devices. However, these codes have been "disposable" (i.e. written to model one specific device) and cumbersome to use. Disposable codes require extensive rewriting each time a given device configuration changes. This re-coding is both time consuming and onerous. As such, modeling of thermoacoustic heat engines has been tedious and has detracted from the overall goal of producing efficient, well designed devices.

B. OBJECTIVE

The goal of this thesis project is to produce a new expert system code to model thermoacoustic devices. The code should be easy to use as well as proving the flexibility to model many types of thermoacoustic engines without rewriting the code. It should be designed so that incorporations of new thermoacoustic algorithms are accomplished with relative ease. Lastly, speed of computation should be considered in accomplishing the above goals.

By its nature, this project is very multi-disciplinary. Although the simulation is based in physics, the implementation required aspects of numerical analysis as well as computer science.

C. THESIS ORGANIZATION

Chapter II begins by describing qualitatively the foundations of thermoacoustic theory. A basic understanding of the theory will give the reader added insight into the nature of the simulation and its operation.

Chapter III describes the primary mathematical techniques used to solve the differential equations describing a thermoacoustic device. These techniques involve advanced numerical integration methods, matrix algebra, and multi-dimensional root finding algorithms.

Chapter IV details the computational methodology used to encapsulate the numerical methods and device geometry into a coherent scheme for thermoacoustic modeling. It is the object-oriented nature of the code, described in this chapter, which results in a simulation that is both flexible and easy to use.

Chapter V provides a guide for the simulation user interface. Here, the use of the interface is described rather than the code required to create it.

Chapter VI shows a practical example of the simulation by modeling a previously built thermoacoustic prime mover and then modifying its design to improve efficiency.

Lastly, Chapter VII offers some conclusions about the resultant code and some suggestions for future work.

II. THERMOACOUSTIC HEAT ENGINES

Thermoacoustic heat engines use acoustic waves in a resonant vessel to pump heat from a lower temperature reservoir to one of higher temperature, or conversely, use an externally imposed temperature gradient to produce acoustic waves. Much of our analytical understanding thermoacoustic devices stem from the work of Nikolaus Rott and his coworkers. Rott published a series of papers beginning in 1970 which laid the theoretical foundation for the work that continues today. Rott's theory is based on a linearization of the Navier-Stokes, continuity, and energy equations, with sinusoidal oscillations of all variables [Ref. 1:p. 23]. A theoretical description as well as the complete analytic treatment are provided by Swift [Ref. 2] and Wheatley et. al. [Ref. 3 & 4] and are paraphrased herein.

A. HEAT ENGINE EFFICIENCY AND THE CARNOT CYCLE

All heat engines can operate in one of two fundamental modes as shown in Figure 2.1. The first mode, that of a prime mover, involves the flow of heat from a higher temperature reservoir to one of lower temperature. In the process of this heat-flow, some of the heat is converted to usable work. For the alternate heat pump mode of operation, the opposite occurs. In a heat pump, the addition of work pumps heat from a low to a high temperature reservoir. Ideally, an engine can be functionally reversible but practically most heat engines are designed to operate in only one mode.

The Carnot cycle represents the ideal reversible thermoacoustic heat engine cycle. This four-step cycle, as illustrated in Figure 2.2, involves two adiabatic steps and two isothermal steps. During the adiabatic compressions

and expansions, no heat-flow occurs and entropy remains fixed. As a result, any work flux that occurs will cause a corresponding change in the local temperature of the medium. Conversely, during the isothermal processes, the temperature is fixed, and flows of entropy, work, and heat occur. For the Carnot cycle, the change in entropy along one isotherm exactly balances that

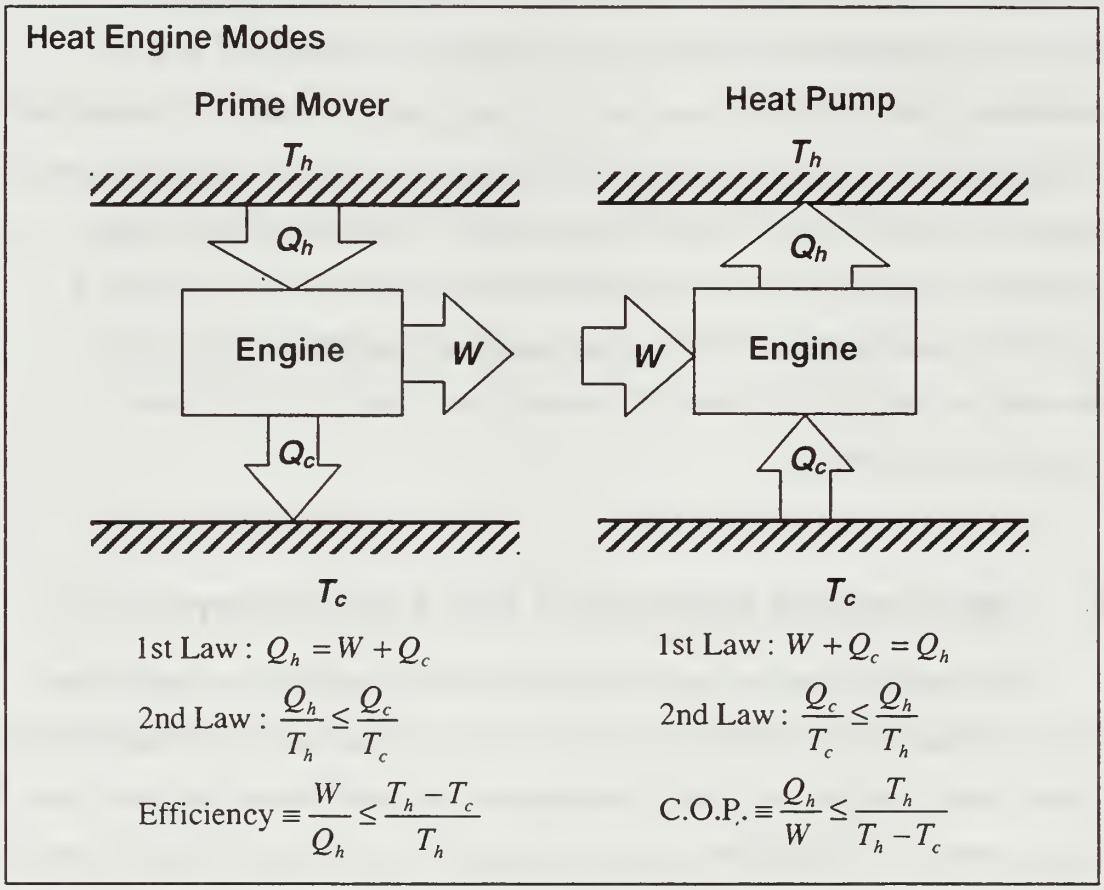


Figure 2.1. Heat engines can operate in two fundamental modes. As a prime mover, the heat engine converts some of the heat flowing from a hot temperature to a cold temperature into work. As a heat pump, the flows of heat and work are reversed. The first law of thermodynamics details the fundamental relationship that exists between the heat and work. The second law requires that the entropy created per cycle must be positive or zero. The resultant prime mover efficiency and heat pump coefficient of performance (C.O.P.) are therefore bounded as shown. Note, the C.O.P. for a refrigerator is defined as Q_c/W. See Appendix B for variable definitions. [Ref. 3:p. 4]

along the other, hence there is no net change in entropy over the entire cycle. This ideal process, carried out in near equilibrium conditions, demonstrates the upper bound on heat engine efficiency that is imposed by the first and second laws of thermodynamics. [Ref. 3:p. 2-4]

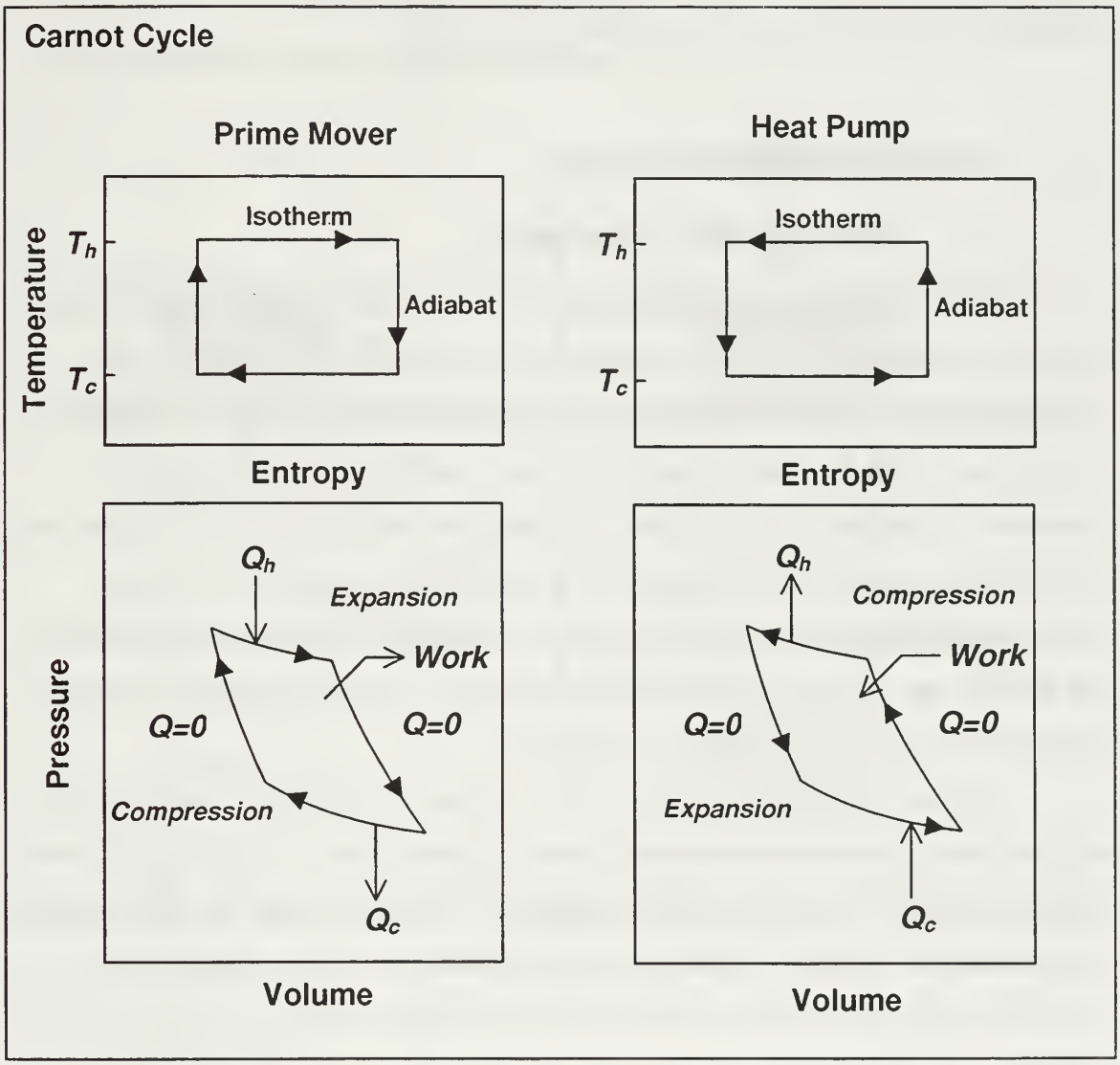


Figure 2.2. The Carnot cycle and associated temperature-entropy and pressure-volume diagrams. This near-equilibrium cycle represents the most efficient manner in which a heat engine may operate. The area enclosed by each of the pressure-volume diagrams equals the net work done by or on the engine in the complete cycle. [Ref. 3:p. 6]

In practice, heat engines do not operate in near-equilibrium cycles. Instead, inherent irreversibilities due to temperature and pressure gradients generate entropy and as a result, decrease efficiency. Furthermore, a cycle that operates at maximum efficiency (i.e. near equilibrium) will have, in general, very low power output. [Ref. 3: pp. 2-4]

B. ACOUSTIC HEAT ENGINES

1. Requirements of Operation

Some irreversible processes that decrease heat engine efficiency are, by nature, central to an acoustic heat engine's operation. An acoustic heat engine is one in which the reciprocating cycles are accomplished without the use of moving parts, but instead by acoustic oscillations. These engines require the irreversibility of heat conduction across a temperature gradient to provide the necessary phasing of the thermodynamic cycles. As such, acoustic heat engines are intrinsically irreversible thermodynamically, but, as we will see, functionally reversible. Figure 2.3 shows the basic structure for the two types of acoustic heat engines.

The requisite phasing of an acoustic heat engine is accomplished by the introduction of a second thermodynamic medium into the heat engine's cycle. The presence of a second medium alters thermodynamic phase relationships that exist within the oscillating working fluid alone. This second medium usually takes the form of a set of thinly spaced plates known as the stack. This stack will have, in general, a thermal gradient along the direction of acoustic oscillations. As acoustic oscillations occur in the engine, gas parcels move back and forth across the surface area provided by the plates. Since the pressure oscillations of the parcel are nearly adiabatic, compressions and

expansions will induce accompanying changes in temperature. In addition to the temperature changes brought about by the acoustic oscillations, a gas parcel will also experience changes in temperature due to heat-flow from (or to) the stack material and the thermal gradient that is imposed therein.

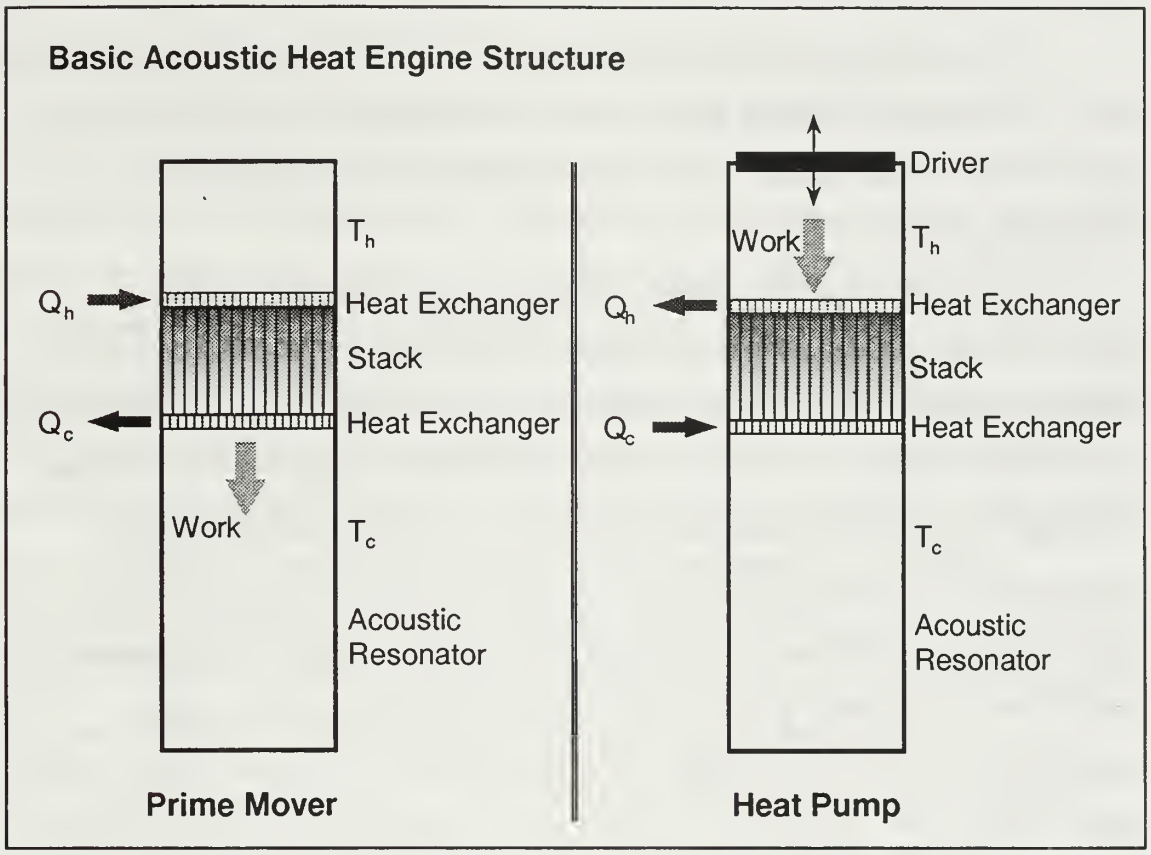


Figure 2.3. The basic structure of the two types of acoustic heat engines. The prime mover uses the externally applied temperature gradient to create acoustic waves. The heat pump absorbs work from an acoustic wave and pumps heat from a lower to a higher temperature. The interaction of two different thermodynamic media in the stack produces the resultant thermoacoustic effects.

If a gas parcel is sufficiently distant from the stack plates, this process of heat-flow is not instantaneous, however. Instead there is a lag between the motion of the gas parcel and the temperature change that occurs. This latency in the temperature change is a consequence of the distance of the

parcel from the stack and its relatively poor thermal contact with it. The resultant time lag introduces the necessary phasing required to articulate the acoustic heat engine cycle.

2. The Acoustic Heat Engine Cycle

To describe the physical processes that occur in an acoustic heat engine cycle, we follow in detail a gas parcel as it completes an oscillation in the presence of a stack plate. This simple model serves to illustrate the underlying theory.

Not all parts of the oscillating gas contribute equally to the thermoacoustic effect. Only the parcels of gas that are at the appropriate lateral distance from the stack material play an important role. The thermal penetration depth, δ_κ, is approximately the distance that heat can diffuse through the gas during the time $1/\omega$, where ω is 2π times the acoustic frequency. The parcels of gas that are approximately δ_κ away from the stack plates will have sufficient phasing of the movement and thermodynamics to articulate the necessary cycle. For parcels closer than one thermal penetration depth from the plate, the temperature of the gas closely mirrors that of the plate. Conversely, parcels much farther than δ_κ have insufficient time during the acoustic oscillation to absorb heat from the stack. Furthermore, at one thermal penetration depth, the thermal expansion and contraction will be in the right phase with respect to the oscillating pressure to do (or absorb) net work. Thus, both the heat-flow and the work-flow will be a maximum around this distance. It is therefore these parcels of gas that are the primary carriers of heat and work in an acoustic heat engine.

The stack temperature gradient and its relative magnitude with respect to the adiabatic temperature changes of the oscillating gas parcels provide a key mechanism for completion the acoustic heat engine cycle. As

the gas parcel oscillates back and forth across a stack plate, it will be warmed and cooled as a result of the acoustic pressure oscillations. Additionally, as the parcel moves along the plate, the parcel sees differing plate temperatures. If the resulting temperature of the gas parcel is higher than the local temperature of the plate, heat will flow from the gas to the plate. Conversely, if the gas is cooler, heat will flow from the plate to the gas.

The critical temperature gradient is one for which the adiabatic temperature change in the gas parcel just equals the stack temperature change through which the parcel has just moved. At this gradient, there will be no net heat or work flow. Additionally, the sign of the work-flow is also dependent on the critical temperature gradient. For the prime mover mode, the local pressure is higher as heat-flows from the parcel to the plate than it is when heat-flows from the plate to the parcel. As a result, net work is added to the acoustic oscillation. In the heat pump, the opposite occurs and work is absorbed from the acoustic wave. Consequently, the critical temperature gradient marks the boundary between the heat pump and prime mover modes of operation. By controlling the stack temperature gradient, we can reverse the operation of the acoustic heat engine from that of a prime mover to that of a heat pump. Thus, although the processes of the acoustic heat engine are irreversible, the heat engine is functionally reversible. Figure 2.4 shows the acoustic heat engine cycle for a prime mover. The cycle for a heat pump is the exact reverse.

3. Heat and Work

In general, the length of the stack is greater than the displacement of any given gas parcel during an oscillation. Consequently, there exists an entire train of adjacent gas parcels, each constrained in short longitudinal oscillations, extending the length of the stack.

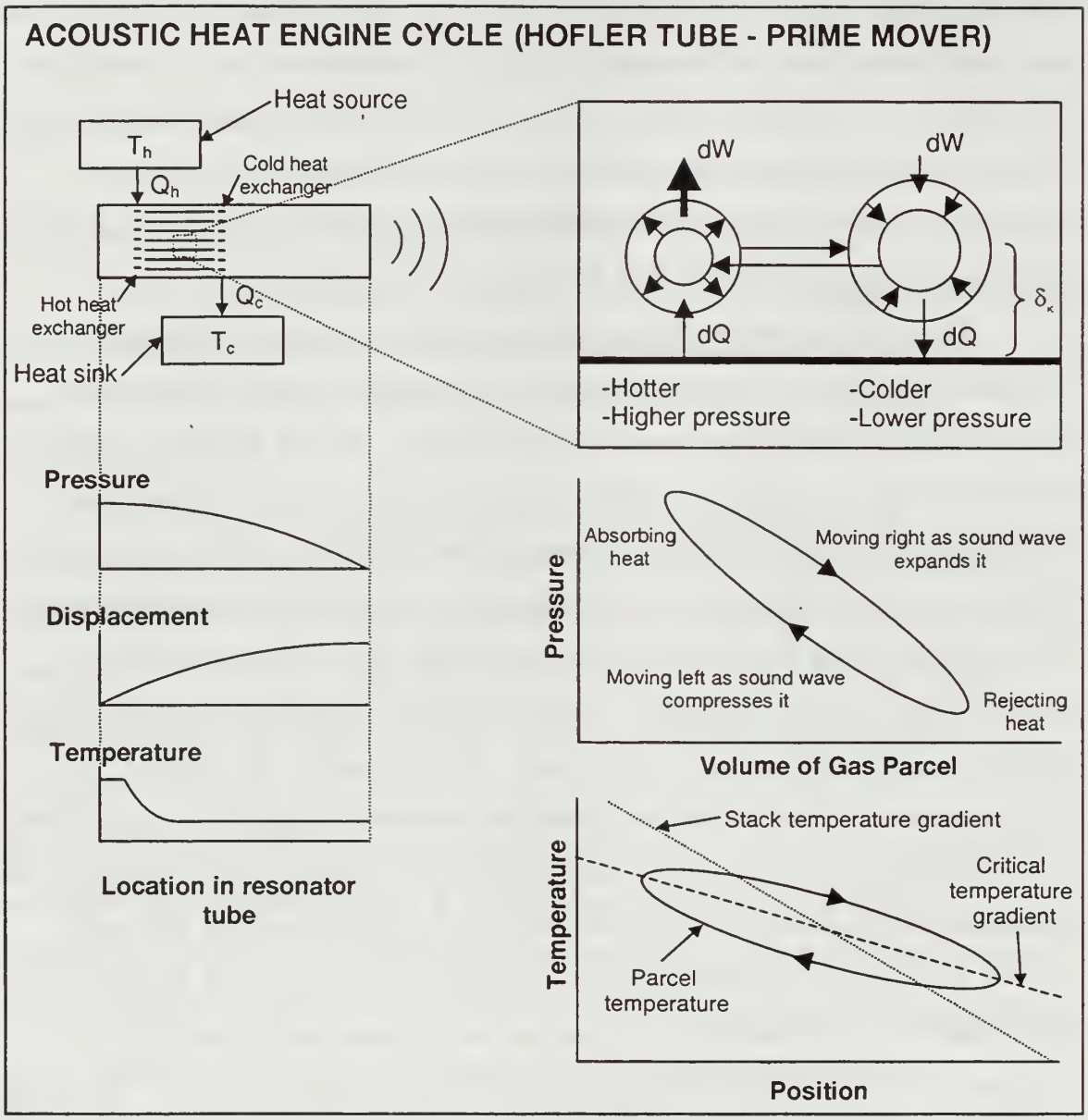


Figure 2.4. The acoustic heat engine cycle for a prime mover. As the gas parcel oscillates in the presence of the stack plates, both the adiabatic temperature change as well as local stack temperature play an important role. When the temperature of the adjacent plate is higher than the parcel temperature, heat will flow from the plate to the parcel. At the other end of the cycle, heat will flow from the parcel to the plate. Since the pressure during the heat-flow expansion step is higher than the pressure during the heat-flow compression step, net work is performed on the acoustic wave. For the heat pump mode of operation, all flows of heat and work are reversed and work is absorbed from the acoustic oscillation. [Ref. 1:p. 23]

Only parcels of gas on the end of the plate contribute to the net heat-flow. As the gas parcels oscillate, each absorbs heat at one end of its displacement and rejects it at the other. However, since the location of absorption for one parcel coincides with that of rejection for an adjacent parcel, no net heat-flow exists in the interior of the stack. Only at the stack ends, where the thermodynamic symmetry is broken, can heat-flow occur. For a prime mover, parcels oscillating beyond the cold end of the stack have nowhere to deposit the heat acquired during adiabatic warming. Instead, these parcels complete their round-trip oscillation and return to thermal contact with the end of the stack. Since the heat deposited by the adjacent parcel (still in thermal contact with the stack) will remain uncompensated, the cold end of the stack will begin to heat up. Consequently, a heat exchanger is used to remove the heat deposited and maintain the desired temperature gradient. Similarly, if the hot end of the stack is not supplied with a source of heat, the driving temperature gradient will also begin to fade. So long as the length of the stack does not exceed one quarter of the acoustic wavelength (The zero heat-flow at pressure and velocity nodes would otherwise alter its performance), the heat-flow remains independent of the stack length.

Since each gas parcel in the "bucket brigade" does (or absorbs) net work, the entire chain contributes to the overall work-flow. As a result, the total work done on (or by) a gas is roughly proportional to the length of the stack.

4. The Rott Wave and Energy Flow Equations

Though the previous sections were designed to give the reader a basic understanding of the theory behind thermoacoustic engine operation, a quantitative analysis would have included a lengthy derivation of the wave

and energy flow equations of Nikolaus Rott and modified by G. W. Swift. It is these equations that provide the foundation upon which numerical models of thermoacoustic devices are built. As such a derivation is beyond the scope of this thesis, the equations are presented in Appendix B without further explanation. A complete treatment is available in Ref. 2.

In addition to the Rott/Swift equations, a set of normalizations are defined and a set of non-dimensional thermoacoustic equations are also listed in Appendix B. It is these non-dimensional equations that are implemented in DSTAR and enable the use of normalized parameters.

When designing a new engine device, normalized parameters are more fundamental quantities and are relatively independent of the scale of the device. With experience, the designer discovers that fairly narrow ranges of values for the normalized parameters lead to optimal performance over a wide range of devices. Also, the operating frequency or a specified tube length can be allow to vary under the control of the boundary value solver, as a means of meeting the resonance condition. Under these design conditions, the device model is a rather "plastic" entity whose shape or geometry may vary from one iteration of the model to the next.

When modeling existing experimental devices, parameters can be expressed in standard mks or cgs units in order to simulated the experiment in concrete terms. As such, the "Design" and "Simulation" tasks are significantly different and both can be performed efficiently with DSTAR.

III. NUMERICAL COMPUTATIONAL METHODS

A. FIFTH ORDER ADAPTIVE STEPSIZE RUNGE-KUTTA

In simulating a thermoacoustic device, it is necessary to solve systems of first order ordinary differential equations. For some components, the analytic solutions can be easily obtained. However, for the bulk of the devices of interest, a numerical solution is required. To this end a Runge-Kutta method was used to solve the basic initial value problems. The Runge-Kutta algorithms used in DSTAR are derived from the code available from the University of British Columbia [Ref. 5]. These algorithms provided a useful foundation on which to develop the computational approach used in the code.

1. Runge-Kutta Methodology

There is a wide range of numerical methods available to solve ordinary differential equations. The simplest, and perhaps most well known method, is the forward Euler method. The forward Euler method takes the solution value, y_n, at position x_n and advances it to position x_{n+1} using the value of the derivative $f(y_n, x_n)$ at x_n,

$$y_{n+1} = y_n + hf(y_n, x_n) \quad , \tag{3.1}$$

where $h = \Delta x$. This method approximates a straight-line solution between the two points. While this method is fast, it is also inherently inaccurate. [Ref. 5]

In contrast to first order schemes such as the forward Euler method, Runge-Kutta methods are higher order, one-step schemes that make use of information at different stages between the beginning and end of a step. They are generally more stable and accurate than the forward Euler method but

are still relatively simple [Ref. 5]. For example, in a second order Runge-Kutta scheme, the derivative at the starting point is used to approximate the derivative at the midpoint of the interval. This midpoint derivative is then used to calculate the solution at the end of the interval. The midpoint method, or 2-stage Runge-Kutta, is written as follows [Ref. 5]:

$$\begin{aligned} k_1 &= hf(y_n, x_n) \\ k_2 &= hf(y_n + \frac{1}{2}k_1, x_n + \frac{1}{2}h) \\ y_{n+1} &= y_n + k_2 . \end{aligned} \tag{3.2}$$

In this case, k_1 and k_2 are intermediary values calculated in producing the final solution y_{n+1}. Higher order schemes will involve more intermediary terms but follow the same basic principle. A general s-stage Runge-Kutta method is written as,

$$\begin{aligned} k_i &= hf(y_n + \sum_{j=1}^{n} b_{ij}k_j, a_i h), \; i = 1,...,s \\ y_{n+1} &= y_n + \sum_{j=1}^{n} c_j k_j \end{aligned} \tag{3.3}$$

The Runge-Kutta formula coefficients, a_i, b_{ij}, and c_j are expressed in a tabular form known as the Runge-Kutta tableau as shown in Table 1.

i	a_i	b_{ij}				c_i
1	a_1	b_{11}	b_{12}	...	b_{1s}	c_1
2	a_2	b_{21}	b_{22}	...	b_{2s}	c_2
⋮	⋮	⋮	⋮		⋮	⋮
s	a_s	b_{s1}	b_{s2}	...	b_{ss}	c_s
j	=	1	2	...	s	

Table 3.1. The Runge-Kutta tableau.

2. Fehlberg RK45

Among the higher order Runge-Kutta methods, the Fehlberg RK45 provides a good compromise between speed of computation and accuracy of results. Additionally, this fifth order routine provides the added benefit of error estimation by the use of an embedded fourth order scheme. Both the fifth order and the embedded fourth order scheme use the same intermediary stages, k_1-k_6, but compute the final step using different coefficients, c_i and c_i^*.

$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 \quad 5^{\text{th}} \text{ order} \tag{3.4}$$

$$y^*_{n+1} = y_n + c^*_1k_1 + c^*_2k_2 + c^*_3k_3 + c^*_4k_4 + c^*_5k_5 + c^*_6k_6 \quad 4^{\text{th}} \text{ order} \tag{3.5}$$

The difference of these two results can be taken as an error estimate for the fourth order method. Since higher order methods are, in general, more accurate than lower order methods, the fourth order error can, in turn, be used as an estimate of the error for the fifth order method. This error estimate can now be used to adjust the stepsize, h, such that the integration is completed within the user-specified tolerances. This adaptive stepsize methodology enables the integration speed to be commensurate with the

nature of the differential equations being solved. For slowly varying functions, the adaptive stepsize routine will require very few intermediate points to compute the integration. For rapidly changing functions, the stepsize need only be reduced as small as is necessary to produce the desired accuracy.

The coefficients for the embedded Runge-Kutta scheme used in DSTAR are shown in Table 2 [Ref. 5].

i	a_i	b_{ij}						c_i	c_i^*
1								37/378	2825/27648
2	1/5	1/5						0	0
3	3/10	3/40	9/40					250/621	18575/48384
4	3/5	3/10	-9/10	6/5				125/594	13525/55296
5	1	-11/54	5/2	-70/27	35/27			0	277/14336
6	7/8	1631/55296	175/512	575/13824	44275/110592	253/4096		512/1771	1/4
j	=	1	2	3	4	5	6		

Table 3.2. Cash-Karp embedded Runge-Kutta tableau

B. NEWTON-RHAPHSON METHOD FOR MULTI-DIMENSIONAL ROOT FINDING

Rarely in solving the ordinary differential equations that describe a thermoacoustic device are all of the initial conditions completely known. Instead, one or more of the initial conditions is guessed, a solution is evaluated, and then boundary conditions are compared to the solution. If there is a match, the solution is correct. If not, the guess must be altered and a new solution computed. This process is repeated until the solution and boundary values converge. Consider the difference between a given target boundary condition and the corresponding calculated solution as a function in

and of itself. Then the problem reduces to finding the root, or zero, of this function. To this end, a Newton-Rhaphson method for root finding is employed by DSTAR.

The Newton-Rhaphson method involves evaluation of the function and its derivative at the guessed root position. The derivative is used to construct a geometric tangent to the function at this position. The tangent line intercept is then chosen as the next guess for the root. This process is iterated until the root is found. Algebraically, this process is equivalent to expanding the function in a first order Taylor series to make the linear approximation,

$$f(x+\delta) \approx f(x) + f'(x)\delta \ , \tag{3.6}$$

to determine the next point to try, $x+\delta$:

$$f(x+\delta) = 0 \rightarrow \delta = -\frac{f(x)}{f'(x)} \ . \tag{3.7}$$

If the iteration brings the function close to a local maximum or minimum, δ may become very large and the method may fail. Aside from these possible failures, the rate of convergence on the root is very large. [Ref. 6:p. 362]

Figure 3.1 gives a graphic illustration of the Newton-Rhaphson method of root finding [Ref. 6:p. 363].

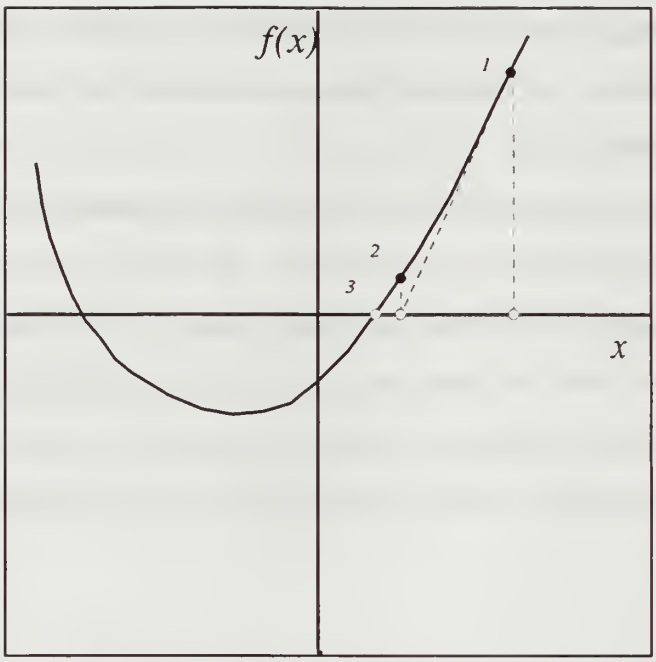


Figure 3.1. Graphical depiction of the Newton-Rhaphson method for finding the root of a function.

The Newton-Rhaphson method can easily be extended for multi-dimensional root finding. The general problem computed in the DSTAR model consists of an equal number of guessed initial conditions and targeted boundary conditions. If there are N such guesses and targets, then the problem gives N functions to be zeroed involving variables x_i, $i=1,2,\ldots,N$,

$$F_i(x_1, x_2, \ldots, x_N) = 0 \quad i = 1,2,\ldots,\mathrm{N} \ , \tag{3.8}$$

where **x** is now the entire vector of variables and **F** is the entire vector of functions to be zeroed. The Newton-Rhaphson method is now extended to N dimensions as [Ref. 6:p. 381],

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j} \delta x_j + \ldots \tag{3.9}$$

Noting that the matrix of partial derivatives in equation (3.9) is the Jacobian matrix **J**, the equation can be rewritten in matrix notation as [Ref. 6:p. 381]:

$$\mathbf{F}(\mathbf{x}+\delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J}\cdot\delta\mathbf{x} + \text{higher order terms.} \tag{3.10}$$

Again the left-hand side is equated to zero and the following equation results [Ref. 6:p. 381]:

$$\mathbf{J}\cdot\delta\mathbf{x} = -\mathbf{F}\ . \tag{3.11}$$

This matrix equation can now be solved for the guess corrections, $\delta\mathbf{x}$, using the matrix decomposition method described in the next section. This correction vector is added to the guess vector **x**, and the process is repeated until the boundary conditions are satisfied. The computational algorithm used in the DSTAR model is a modified version of the MNEWT algorithm provided in [Ref. 6].

C. LOWER-UPPER TRIANGULAR MATRIX DECOMPOSITION

1. Solving Linear Systems Using LU Decomposition

To solve the matrix equation (3.11), the DSTAR code makes use of a Gaussian elimination scheme known as LU decomposition. Given the linear system **Ax** = **b**, the matrix **A** can be decomposed into two triangular matrices, **L** and **U**,

$$\mathbf{A} = \mathbf{LU}\ , \tag{3.12}$$

where **L** is lower triangular and **U** is upper triangular. Once the matrix has been decomposed, the solution to the linear system can be easily solved in the following manner:

$$\mathbf{Ax} = \mathbf{b} \quad \mathbf{L}(\mathbf{Ux}) = \mathbf{b} \quad \text{Let } \mathbf{Ux} = \mathbf{y} \tag{3.13}$$

Now solve for the vector **y** in the resulting equation,

$$\mathbf{Ly} = \mathbf{b} \ . \tag{3.14}$$

The procedure is straightforward since a triangular matrix system may be easily solved by forward or backward substitution. In this case, the corresponding system of linear equations is,

$$\begin{array}{llll} L_{11}y_1 & & = b_1 & y_1 = \dfrac{b_1}{L_{11}} \\ L_{21}y_1 + L_{22}y_2 & & = b_2 \rightarrow & y_2 = \dfrac{(b_2 - L_{21}y_1)}{L_{22}} \\ L_{31}y_1 + L_{32}y_2 + L_{33}y_3 & & = b_3 & \vdots \\ \vdots \quad \vdots & \vdots & \vdots & \vdots \end{array} \tag{3.15}$$

Once **y** is a known, it is possible to solve for **x** in,

$$\mathbf{Ux} = \mathbf{y} \ , \tag{3.16}$$

in the same manner. [Ref. 7]

2. The LU Decomposition Algorithm

The LU decomposition is accomplished through the use of Crout's algorithm [Ref. 6]. This algorithm takes the input matrix **A** and performs an overwrite of the i^{th} row elements according the formula [Ref. 7],

$$\begin{aligned} L_{ij} &= U_{jj}^{-1}(A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}) \quad j \le i-1 \\ U_{ij} &= A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj} \quad j \ge i \ , \end{aligned} \tag{3.17}$$

which results in a combined matrix containing both the upper and lower triangular forms. Since a given LU decomposition is not unique, the algorithm is initiated by choosing the diagonal elements of the **L** matrix to be 1. This choice of diagonal elements precludes the need to store them and allows the combined matrix form. For a 4x4 matrix the result would be,

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ L_{21} & U_{22} & U_{23} & U_{24} \\ L_{31} & L_{32} & U_{33} & U_{34} \\ L_{41} & L_{42} & L_{43} & U_{44} \end{bmatrix}, \tag{3.18}$$

This result can then be used as shown previously to solve the matrix equation (3.11).

IV. PROGRAM ORGANIZATION AND OPERATION

A. DSTAR AND OBJECT ORIENTED PROGRAMMING

At its most fundamental level, any computer simulation is merely an algorithmic representation of the physical objects it describes. With the advent of object oriented programming, these algorithms have become both easier to construct and to maintain as faithful representations of real-world objects. It is with these advantages in mind that DSTAR was written in C++, the fully object oriented version of the C programming language.

Before proceeding further with the description of the DSTAR computational approach, it will become advantageous to briefly define some of the more important aspects of C++ object oriented programming:

Object – An essentially reusable software component that models items in the real world [Ref. 8:p. 10].

Classes – The programmatic description of an object (i.e. the code). This includes both the data members (variables) as well as the methods (functions) that manipulate the data.

Inheritance – A form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and embellishing these with the capabilities the new classes require [Ref. 8:p. 520]. A class that inherits from another class is said to be a *derived* class. The class from which another class is derived is said to be the *base* class.

Virtual Functions – A method in a derived class that can be accessed through a pointer to its base class [Ref. 8:p. 565].

Polymorphism – The ability for objects of different classes related by inheritance to respond differently to the same member function call [Ref. 8: p.566].

Abstract Base Class – A base class that is never instantiated and merely serves as a template from which derived classes will be defined.

Operator Overloading – Allows the objects to respond to commonly used operators (e.g. + or -).

The ANSI Standard Library provided with most modern C++ compilers uses operator overloading and polymorphism to provide complex number support comparable to that of Fortran 77. This is useful for thermoacoustic calculations where sinusoidal time dependence is assumed in the form of an $e^{i\omega t}$ factor.

Similarly, overloading can be used to extend the math capabilities for vector and linear algebra systems. The class library MV++ [Ref. 9] is used in DSTAR to provide "loopless" vector operations comparable to that of Fortran 90.

The DSTAR program makes use of all of these advanced features of the C++ language resulting in code that is both easy to maintain and to extend for greater future capability.

B. THE DSTAR OBJECT MODEL

The structure of the DSTAR object model provides the central mechanism by which solutions to the one-dimensional wave equations for a given device geometry are solved. Through careful design and interaction of the objects, the model was created such that the precise definition of a particular device is not known at compile time. Instead, the user dynamically creates the design of a thermoacoustic engine at run time using the graphical interface. This is perhaps the key advantage of DSTAR over previous codes. Such dynamic construction of a particular simulation allows rapid

prototyping of thermoacoustic devices without having to rewrite the simulation code.

1. The Core Classes

To facilitate the run time definition of a thermoacoustic device, several key classes were developed which model the actual real world components. These core classes represent the thermoacoustic engine as a whole, its constituent components, and the physical attributes that define these components.

a) CTAEngine Class

The highest level object that is modeled is the thermoacoustic engine itself. This object, which is encapsulated in the class CTAEngine, provides the variables and methods that are common to the thermoacoustic device as a whole. This includes such things as the physical constants for the enclosed gas as well as the design frequency of the acoustic oscillations. Additionally, the CTAEngine class provides all the methods required to compute a continuous solution to the differential equations from one end of the device to the other. When boundary conditions are present, the CTAEngine iterates the solutions to the model until these conditions are satisfied. These boundary value problem solutions are calculated through use of the Newton-Rhaphson algorithm described in chapter 3.

b) CTAModule Class

Figure 4.1 shows an example of a thermoacoustic device, the Hofler Tube [Ref. 4], which has been subdivided into individual components.

These components or modules provide the next level of programmatic abstraction in the DSTAR model. Classes that are derived from the *abstract base class*, CTAModule, represent each component of the thermoacoustic engine. This base class provides the variables and methods common among all the thermoacoustic components. The Runge-Kutta integrator, previously described, is among the methods incorporated in this class. Since the CTAModule class is abstract, no objects of this class are ever instantiated. Instead, each particular component class is derived from CTAModule and thereby *inherits* its capabilities. This allows each component to be fundamentally different with respect to its geometry and computational methodology (i.e. the differential equations) and yet still conform to a common interface. CTAModule derived classes include models of tubes, stacks, heat exchangers and lumped elements as shown in Figure 4.2.

The CTAModule class contains several *virtual functions*. The `propagate()` method holds all the code that is required to compute the solution to the wave equation from one end of the component to the other. Ordinarily this would consist of a simple Runge-Kutta integration to find the solution. However, some components may have analytic solutions while others may require special mathematical treatment. Hence, the method `propagate()` is ordinarily overridden in the derived classes to provide the unique computational code required for that specific component.

The second *virtual function* in the CTAModule class is the `derivative()`method. As its name suggests, this method contains the code that provides numerical derivatives for the appropriate differential equations. These derivatives are required for the Runge-Kutta integration algorithm shown in equation (3.3). As such, the `derivative()` method must be uniquely implemented in all CTAModule derived classes.

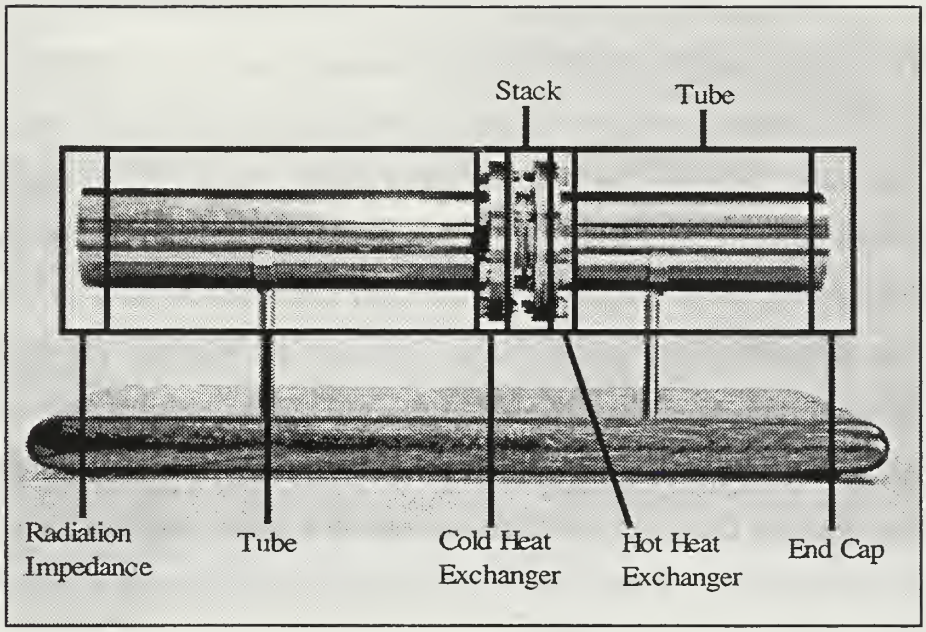


Figure 4.1. The Hofler Tube and its constituent components.

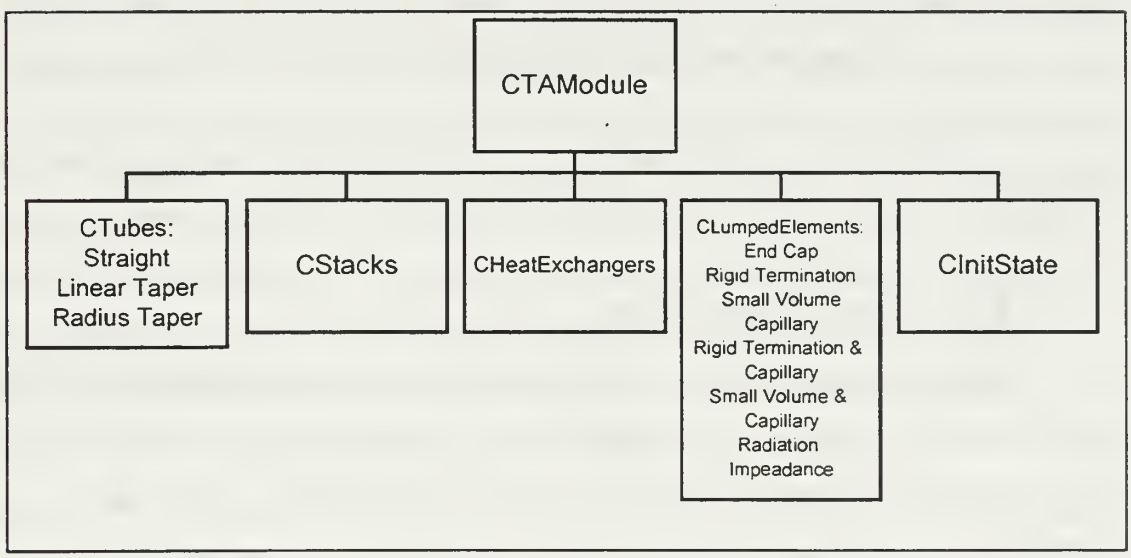


Figure 4.2. CTAModule class and its derived class hierarchy.

c) CTAElement Class

The CTAElement class provides the final basic building block of the DSTAR object model. Objects of this class are essentially variables that represent the physical structure of any given component (e.g. radius, length, etc.). In addition to providing the double precision value of that variable, the CTAElement object contains other vital information. This includes items such as the string name displayed by the user interface as well as Boolean values that flag different computational modes of the model. The objects of the CTAElement class are grouped in container classes within the CTAEngine and CTAModule classes. These container classes provide a convenient way for the user interface to display a given component's geometric properties without having to know a priori what they are.

2. Other Classes

The CTAEngine, CTAModule, and CTAElement classes provide the central building blocks of the DSTAR model but only comprise a few of the many classes in the code. Additionally there are classes which control the user interface, commercial classes purchased to enhance the program, and classes designed to provide additional mathematical ability (e.g. MV++). Figure 4.3 gives a summary of classes in the DSTAR object model.

3. Class Organization

While the core classes provide the framework upon which the DSTAR object model is built, it is the organization and interaction of these classes that provides the power and flexibility of the simulation. Again, the CTAEngine class lies at the heart of the simulation. There will only be one object of this class in any given model. It is, in a sense, the glue that holds

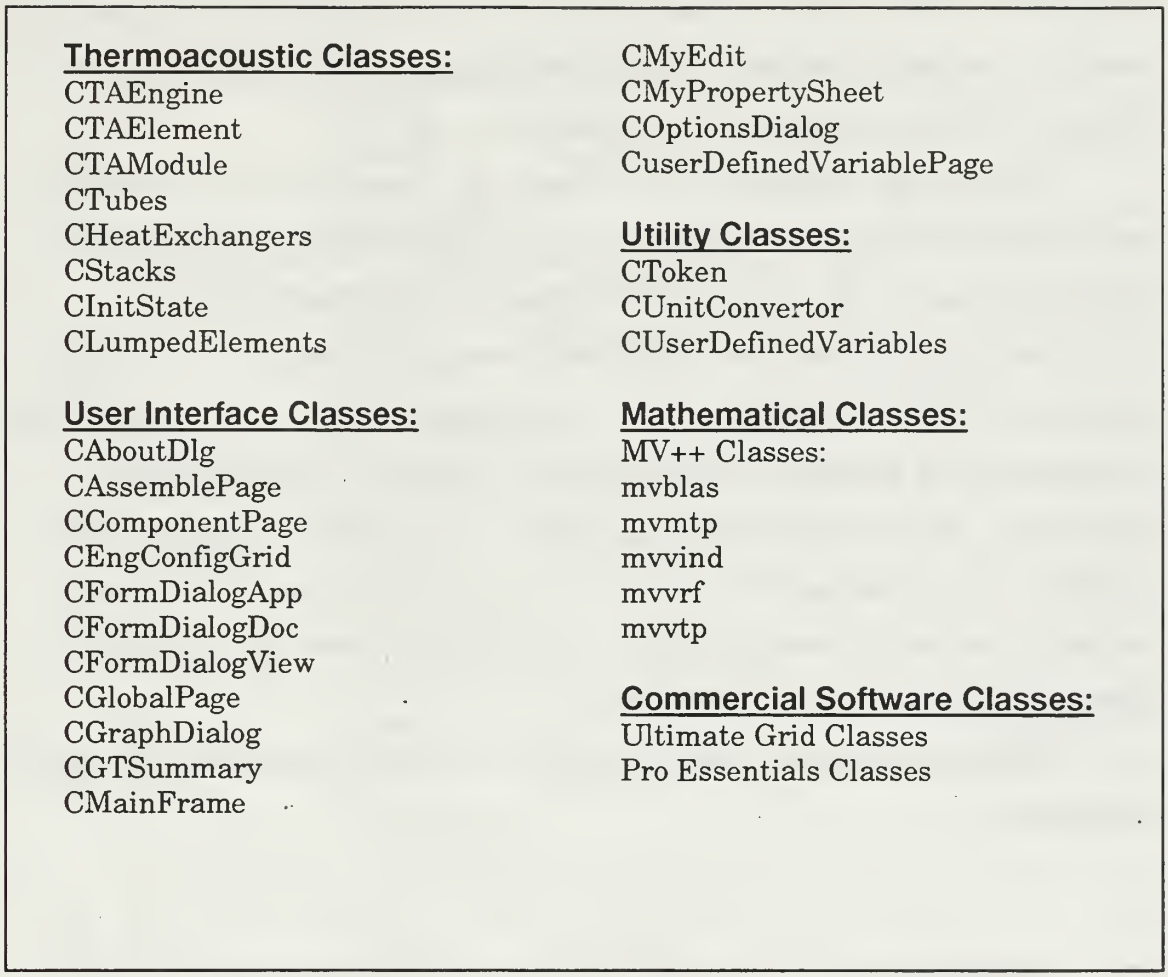


Figure 4.3. The classes of the DSTAR object model.

the model together. Within the CTAEngine object there are two data structures that comprise the totality of any given device geometry. The first structure is a collection of CTAElement objects that comprise the properties of the engine as a whole. This array is housed in a Microsoft Foundation Classes (MFC) container class called CObArray. The second structure is an ordered array of CTAModule objects that represents the particular thermoacoustic components of the given device. This array is also contained in a CObArray. The key advantage of this approach lies in the ease with which a given geometry can be altered by merely changing the makeup of

this array. New components can be inserted, moved and deleted with relative ease. In the past, changing a given simulation to reflect device geometry change would have required a rewrite of the code.

Within the CTAModule class objects there are three data structures which effectively describe the geometry and current physical state of the given component. The CObArrays entitled m_GeometryElements, m_InputStateElements, and m_DerivedElements contain all this data as arrays of CTAElement objects. As its name suggests, m_GeometryElements contains all the variables that describe the thermoacoustic component's geometry. The current states of the temperature, complex pressure, and complex volume velocity are stored in m_InputStateElements. The final array, m_DerivedElements, contains values derived from the local state elements such as acoustic impedance.

Figure 4.4 shows a Hofler Tube and the DSTAR class structure used to represent it.

C. COMPUTING AN INITIAL VALUE PROBLEM SOLUTION

Solving the system of first order differential equations in a continuous manner from one end of the thermoacoustic engine to the other is the central computational task of DSTAR. For each component, a steady state solution to the one-dimensional wave equation, described in Chapter II, must be calculated. These component solutions are pieced together to make a continuous solution for the entire device. The CTAEngine class method entitled `solve()` accomplishes this task.

The mechanism of the `solve()` method is made possible through the use of the *virtual polymorphic* function `propagate()` in each CTAModule derived class object. The solution is computed by iterating through the array

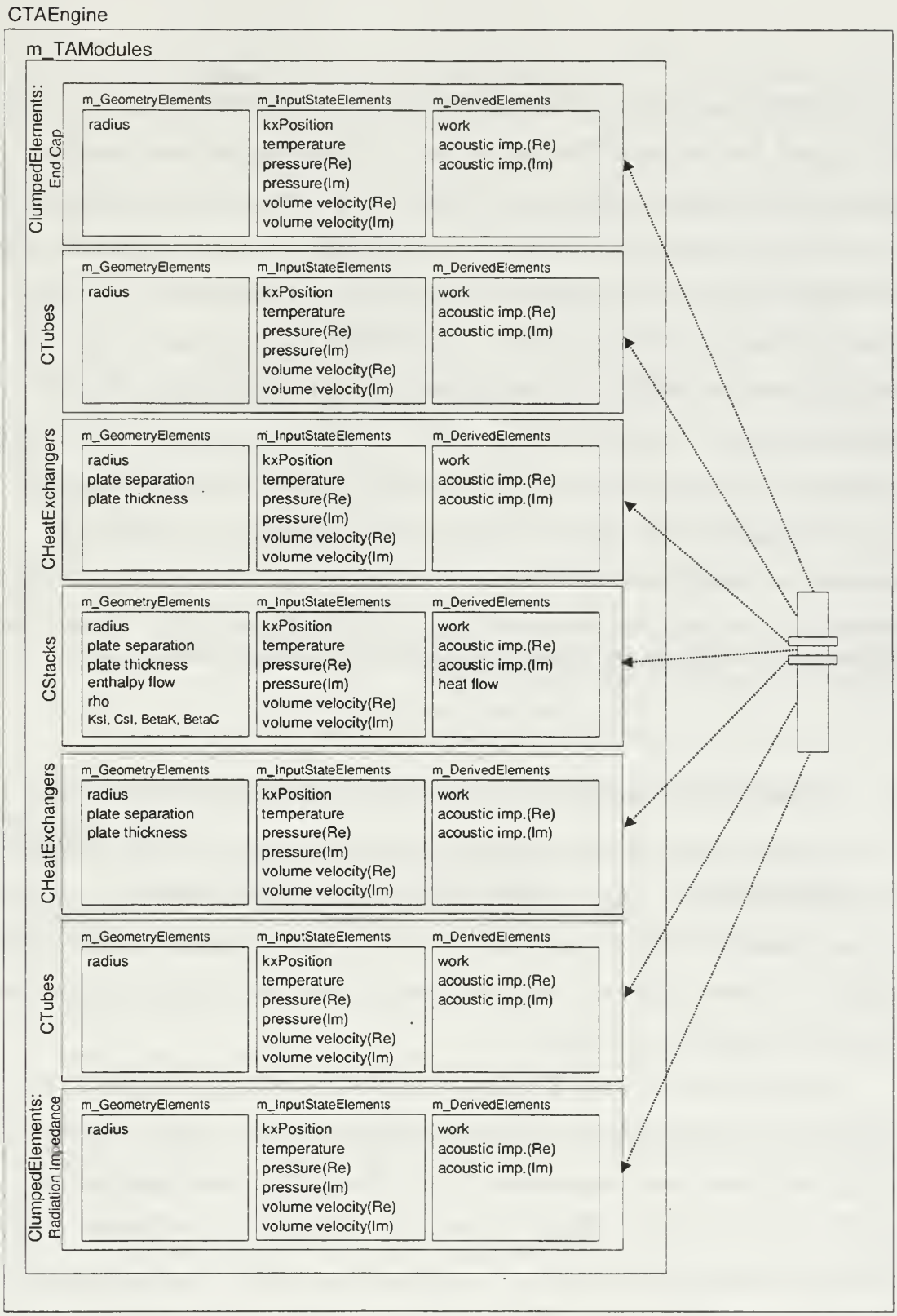


Figure 4.4. The Hofler Tube and the classes used to model it.

of CTAModule class derived objects and calling the `propagate()` function for each object. The values of the local state variables (temperature, complex pressure, and complex volume velocity) are passed in as initial conditions to the function. During the integration through the particular component, the intermediate values for the local state variables are recorded for later use. After the integration of a component is complete, the final local state quantities are retrieved and then passed on to the next component as its initial conditions. Since each `propagate()` function is unique to the type of component it models, the solution that results is that of a one-dimensional wave propagating from one end of the device to the other, in addition to temperature distribution and heat and enthalpy flow in the stack. Figure 4.5 displays a block diagram example of the `solve()` function for a three tube device and the DSTAR plot that resulted.

D. COMPUTING A BOUNDARY VALUE PROBLEM SOLUTION

Solving an initial value problem, although illustrative of the flexibility of the DSTAR code, is only the first step in calculating the wave equations for a given device. To find a true physical solution, boundary conditions must be applied. To accomplish this task, the Newton-Rhaphson method coupled with the LU decomposition are used.

Recalling the Newton-Rhaphson method described in chapter 3, the initial step of the algorithm required the generation of a vector of guessed initial conditions, **x**, as in equation (3.11). The CTAEngine class method `constructVectors()` realizes this process. This method iterates through the array of component objects and searches for variables that have been tagged by the user as guesses. When such a variable is encountered, a

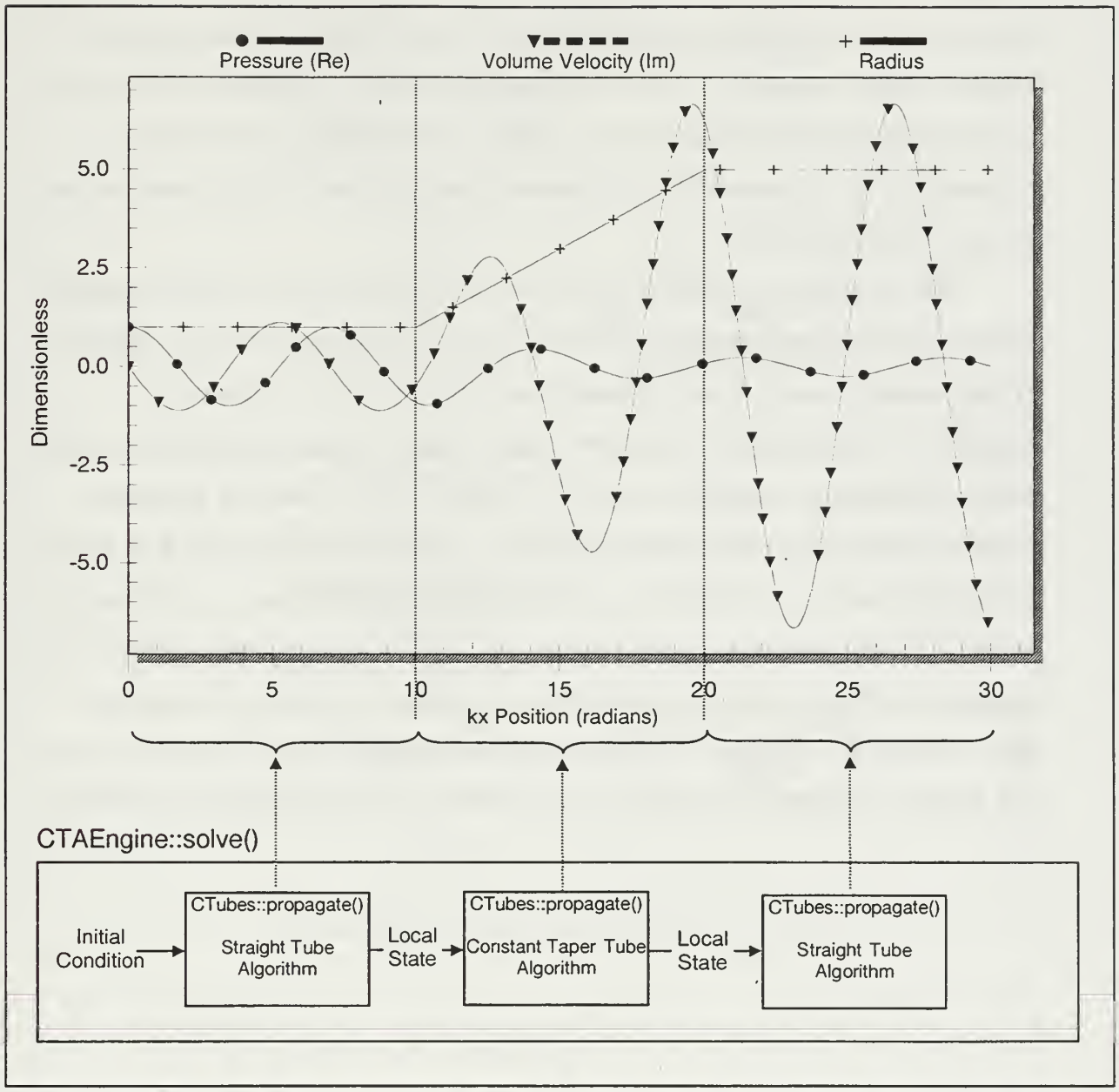


Figure 4.5. The bottom of the figure shows a block diagram of the solve() function in operation. After an initial condition is inserted, each component's propagate() function is called and the acoustic wave is transmitted from one end of the device to the other. For this example case there are three tube sections of different radii. This plot, exported from DSTAR, shows the effect of the increasing radius on the local state quantities, pressure (Re) and volume velocity (Im). See Appendix B for normalized variables and parameters.

pointer to that variable is appended to the guess vector. Additionally, the method collects pointers to all output variables that are tagged as targets as well as pointers to the actual target values. The difference of the target values and the corresponding calculated solution value is the function vector, **F,** as in equation (3.11).

To complete the solution, the `mnewt()` method is called. This method uses the previously described `solve()` method to compute the first solution. If the function vector, **F**, is sufficiently small in magnitude, then the boundary value problem is complete. More likely, at least one iteration of the Newton-Rhaphson algorithm will be required. In this case, the Jacobian matrix is calculated using finite difference partial derivatives. With **J** and **F** calculated, the LU decomposition and backward substitution are performed to solve (3.11) for the guess vector change, $\delta\mathbf{x}$. These changes are applied to the guesses and the process repeats until the solution converges or the process fails. Figure 4.6 shows a flow chart representation of this procedure. Figure 4.7 details an example of a computed boundary value problem from DSTAR.

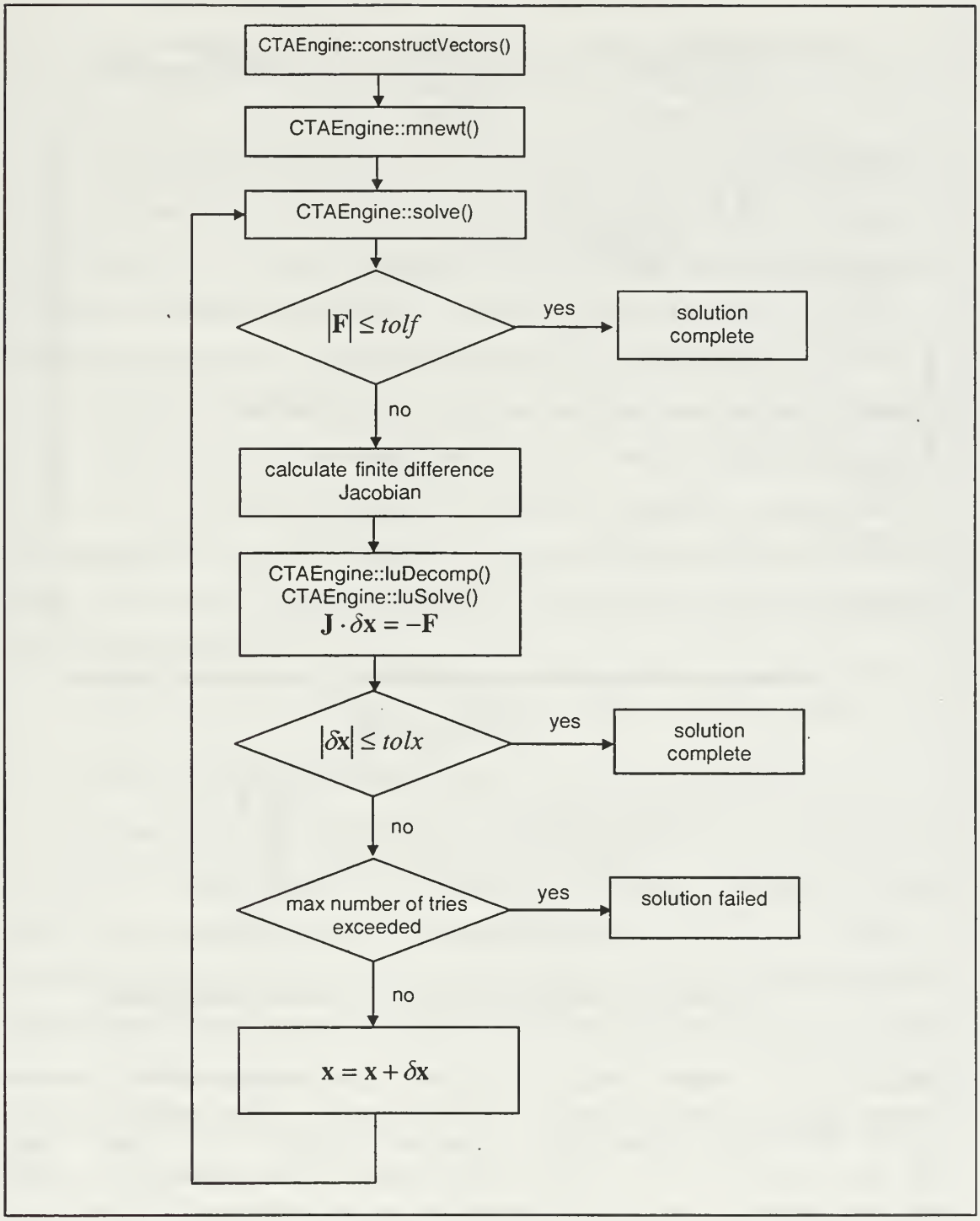


Figure 4.6. The flow chart shows the mechanism for calculating boundary value problems. The values *tolf*, and *tolx* are user specified tolerances for exiting the Newton-Rhaphson routine.

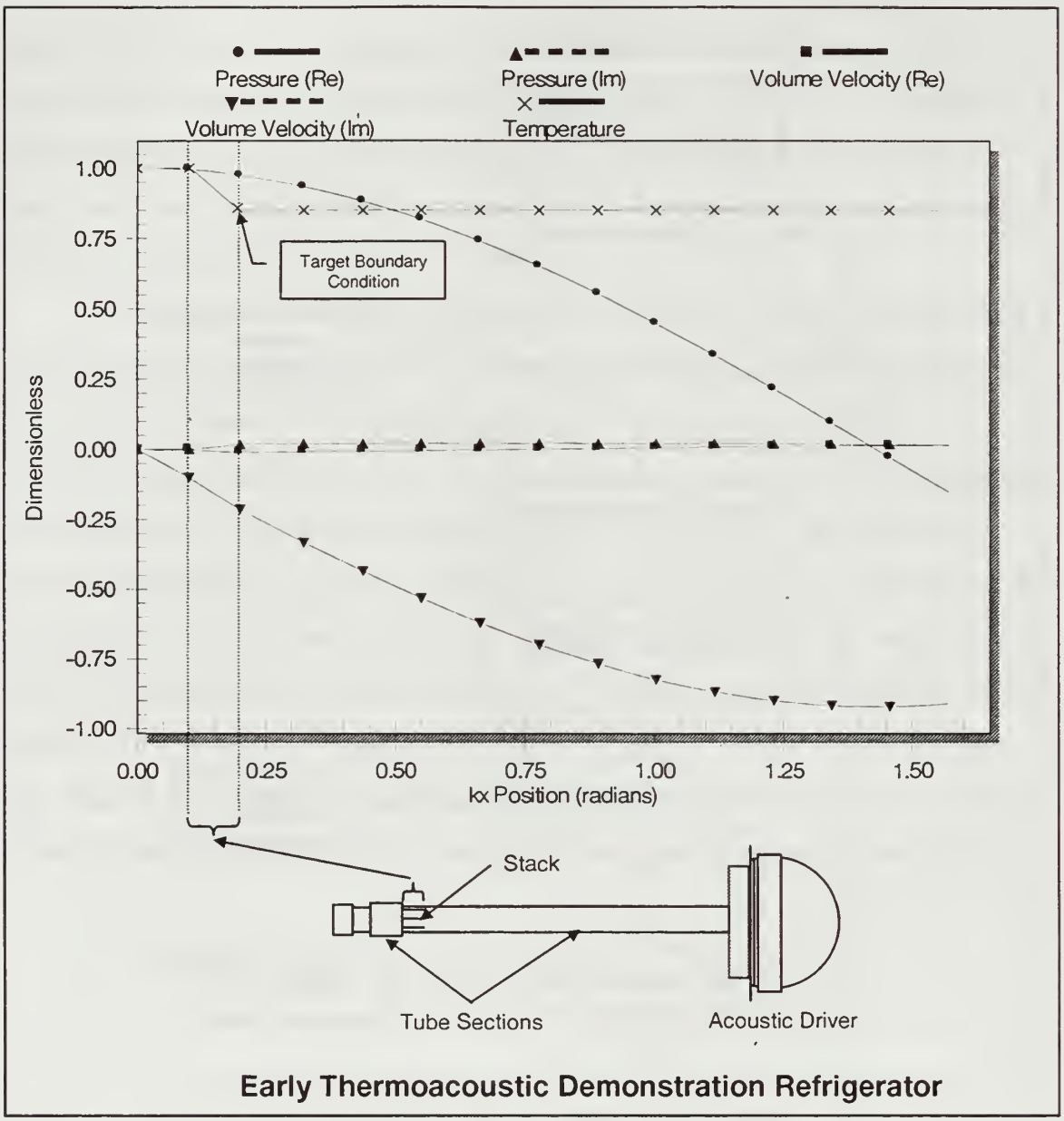


Figure 4.7. A simple model of an early thermoacoustic demonstration refrigerator shows the application of the boundary value problem solver in DSTAR. The initial temperature, a pressure anti-node, and volume velocity node were specified at the left end of the device. The enthalpy flow of the stack was guessed at –0.01 (ND) while the target boundary condition was a final temperature in the stack of .85 dimensionless temperature units (-20 C). After completion of the calculation, the required enthalpy flow to achieve this temperature span was –0.038 (ND). See Appendix B for normalized variables and parameters.

V. DSTAR GRAPHICAL USER INTERFACE

The DSTAR graphical user interface (GUI) works hand-in-hand with the core thermoacoustic classes to dynamically create a model of a given scalable design or experimental device. The GUI was constructed using the Microsoft Foundation Classes and the document/view paradigm. The resulting interface and application code provide a mechanism for manipulating the thermoacoustic classes previously described. This includes basic construction of a thermoacoustic model as well as disk storage and retrieval. To fully describe the code required to create the user interface is well beyond the scope of this paper. As such, the details of the GUI will be presented so that the reader may become familiar with their use rather than the underlying code.

A. MAIN PROGRAM WINDOW

Figure 5.1 shows the DSTAR main window. At the top of the screen is the menu bar which houses the program's four menus. The toolbar contains the icon shortcuts for some of the menu commands as well as the icons to initiate a computation. The bulk of the window is used by a multi-function tabbed view. Selecting one of the five tabs changes the tabbed portion of the screen revealing different aspects of program functionality. Lastly, the status bar at the bottom of the screen relays information about computational processes as well as some basic program help.

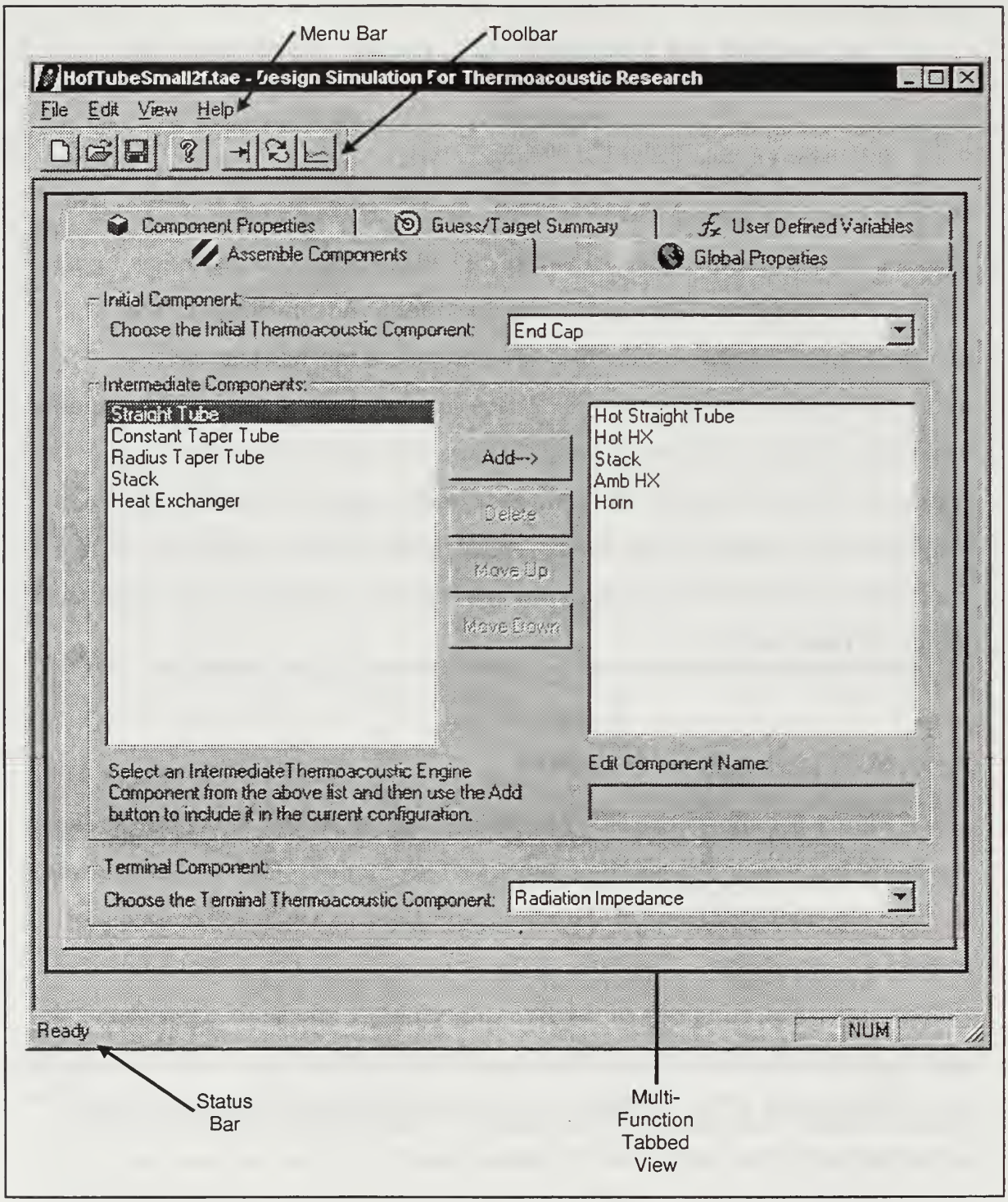


Figure 5.1. The figure shows the details of the DSTAR main program window. In this screen shot, the Assemble Components tab is selected.

1. Menu Commands

The File menu, as shown in Figure 5.2, contains all the program functions related to saving and retrieving DSTAR model files. Saving a DSTAR file results in the entire model geometry, including the current values of all the variables, being permanently stored to disk. Likewise, retrieving a file will result in dearchival, allowing resumption of previous work. The DTSAR files, extension .tae, are an encoded binary format and are not readable by other programs or text editors.

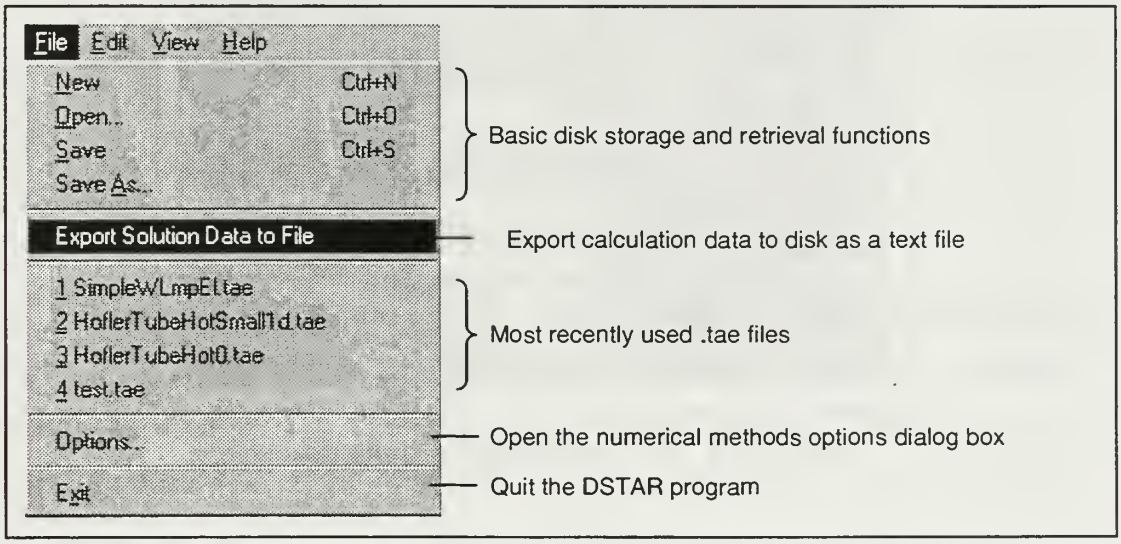


Figure 5.2. The File Menu and its functions.

The File menu also contains two other important features. The first is the Export Solution Data to File function. Once a calculation has been completed, the user can export all the data points that were collected during the integration to a text file on disk. Importing this file into a spreadsheet program will enable further manipulation of the data or creation of custom charts if desired. The last menu item, Options..., displays the numerical integration options dialog box. This window, displayed in Figure 5.3,

contains all the user selectable numerical tolerances for integration and boundary value problem computation.

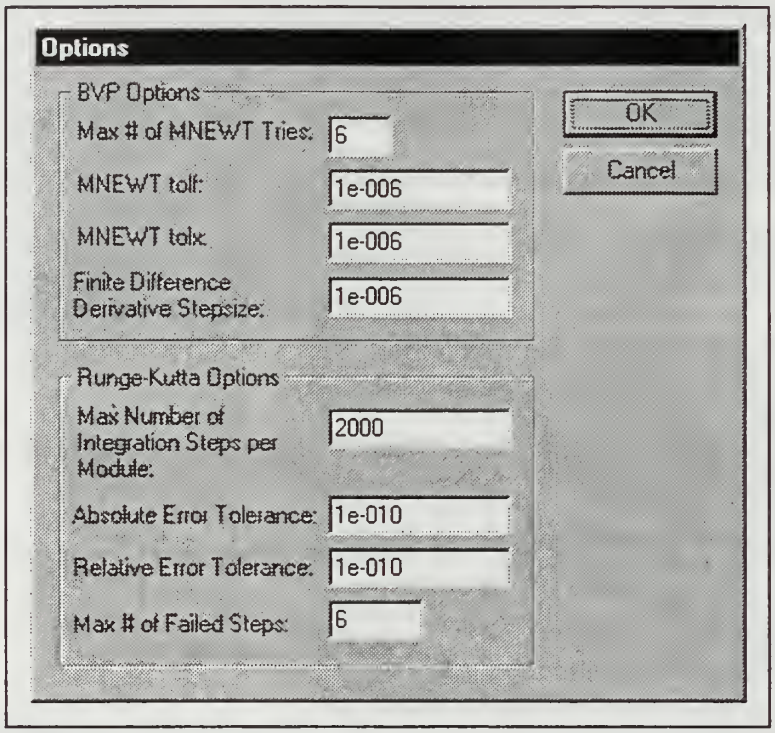


Figure 5.3. The numerical options dialog box.

The View menu allows the user to hide or display the toolbar, status bar, and output window.

The Edit and Help menus are not implemented in this version of DSTAR. However, they are included in the interface in anticipation of future capabilities.

2. The Toolbar

The toolbar contains menu shortcut icons as well as two buttons which initiate DSTAR calculations. The toolbar is normally located as shown in

Figure 5.1, however it may be dragged to other places in the main window as well as to the desktop. Figure 5.4 shows the toolbar buttons and their function.

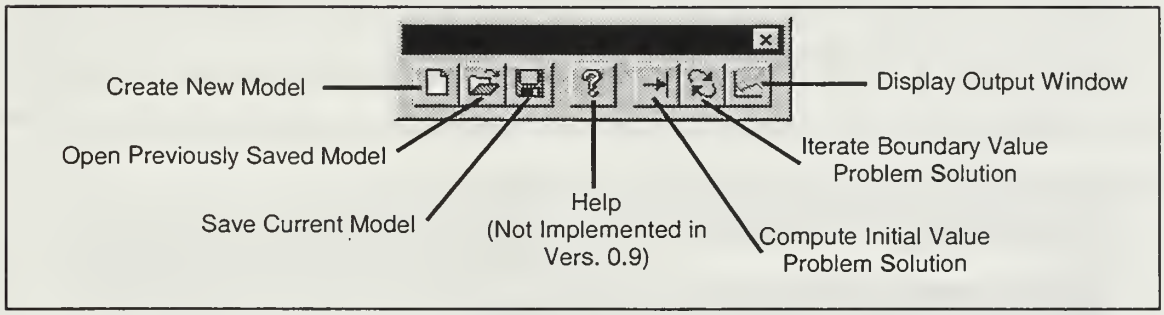


Figure 5.4. The DSTAR toolbar and icon functions.

3. The Multi-Function Tabbed View

The multi-function tabbed view provides the bulk of the user interface features of DSTAR while using a minimal amount of screen space. DSTAR can easily be used on a computer with an 800x600 display. There are five tabs in this view, each holding different interface features. The Assemble Components tab, which is the default, allows the user to construct the building block model of a thermoacoustic device being simulated. The Global Properties tab contains all the required information common to the entire model. After completion of the basic model and global properties, the user may select the Component Properties tab to enter the detailed description of each component. The User Defined Variables tab allows the user to construct specialized variables as functions of the standard DSTAR variables. Finally, the Guess/Target Summary tab is a compilation of all the variables in the model which have been selected as guessed initial conditions or targeted boundary conditions.

a) Assemble Components

The Assemble Components tab provides the GUI components required to build the black-box model of a thermoacoustic device. This tab is subdivided into three sections as shown in Figure 5.5. These three sections allow the selection of the initial, intermediate, and terminal thermoacoustic components of a DSTAR linear model. In general, integration begins with the initial component, passes through the intermediate components in the depicted order, and then ends at the terminal component.

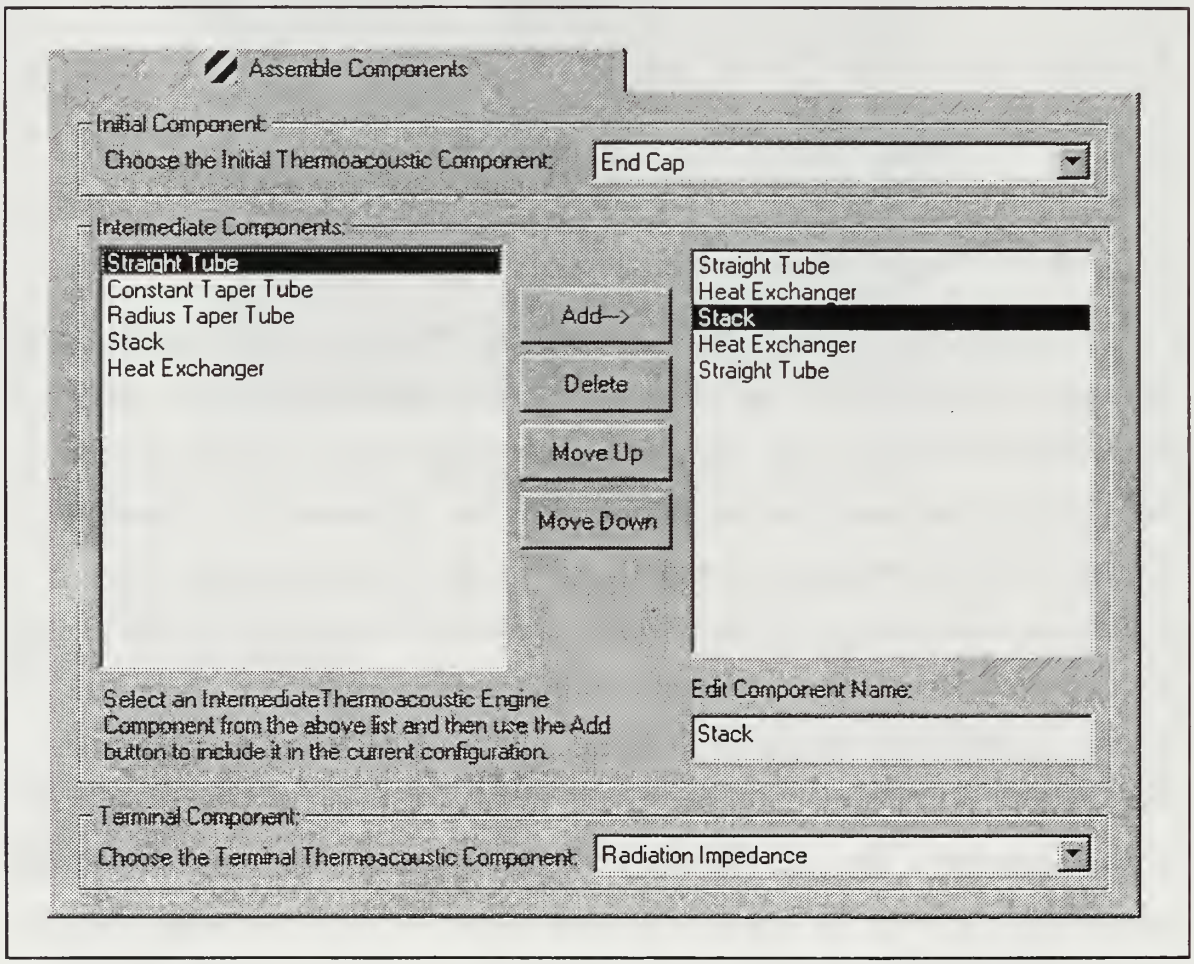


Figure 5.5. The Assemble Components tab allows the core components to be assembled in the proper order.

The Initial Component subsection provides a drop-list that allows the user to select one of several predefined initial thermoacoustic components. The Initial State component, unique to this subsection, has no physical geometry, but is rather an insertion mechanism for a known set of local state variables. The other six options, as shown in Figure 5.6, are all thermoacoustic lumped elements that require no integration and are coded with analytic solutions.

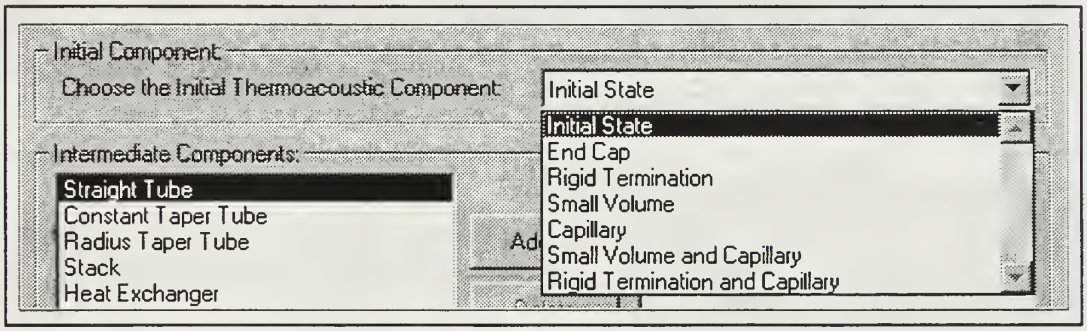


Figure 5.6. The Initial Component drop-list displays the options for the first component in the DSTAR model.

The Intermediate Components subsection provides for assembly of the major components of a device. The left-hand list-box shows an inventory of the thermoacoustic components that have been coded into DSTAR. The right-hand list-box details the configuration of the device currently being modeled. Buttons labeled Add, Delete, Move Up, and Move Down are used to manipulate the components into the proper configuration. Once a component has been added to the model, its name should be changed both to allow easier identification, as well as proper functioning of the user defined variables mechanism which requires unique names. User defined variables will be described in more detail later in this chapter.

The Terminal Component subsection works identically to the Initial Component subsection. As shown in Figure 5.7, the choices for the

terminal thermoacoustic component are the same with the exception of the addition of a Radiation Impedance lumped element and no Initial State option.

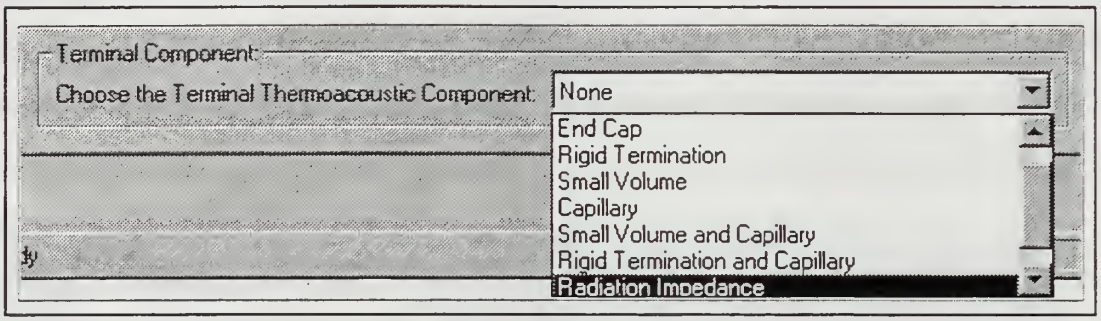


Figure 5.7. The Terminal Component drop-list.

b) Global Properties

The Global Properties tab, shown in Figure 5.8, contains a small spreadsheet-like interface for the input, and modification of the global engine properties. The three columns at the far right are used to designate a given variable as a guess, target, or optimized quantity. Note that optimized quantities are not implemented in DSTAR version 1.0. Quantities that are colored gray as well as checkboxes that have a gray background are not user editable. The units column depicts the appropriate dimensions that a given quantity should be entered in. In general, the global properties should be entered prior to editing any individual component properties since dimensional conversions may rely on the frequency, sound speed, and nominal radius. The two buttons on the bottom of the screen may be used to export or import the global variables to disk.

c) *Component Properties*

The details of the physical description for each thermoacoustic component are entered on the Component Properties tab. Again, a spreadsheet-like interface is used to ease the process of entering all the relevant data. As with the Global Properties tab, all quantities which are grayed out are not user editable. In general, these quantities are calculated by DSTAR and will be filled in by the program after a calculation is completed. Figure 5.9 shows an example of the parameters for a thermoacoustic stack.

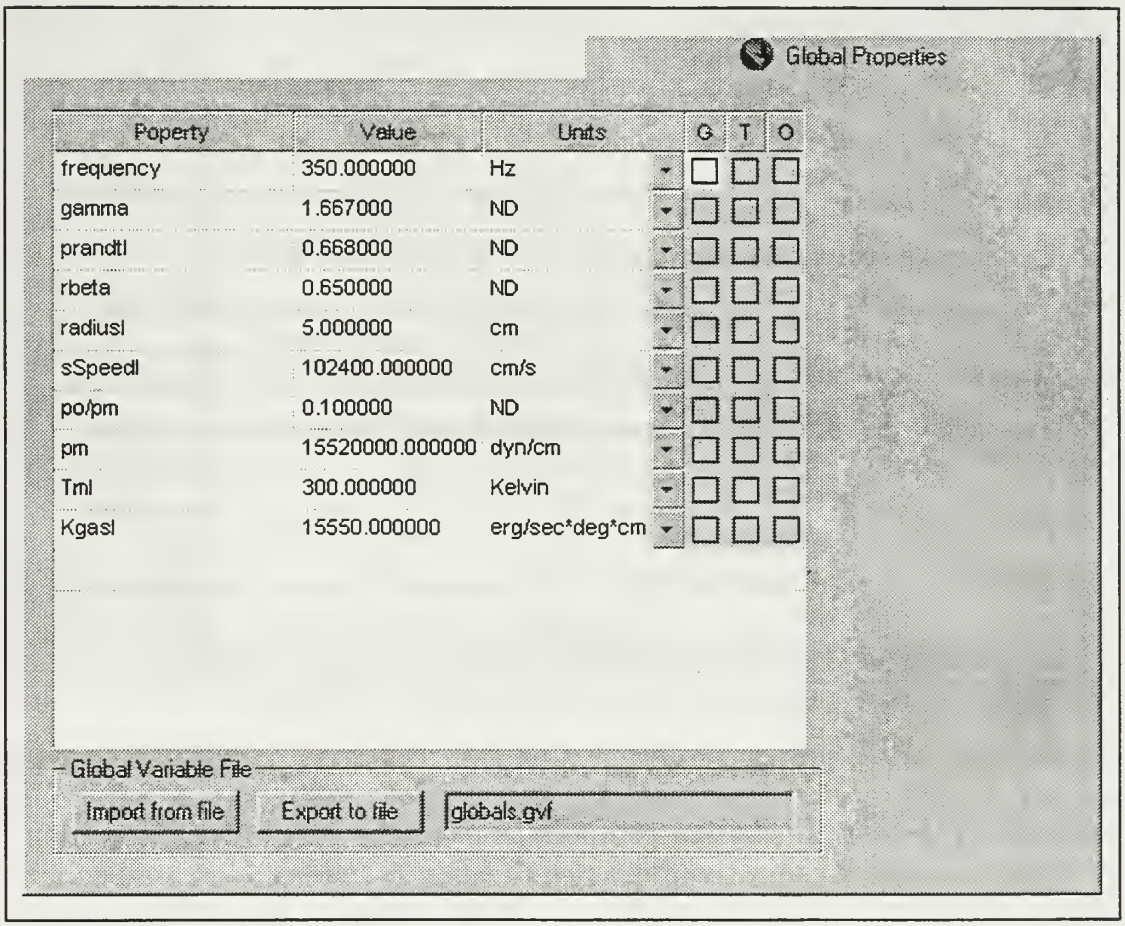


Figure 5.8. The Global Properties tab contains all the data which pertains to the thermoacoustic device as a whole.

The units column on this page allows numerical entries to be entered in any one of several dimensional choices. Most variables default to a non-dimensional unit for entry. If desired, the user may select a different unit to enter a given value. Units such as mks, cgs, english, and non-dimensional can be mixed at will. Once an entry has been made, selection of

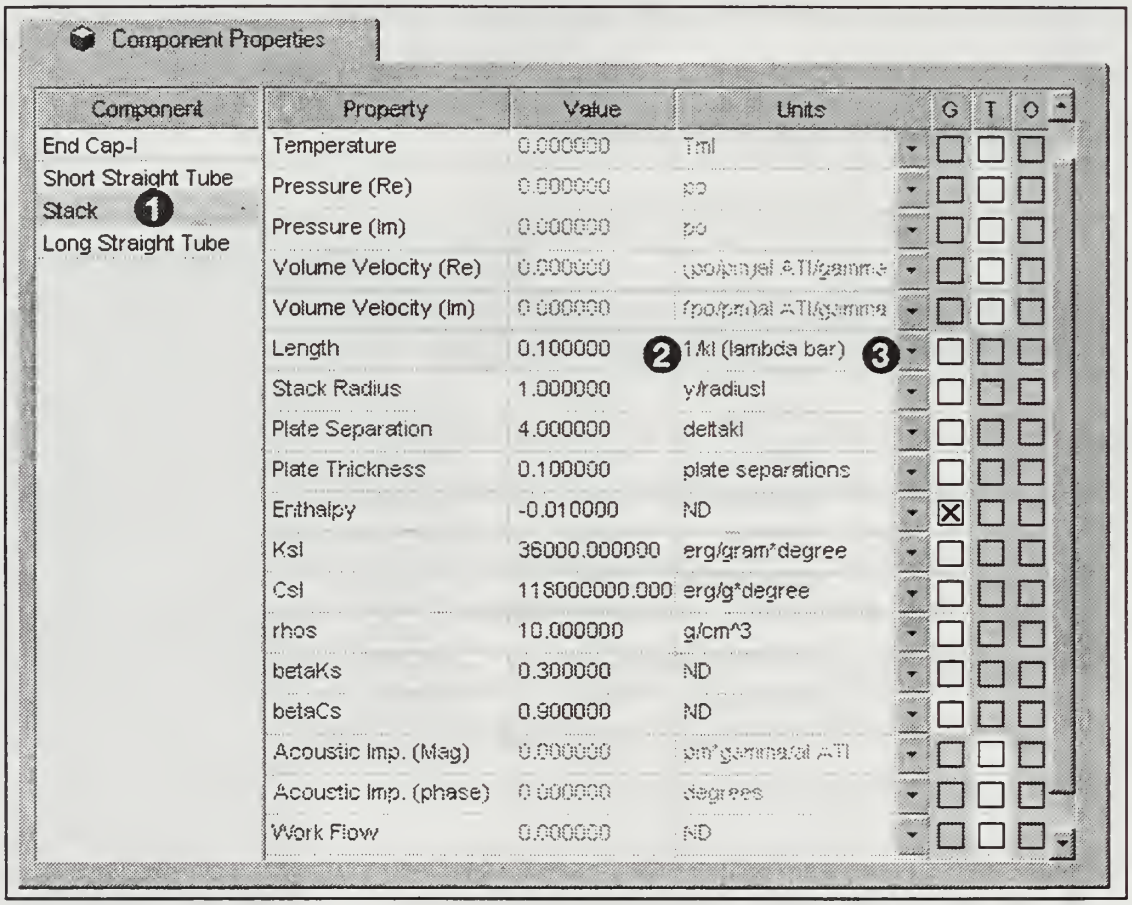


Figure 5.9. The Component Properties tab contains the bulk of the device geometry information required by the model. To edit a component's parameters, select the desired component from the current configuration (1). To enter a value, click on the appropriate column and enter a number (2). To change the displayed units, click the down arrow in the units column and select the desired dimensions (3). Note that all quantities which appear gray are not user editable and, in general, are calculated by the code.

a different unit will result in conversion of the previous value to the new units. For example, to find the dimensioned length of a given component previously specified in dimensionless units, simply select the desired units and a conversion will instantly be performed. As previously stated, these conversions require that the global properties have already been correctly specified.

The use of the units column has an additional feature in DSTAR. If a quantity with dimensions of length is specified in dimensionless form, that quantity will automatically scale as the frequency, initial sound speed, or nominal radius changes. Conversely, if a length is specified in dimensioned units, it will remain fixed at that value regardless of changes in frequency or sound speed.

Creating a scaled model of a laboratory device is now fairly simple. First enter all the component parameters as dimensioned quantities. After the each value has been entered, change its units to the dimensionless form. Now the frequency of oscillations can be adjusted and the size of the device changed. Then the steady-state solution is found again. The scaled dimensioned quantities can then be retrieved from the model by reselecting the desired units.

d) User Defined Variables

The fourth tab in the DSTAR main window is the User Defined Variables tab. The local state quantities of temperature, complex pressure, and complex volume velocity fully describe the thermoacoustic waves that resonate in a given device. These values are the core quantities that are integrated to provide a solution to the various wave equations. Other quantities such as work-flow, heat-flow and acoustic impedance are then

calculated from these local state quantities. However, it is often necessary to define new, device specific quantities in order to gain more informative output from the model. The coefficient of performance (COP), which is a device specific figure of refrigeration merit, provides a good example of such a quantity. To create this kind of output, the User Defined Variables tab uses a Reverse Polish Notation (RPN) syntax as described in Figure 5.10.

e) Guess/Target Summary

The final tab in the multi-function view is the Guess/Target Summary tab. As its name suggests, this tab displays the list of all the variables that have been selected as guesses or targets as described in Chapters 3 and 4. In order to compute a boundary value problem, the number of guesses must equal the number of targets and this display provides a convenient way to check. Figure 5.11 displays this tab.

B. THE OUTPUT WINDOW

In addition to the main program window, DSTAR has a secondary output window. This window, which is displayed following a successful calculation, provides the graphical and textual output data from the model. As shown in Figure 5.12, one half of the output window has a continuous, end-to-end plot of the local state variables from the last calculation. To zoom in on a particular portion of the plot, the mouse may be used to drag a zoom box over the desired region of interest. Right-clicking the mouse in the plot region and selecting the Maximize option will enlarge the plot for easier viewing. Additional options such as printing, exporting, and customization may also be found by right clicking the plot.

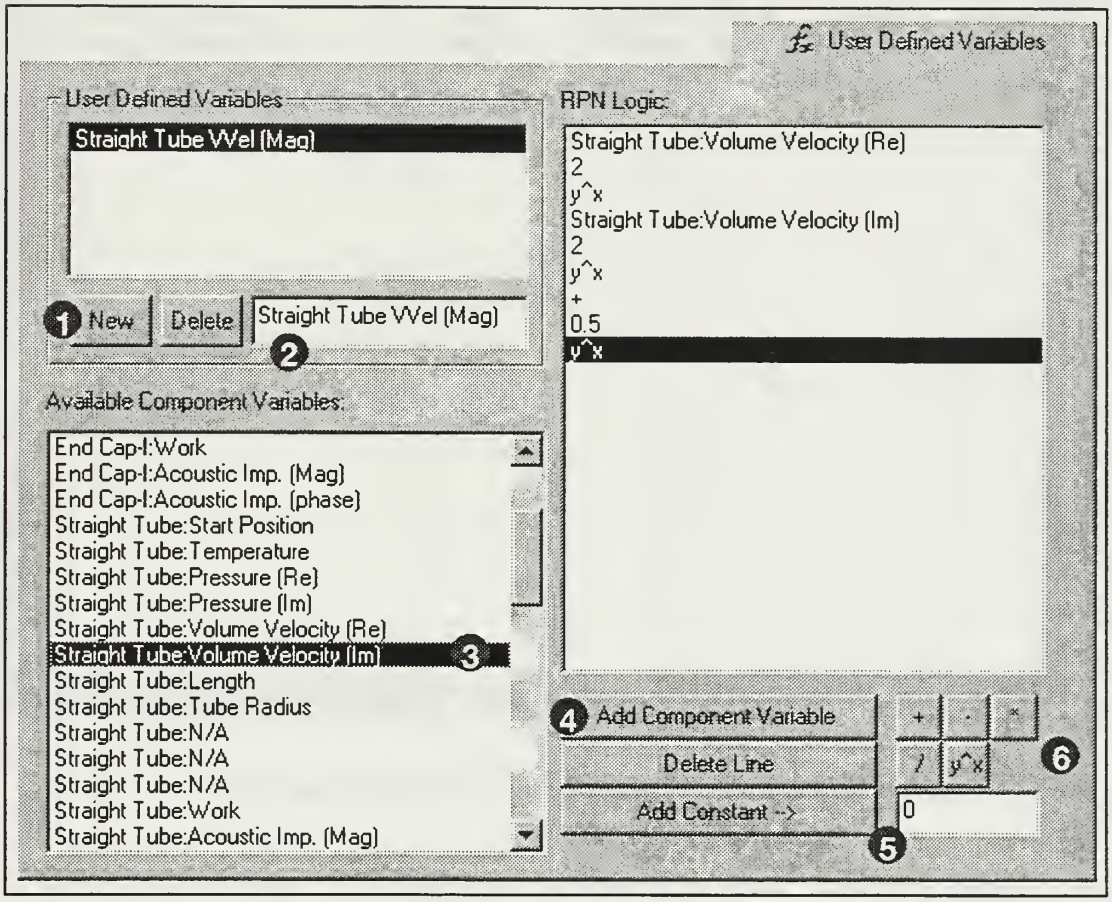


Figure 5.10. The User Defined Variables tab is used to create device specific variables as functions of the DSTAR local state, and geometry variables. To create a new variable, click the New button (1), and then change the default name (2). Here, the magnitude of the complex volume velocity has been created. After selecting a variable name from the model's complete list (3), the Add Component Variable button is pressed (4) inserting the variable's name into the RPN Logic list-box. To add a constant, enter the value in the edit box (5) and press the Add Constant button. Finally, an operator is added to the logic using the appropriate button (6). The expressions are evaluated from top to bottom using RPN syntax. Note that the user-defined variables use a name matching mechanism. As such, all components in the model should have unique names.

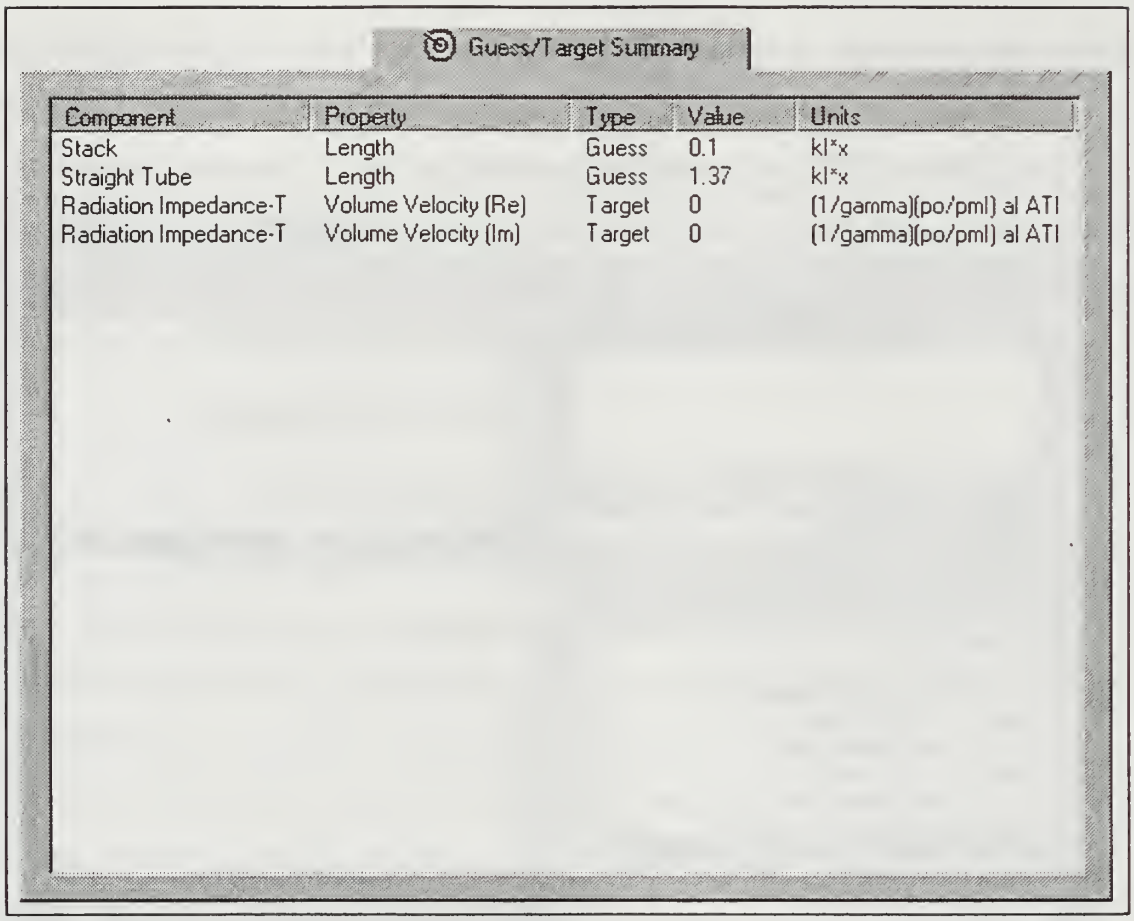


Figure 5.11. The Guess/Target Summary tab.

Below the plot, there is a region for typing notes about the current model. All text in the Model Notes window will be saved with the .tae file when it is archived to disk.

The right half of the output window displays a text dump provided by DSTAR following a calculation. All the model's data including geometry, local state, as well as the guesses both before and after a calculation, are included in this list. The textual output may be saved to disk at any time by using the Save Text button at bottom of the screen. It should be noted that this listing will contain the history of all calculations performed during the current DSTAR session. To clear the window of its content, simply press the

Clear Text button. Note the text contents of this list are not saved to disk when the model itself is archived using the File menu.

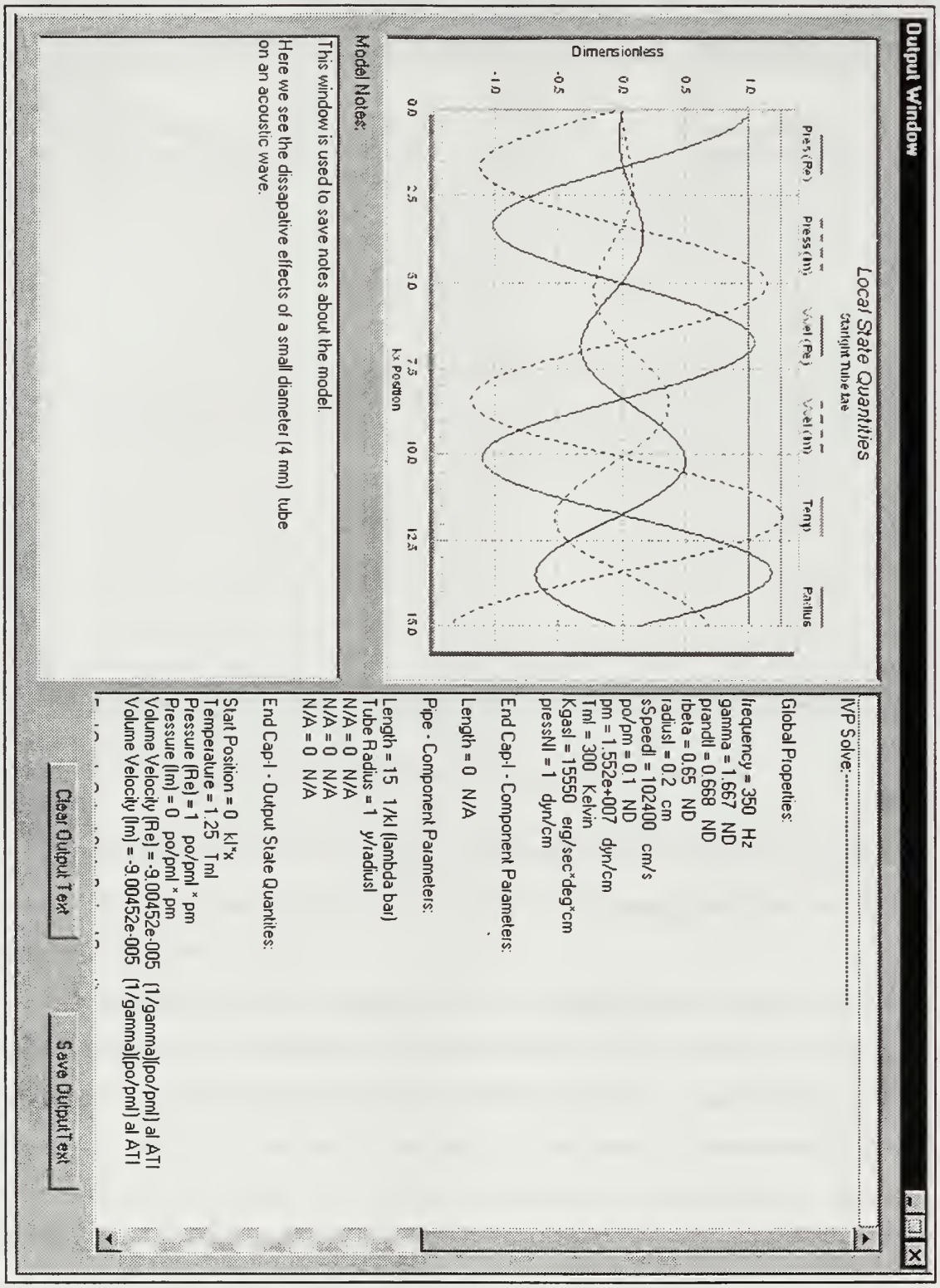


Figure 5.12. The DSTAR output window.

VI. DSTAR PRACTICAL EXAMPLE: AN ENHANCED HOFLER TUBE

To provide the reader with a practical example of the DSTAR code, a previously built thermoacoustic device was modeled and then modified for increased efficiency. Again, the Hofler Tube provides a straightforward and convenient example.

As previously shown in Figure 4.1, the Hofler Tube is a thermoacoustic prime mover that uses the supplied thermal gradient to produce acoustic power. In this case, the open end of the tube is immersed in liquid nitrogen for several minutes bringing its temperature to –190° C. When removed from the fluid, heat from the user's hand will flow from the warm end of the device, through the stack, to the newly created heat sink at the open end. The result is the spontaneous generation loud acoustic oscillations. This device has proved useful as a teaching aid and lecture demonstration.

Figure 6.1 details the DSTAR input parameters used to model the basic Hofler Tube. Although the design of the Hofler Tube is simple and relatively easy to construct, its efficiency suffers. Figure 6.2 shows a plot of the output acoustic state variables (steady state) within the Hofler Tube as well as the calculated efficiency retrieved from the DSTAR model. The efficiency of the Hofler Tube, as defined in Figure 2.1 where W is the radiated sound power, is quite poor at only 0.16%. (Note: the DSTAR model solved is very similar to the actual Hofler Tube except that the ambient-to-cold temperature span is replace with a hot-to-ambient temperature span of comparable ratio. This avoids the modeling ambiguities of a discontinuous temperature at the open end of the tube, in addition to modeling a genuine high temperature heat source, which is of interest for more practical engine designs.)

Global Properties:

```
frequency = 250 Hz
gamma = 1.4 ND
prandtl = 0.715 ND
rbeta = 0.75 ND
radiusI = 1.905 cm
sSpeedI = 34700 cm/s
po/pm = 0.1    ND
pm = 1.03e+006    dyn/cm
TmI = 300 Kelvin
KgasI = 2510 erg/sec*deg*cm
```

Component Parameters:

End Cap-I:
Length = 0 N/A

Hot End Straight Tube:
Length = 0.63 1/kI (lambda bar)
Tube Radius = 1 y/radiusI

Hot Heat Exchanger:
Length = 0.02 1/kI (lambda bar)
HX Radius = 1 y/radiusI
Plate Separation = 4.65 deltakI
Plate Thickness = 0.5 plate separations

Stack:
Length = 0.085 1/kI (lambda bar)
Stack Radius = 1 y/radiusI
Plate Separation = 4 deltakI
Plate Thickness = 0.1 plate separations
Enthalpy Flow = 1.123 ND
KsI = 1.344e+006 erg/gram*degree
CsI = 5.02e+006 erg/g*degree
rhos = 8.03   g/cm^3
betaKs = 0.42   ND
betaCs = 0.5   ND

Cold Heat Exchanger:
Length = 0.02 1/kI (lambda bar)
HX Radius = 1 y/radiusI
Plate Separation = 4.65 deltakI
Plate Thickness = 0.5 plate separations

Cold End Straight Tube:
Length = 0.8227 1/kI (lambda bar)
Tube Radius = 1 y/radiusI

Radiation Impedance-T:
Length = 0 N/A
Radius = 1.905 cm

Figure 6.1. The components and properties used to model the basic Hofler Tube using DSTAR.

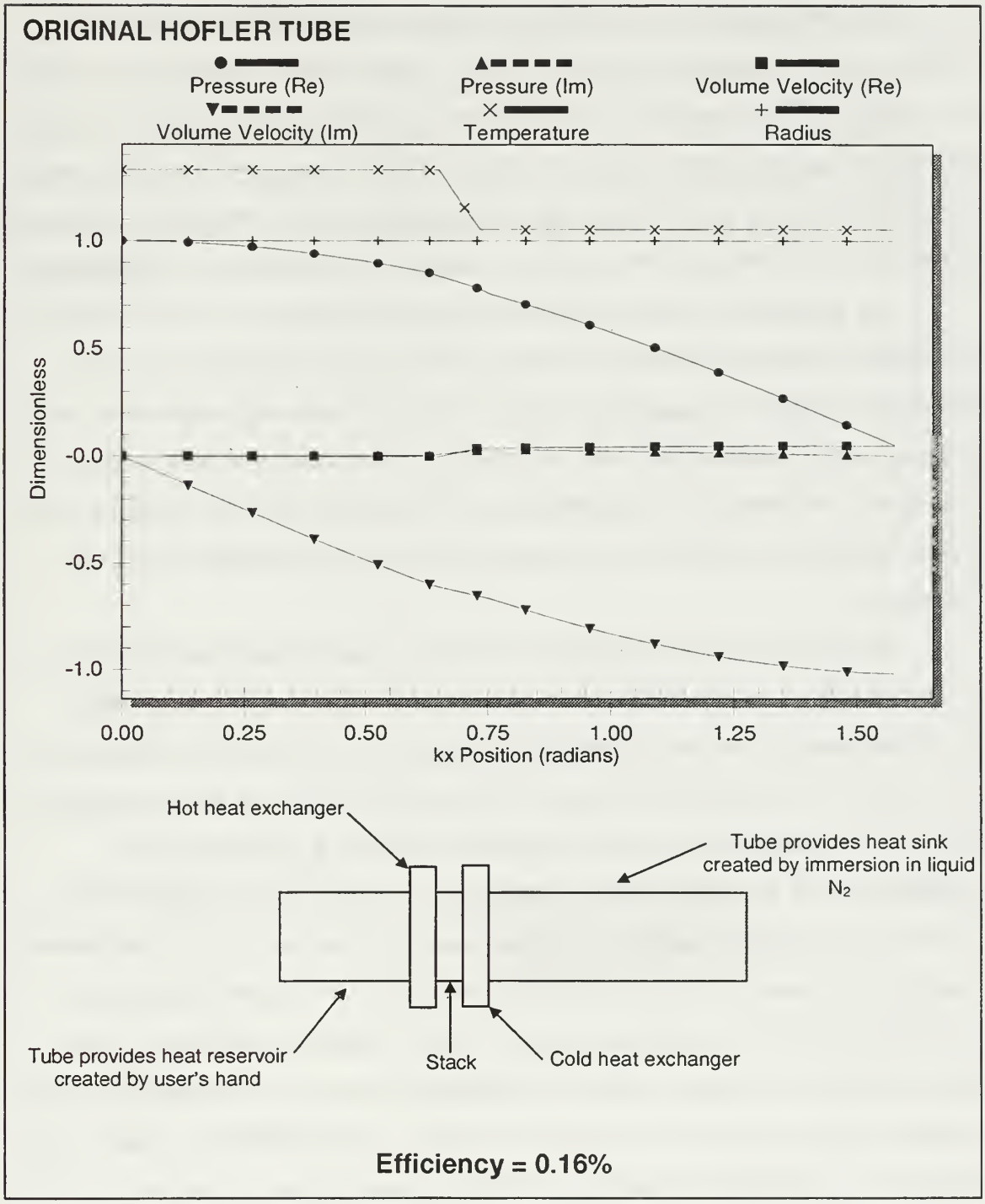


Figure 6.2. The DSTAR model plot of the steady state oscillations in the original Hofler Tube. The simple design yields very low efficiency . See Appendix B for normalized variables and parameters.

The original Hofler Tube was not designed with efficiency in mind. Rather, it was designed to operate with a temperature span that is as small as possible. This constraint contributes to the tube's inefficiency by limiting the design options available for the stack and other components. As a result, the stack position, stack length, and stack plate spacing were never optimized for efficiency but rather just to achieve onset of the spontaneous oscillations.

An additional source of inefficiency in the Hofler tube is the reflection of acoustic energy back into the tube at its open end. Since the room into which the sound is propagating is of relatively low acoustic impedance with respect to the inside of the tube, much of the wave energy which reaches the interface is reflected back into the tube. The result is a further decline in the overall ability to project acoustic power into the outside environment (i.e. efficiency).

Given our current knowledge of thermoacoustics and the DSTAR code we can design a more efficient demonstration device. Rather than using liquid nitrogen to create the temperature span, we will use an ordinary gas (e.g. butane or propane) heat source. This will free some of the constraints imposed by the use of the liquid nitrogen and allow a more thorough optimization of the tube's other components. As such, the modified Hofler tube's stack position, length, and plate spacing have been altered for better overall performance. A final optimization done to the Hofler tube is the addition of a horn element to the tube mouth. This gradual flare in tube diameter helps to reduce some of the acoustic reflections back into the tube, thereby transmitting more power to the room. The DSTAR tube component handles the varying cross section of the horn tube, plus a radiation component has been added to DSTAR to model the complex radiation impedance coupling power out of the mouth of the horn.

As a result of all the modifications, the efficiency of the Holfer Tube was increased to 2.25% while maintaining a relatively low temperature ratio from hot to ambient of 1.75. This improved efficiency is a two order of magnitude increase in the efficiency of the device. The DSTAR model parameters for the modified Hofler Tube are shown in Figure 6.3. Figure 6.4 shows the resultant plot of the state variables as well as a depiction of the new design.

Global Properties:

```
frequency = 900 Hz
gamma = 1.4 ND
prandtl = 0.715 ND
rbeta = 0.75 ND
radiusI = 0.7 cm
sSpeedI = 34400 cm/s
po/pm = 0.1   ND
pm = 1.03e+006   dyn/cm
TmI = 296 Kelvin
KgasI = 2510 erg/sec*deg*cm
```

Horn:
```
Length = 1.83 1/kI (lambda bar)
Initial Radius = 1.0 y/radiusI
Final Radius = 3.0 y/radiusI
Small Radius Angle = 0 degrees
Large Radius Angle = 14.3 degrees
```

Radiation Impedance-T:
```
Length = 0 N/A
Radius = 2.1 cm
```

Component Parameters:

End Cap-I:
```
Length = 0 N/A
```

Hot End Straight Tube:
```
Length = 0.25 1/kI (lambda bar)
Tube Radius = 1 y/radiusI
```

Hot Heat Exchanger:
```
Length = 0.03 1/kI (lambda bar)
HX Radius = 1 y/radiusI
Plate Separation = 5.0 deltakI
Plate Thickness = 0.6 plate separations
```

Stack:
```
Length = 0.16 1/kI (lambda bar)
Stack Radius = 1 y/radiusI
Plate Separation = 3.75 deltakI
Plate Thickness=0.083 plate separations
Enthalpy Flow = 0.454141 ND
KsI = 1.344e+006 erg/gram*degree
CsI = 5.02e+006 erg/g*degree
rhos = 8.03   g/cm^3
betaKs = 0.42   ND
betaCs = 0.5   ND
```

Ambient Heat Exchanger:
```
Length = 0.03 1/kI (lambda bar)
HX Radius = 1 y/radiusI
Plate Separation = 5 deltakI
Plate Thickness = 0.6 plate separations
```

Figure 6.3. The components and properties used to model the modified Hofler Tube using DSTAR.

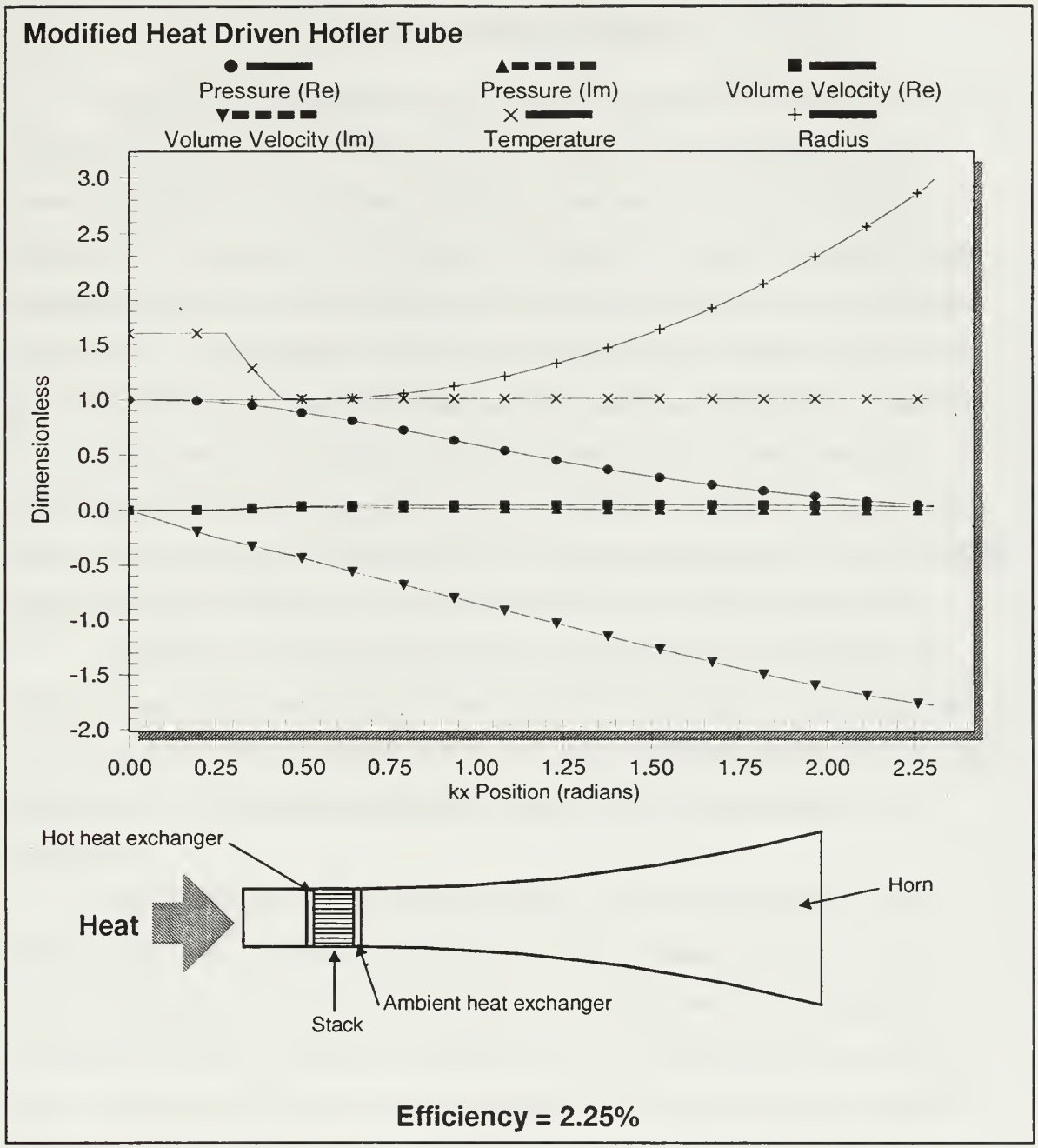


Figure 6.4. The DSTAR model plot of a modified heat driven Hofler Tube. For this design, the driving temperature gradient is created by addition of heat to the left end of the device using a suitable source such as a gas flame. Two separate tapers are used to decrease losses within the device. These modest changes result in an increase in efficiency of about two orders of magnitude. See Appendix B for normalized variables and parameters.

VII. CONCLUSION

This thesis has attempted to develop a new expert system code for the simulation and design of thermoacoustic devices. The assembled code provides a unique approach to modeling these devices using the object-oriented C++ language. It includes a Windows™ compliant graphical user interface as well as data storage and retrieval capabilities. As a result, the simulation is very flexible yet easy to use. Considerable effort was given to preserving the flexibility and breadth of the possible simulations, in addition to allowing easy modification of the source code for new thermoacoustic components. To demonstrate the utility of the code, a thermoacoustic prime mover was modeled and then optimized for better performance. The resulting model yielded an efficiency increase of nearly two orders of magnitude.

Totaling over 39,000 lines of code on approximately 900 pages, the DSTAR C++ code is too lengthy to be included in this thesis. The final code is approximately 1/3 commercial software add-ins (plotting capabilities and grid component), 1/3 graphical user interface and 1/3 thermoacoustics and numerics.

The DSTAR model, as it stands now, provides a complete set of components to design and simulate basic thermoacoustic devices. It is expected, however, that more components will be added as the program reaches maturity. Furthermore, DSTAR was designed with optimization routines in mind. Inclusion of an optimizer in future versions will greatly enhance the program's already capable performance and would provide an invaluable tool to aid the experimental physicist.

Those interested in obtaining the latest copy of the program should contact Professor Tom Hofler at the address listed in the distribution list at the end of this document.

APPENDIX A: EXTENDING DSTAR

A. ADDING PROGRAM FUNCTIONALITY

One of the great advantages to the DSTAR object oriented code is the ease with which it can be extended to add greater functionality. The most basic upgrade is the addition of new thermoacoustic component models to the code. Once added, the new components will integrate seamlessly into the existing code and are immediately available to create simulations. It should be noted that this appendix is intended for those familiar with the C++ language and not the general reader.

To create a new thermoacoustic component in DSTAR, it is necessary to write a new class that derives from the abstract base class CTAModule. As a derived class, the new thermoacoustic component class will inherit the functionality of the CTAModule class. The inherited methods, although not present in the new class's code, are always available to be called by the derived class. Figure A.1 shows the methods that all CTAModule derived classes will inherit.

In addition to defining methods passed on to derived classes, the CTAModule class also contains several pure virtual functions. A pure virtual function is one for which the interface, or function prototype, is defined in the base class but no implementation of the function is provided. As such, all derived classes must implement the details of the function. These functions are denoted in the base class's code by the "=0" appended to the end of the function prototype. Other functions which have the virtual keyword but are not pure virtual will have some type of basic implementation in the base class but are ordinarily overridden in the derived classes. The virtual functions defined in the CTAModule class are shown in Figure A.2.

The CTAModule class also defines some member variables and data structures that are required by the user interface. These variables and data structures are presented in Figure A.3.

```
void init() - Initialize Runge-Kutta tableau
void initializeStorageArrays(void) – Create arrays for data storage
void finalizeStorageArrays(void) – Compact data storage arrays
void initDerivedElements() – Allocate dynamic memory for Derived elements
void initStateElements(void) – Allocate dynamic memory for Local State elements
void cleanUp(void) – Delete all dynamically allocated memory
void calculateDerivedElements(const MV_Vector<double>& localState)
      – Calculate acoustic impedance, work and heat-flow as a function of a given set of local state
      variables
void adaptiveRK45Solve(MV_Vector<double>& localState, double positionStep)
      - Integrate a component from end to end using the Runge-Kutta adaptive method
void takeAdaptiveStep(MV_Vector<double>& localState, double& positionStep)
      - Calculate one step of a particular integration using adaptive stepsize to produce desired
      accuracy
void adaptiveRK45(MV_Vector<double>& localState, double& positionStep)
      - Performs single Runge-Kutta step calculation
std::complex<double> CDiffTanh(std::complex<double> z1)
      - Computes complex tanh(z1) for the SPECIAL CASE of Re(z1) = Im(z1)
void CBess(std::complex<double> z, std::complex<double>& J0,
      std::complex<double>& J1)
      - Complex Bessel functions
std::complex<double> CJ10r(std::complex<double> z)
       - Compute the bessel func. ratio J1(z)/J0(z) for z = (-x, x) where x is pos. & real.
double work(const MV_Vector<double>& localState)
      - Calculate the acoustic work-flow done as a function of a given set of local state variables
```

Figure A.1. The methods of the CTAModule class which are inherited by its daughter classes. Some of the more self-explanitory functions have been omitted for brevity.

Pure Virtual Functions:

`virtual MV_Vector<double> derivative(const MV_Vector<double>& localState, double position) = 0`
– Provides derivatives for the Runge-Kutta algorithm. MUST be implemented in all derived classes. If a class requires no integration, simply include an empty function that returns the passed-in argument.

`virtual void getModuleVariables(void) = 0` – Fills the components member variables with copies of the user interface values. These variables are then used to do all calculations. MUST be implemented in all derived classes.

Virtual Functions:

`virtual void propagate(MV_Vector<double>& localState)` – Generic algorithm to compute the values of the local state variables from end to end using the Runge-Kutta integrator while storing data and performing other housekeeping tasks . This function is most often overridden in derived classes to provide unique functionality.

`virtual void Serialize(CArchive& ar)` – Saves/retrieves components variables to disk. Should be overridden in derived classes.

`virtual void ensureConsistentObject(void)` – Called by user interface after a value has been entered. This is the programmers chance to scrutinize entries and ensure they are consistent with each other and with the model. Should be overridden in all derived classes

Figure A.2. The virtual functions defined in the CTAModule class.

`CTAEngine* theEngine` – Holds a pointer to the CTAEngine object at runtime

`CObArray m_InputStateElements, m_GeometryElements, m_DerivedElements`
- Arrays of CTAElement objects used by GUI to hold model's variables

`CArray<double,double> positData, tempData, presRealData, presImagData, vvelRealData, vvelImagData, radiusData` – State vs. Position storage arrays

`bool m_storingData` – Flag to store data

`bool m_stepsizeFixed` – Flag to turn off adaptive stepsize

`CString m_name` – String name of component

`double stepSize` – Initial stepsize

`std::complex<double> pres, vvel` – Can be used for calculations if desired

`double kxPos, temp` – Can be used for calculations if desired

`MV_Vector<double> solnError`
`MV_Vector<double> a`
`MV_ColMat<double> b` – Used by Runge-Kutta algorithm
`MV_Vector<double> c1`
`MV_Vector<double> c2`

Figure A.3. Data structures and member variables of the CTAModule class.

B. AN EXAMPLE THERMOACOUSTIC CLASS

To create a new thermoacoustic class, there are two files that must be inserted into the DSTAR project file. The header file, entitled "CClassName.h", contains all the function prototypes as well as the member variable declarations. The following code is an example of a thermoacoustic class header file for the class CMyTAComponent.

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
// CMyTAComponent.h: definition of the CMyTAComponent class.          //
//                                                                    //
////////////////////////////////////////////////////////////////////////

//This preprocessor directive is included to prevent multiple inclusions
//of this header file

#ifndef _MYTACOMPCLS_
#define _MYTACOMPCLS_

//Since this class derives from CTAModule it must have access to it's
//interface

#include "CTAModule.h"


class CMyTAComponent : public CTAModule
{

//To allow proper coordination between the CTAEngine class and this
//class, the CTAEngine is declared a friend

friend class CTAEngine;

//Functions and variables accessible from outside the class
public:

    DECLARE_SERIAL(CMyTAComponent)//This is a macro, not a function

    CMyTAComponent ();//Serialization requires this class to have an
                      //empty constructor

    CMyTAComponent (int dummyArgument);//Single argument constructor
                                       //used by the user interface

    virtual ~CMyTAComponent ();//Destructor

    virtual void Serialize(CArchive& ar);//File I/O

    void propagate(MV_Vector<double>& localState);//Calc. the solution
```

```
    MV_Vector<double> derivative(const MV_Vector<double>& localState,
                                 double position);//Return derivatives

    void getModuleVariables(void);//Get copies of GUI values
    void ensureConsistentObject(void);//Check user inputs

protected:

    double m_radius, m_area;//Member variables
    double calculateSomething(double radius);//A utility function
    int m_type;//A sample variable used to distinguish variants of the
               //class

private:

};

#endif
```

After the header file is created, an implementation file named "CClassName.cpp" must be created. This file contains all the code for the functions declared in the header file. The following is an example implementation file for the preceding header file.

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
// CMyTAComponent.cpp: definition of the CMyTAComponent class.        //
//                                                                    //
////////////////////////////////////////////////////////////////////////

#include "CMyTAComponent.h"//Include the header for this class

//Macro to implement serialization
IMPLEMENT_SERIAL(CMyTAComponent, CObject,1);

CMyTAComponent::CMyTAComponent ()
{
     //Empty constructor required by MFC serialization mechanism
}

//Single argument constructor used by the user interface to add
//components to the model.  The single dummy argument is used to
//distinguish this constructor from the default empty constructor
//but serves no real purpose in this class.  It may be used, however,
//to create variations of the class.
CMyTAComponent::CMyTAComponent (int dummyArgument)
{

     init();//Initializes Runge-Kutta tableau

     initStateElements();//Adds local state variables and allocates
                         //memory for them
```

```
    initDerivedElements();//Adds derived element variables and
                          //allocates memory for them

    m_name = "My New Component";//Set the name of the component
    m_type = dummyArgument;

    m_GeometryElements.Add( new CTAElement("Radius", 1.0, true, false,
    true, true, false) );//Add component specific geometry variables
                         //using this syntax to the m_GeometryElements
                         //array (Index 0 is occupied by the length)
                         //Details about the CTAElement constructor call
                         //can be found in the file CTAElement.cpp

    ((CTAElement*)(m_GeometryElements[1]))->SetAvailableUnits(
      Y_LENGTH_UNITS);//Set the appropriate units for the variable
                      //just created
                      //Details about units are available in the
                      //CTAElement.cpp file

}

CMyTAComponent::~CMyTAComponent ()
{
    cleanUp();//Call the base class function to destroy any allocated
              //memory
}


void CMyTAComponent::propagate(MV_Vector<double>& localState)
{
    //Put copies of GUI variables into local member variables
    //prior to any calculations
    getModuleVariables();

    //Use any member functions specific to class to perform
    //additional calculations or to make alterations
    //to the passed in local state prior to integration.
    m_area = calculateSomething(m_radius);
    localState(TEMP) *= m_area; //Note this is a nonsense calculation
                                //It is just used to illustrate a point

    //Prep storage arrays for data
    initializeStorageArrays();
    //Reset flag
    done = false;
    //Guess an initial stepsize
    stepSize = 0.00001;

    //Integrate the component from end to end using the adaptive RK
    adaptiveRK45Solve(localState,   stepSize);

    //Now that the integration is complete,
    //place the last calculated localState into the GUI accessable
    //variables.  This allows for target value comparison and access
    //by the user.
    ((CTAElement*)(m_InputStateElements[1]))->
```

```
            setValue(localState(TEMP));
    ((CTAElement*)(m_InputStateElements[2]))->
            setValue(localState(PRES_REAL));
    ((CTAElement*)(m_InputStateElements[3]))->
            setValue(localState(PRES_IMAG));
    ((CTAElement*)(m_InputStateElements[4]))->
            setValue(localState(VVEL_REAL));
    ((CTAElement*)(m_InputStateElements[5]))->
            setValue(localState(VVEL_IMAG));

    //Compact storage arrays after all data points have been added
    finalizeStorageArrays();

    //Calculate work, and acoustic impedance based on last values
    //of local state variables
    calculateDerivedElements(localState);

    return;

}

//Function returns the derivative of each local state variable in
//same position as the passed in variable
MV_Vector<double> CMyTAComponent::derivative(const MV_Vector<double>&
    localState, double position)
{
    //Make an empty vector to hold the derivatives
    MV_Vector<double> deriv(localState.size(), 0.0);

    //These are nonsense derivatives but illustrate where the proper
    //derivative should be placed in the returned MV_Vector
    deriv(1) = 0;//Temperture derivative
    deriv(2) = localState(VVEL_REAL);//Pressure (Re) derivative
    deriv(3) = localState(VVEL_IMAG);//Pressure (Im) derivative
    deriv(4) = 0;//Vvel (Re) derivative
    deriv(5) = 0;//Vvel (Im) derivative

    return deriv;
}


//Function stores and retrieves the component data to/from disk
void CMyTAComponent::Serialize(CArchive& ar)
{
    m_InputStateElements.Serialize(ar);
    m_GeometryElements.Serialize(ar);
    m_derivedElements.Serialize(ar);

    if (ar.IsStoring()){
      ar << m_name
         << m_type;//Add additional member variables to be
                   //stored in this way
    }
    else {
      ar >> m_name   // Variables must appear in exact same order
         >> m_type;//here as above
    }
```

```
}

void CMyTAComponent::getModuleVariables(void)
{
    //Put a copy of the user interface value of "Radius" into the
    //local member variable copy
    m_radius = ((CTAElement*)(m_GeometryElements[1]))->getValue();
}

//This function demonstrates the proper syntax for utility functions
//which are defined in the class
double CMyTAComponent::calculateSomething(double radius)
{
    return (PI*radius*radius);
}


void CMyTAComponent::ensureConsistentObject(void)
{
    getModuleVariables();
    if (m_radius < 0.0){

        //Reset the radius to a default value
        ((CTAElement*)(m_GeometryElements[1]))->setValue(1.0);

        //In this example, if the radius is < 0 we throw an exception
        //which is caught by the user interface and displayed.
        throw((Cstring)"Radius must be greater than zero");

}
```

C. ADDING THE NEW CLASS TO THE USER INTERFACE

Now that the class has been defined, it is necessary to modify the user interface to reflect the presence of the new thermoacoustic component. To do this, changes must be made to the AssemblePage.h and AssemblePage.cpp files.

First, the header for the new class must be included in the AssemblePage.h file. There is a list of #include's at the top. Append the new one as follows:

```
...

#include "CTubes.h"
#include "CStacks.h"
#include "CHeatExchangers.h"
#include "CLumpedElements.h"
#include "CMyTAComponent.h"

...
```

Second, the function CAssemblePage::OnSetActive(), in the AssemblePage.cpp file, must be modified to include the name of the new component:

```
...
pListBox->AddString("Constant Taper Tube");
pListBox->AddString("Radius Taper Tube");
pListBox->AddString("Stack");
pListBox->AddString("Heat Exchanger");
pListBox->AddString("My New Component Name");
...
```

Lastly, the function CAssemblePage::OnAdd() must be modified to include the following line of code:

```
...
case 3:
      pDoc->m_engine.m_TAModules.InsertAt(componentIndex+1, new
            CStacks(0));
      break;
case 4:
      pDoc->m_engine.m_TAModules.InsertAt(componentIndex+1, new
            CHeatExchangers(0));
      break;
//The number of the case statement must equal the index of the
//component's name in the list box
case 5:
      pDoc->m_engine.m_TAModules.InsertAt(componentIndex+1, new
            CMyTAComponent(0));
      break;
...
```

The project can now be recompiled and the program run. The new class is now an integral part of the program and will function identically to all the other thermoacoustic components.

APPENDIX B: SYMBOLS AND EQUATIONS

LIST OF SYMBOLS

A	area
a	sound speed
COP	coefficient of performance
c_p	isobaric heat capacity per unit mass
$\dot{H}$	total energy flow
i	the imaginary number
I	initial or nominal
Im	imaginary part
K	thermal conductivity
l	plate half-thickness
ND	non-dimensional
P,p	pressure
Q	heat (subscript h or c indicates heat accepted or rejected from a hot/cold reservoir)
$\dot{Q}$	heat-flow
Re	real part
T	temperature (subscript h or c indicates temperature of a hot/cold reservoir)
W	work
$\dot{W}$	work-flow or acoustic power
y	position perpendicular to sound propagation
y_o	plate half-gap
β	thermal expansion coefficient
γ	ratio of isobaric to iscochoric specific heats
δ_κ	thermal penetration depth
δ_ν	viscous penetration depth
ε_s	plate heat capacity ratio
κ	thermal diffusivity
λ	wavelength
μ	dynamic viscosity
ν	kinematic viscosity
Π	perimeter
ρ	density
σ	Prandtl number
ω	angular frequency
1	1st order quantity (subscript)
2	2nd order quantity (subscript)
h	hot (subscript)
c	cold (subscript)
m	mean (subscript)
s	solid (subscript - stack material properties)

THERMOACOUSTIC EQUATIONS FOR IDEAL GASES

Normalization Constants

$N_P = p_o = p_m(p_o/p_m)$	dynamic pressure (p_m, & p_o/p_m are global constants)
$N_U = (1/\gamma)(p_o/p_m)A_T a_I$	volume velocity (A_T is the area of the initial tube bore)
$N_T = T_{mI}$	mean temperature in terms of initial temperature
$Nx = 1/k_I$	x position
$N_r = r_{wI}$	inner radius of tube in terms of initial radius
$N_y = \delta_{\kappa I}$	transverse y position in stack channel
$N_\ell = y_o$	stack plate thickness ℓ in terms of plate gap y_o
$N_{PD} = (1/2\gamma)(p_o/p_m)^2 p_m a_I$	power density
$N_P = N_{PD} A_T$	power
$N_Z = N_P/N_U = \gamma p_m/(A_T a_I)$	acoustic impedance

Normalized Variables & Parameters

Note: Bold capital sans serif symbols are normalized (dimensionless)

$\mathbf{P}$	$= p_1/p_o$	dynamic pressure variable
$\mathbf{U}$	$= U_1/N_U$	volume velocity variable
$\mathbf{T}$	$= T_m/T_{mI}$	mean temperature variable
$\mathbf{X}$	$= k_I x$	x position variable
$\mathbf{R}$	$= r_w/r_{wI}$	inner radius variable or parameter
$\mathbf{Y}$	$= y_o/\delta_{KI}$	stack plate gap parameter
$\mathbf{L}$	$= l/y_o$	stack plate thickness parameter
$\mathbf{H}_2$	$= H_2(1+\mathbf{L})(r_{wI}/r_{st})^2/N_P$	stack enthalpy parameter (r_{st} is stack radius)
$\mathbf{K}$	$= (K+\mathbf{L}K_s)k_I T_{mI}/N_{PD}$	longitudinal thermal conduction parameter in stack
$\mathbf{W}_2$	$= \mathrm{Re}(\mathbf{P}\tilde{\mathbf{U}})$	acoustic work power; Not normalized is $W_2 = N_P \mathrm{Re}(\mathbf{P}\tilde{\mathbf{U}})$

Ideal Gas Relationships

For an ideal gas, the thermal expansion coefficient β can be eliminated with,

$T_m\beta = 1$, where T_m is expressed in absolute units.

The sound speed a can be expressed as,

$a^2 = \gamma p_m/\rho_m$, where $a(T_m) = a_I \mathbf{T}^{-1/2}$ gives the temperature dependence.

The gas specific heat c_p is given by the following relation,

$$\rho_m c_p = \left(\frac{\gamma}{\gamma - 1}\right)\frac{p_m}{T_m} .$$

Stack Equations

The following equation is the Rott wave equation modified by Swift for acoustic propagation in channels formed by parallel plates (the stack) where the plates may have a temperature gradient.

$$\left(1+\frac{(\gamma-1)f_\kappa}{1+\varepsilon_s}\right)p_1+\frac{\rho_m a^2}{\omega^2}\frac{d}{dx}\left(\frac{1-f_v}{\rho_m}\frac{dp_1}{dx}\right)-\beta\frac{a^2}{\omega^2}\frac{f_\kappa-f_v}{(1-\sigma)(1+\varepsilon_s)}\frac{dT_m}{dx}\frac{dp_1}{dx}=0$$

$$f_v=\frac{\tanh[(1+i)y_0/\delta_v]}{(1+i)y_0/\delta_v}$$

$$f_\kappa=\frac{\tanh[(1+i)y_0/\delta_\kappa]}{(1+i)y_0/\delta_\kappa}$$

$$\varepsilon_s=\frac{\sqrt{K\rho_m c_p}\tanh[(1+i)y_o/\delta_\kappa]}{\sqrt{K_s\rho_s c_s}\tanh[(1+i)y_o/\delta_s]}$$

$$\sigma=c_p\mu/K=\nu/\kappa$$

$$\delta_\kappa=\sqrt{2\kappa/\omega}$$

$$\delta_v=\sqrt{2\nu/\omega}$$

While this equation is accurate for liquids as an acoustic medium, we will restrict ourselves to gaseous media and use ideal gas relationships. Simplifying the equation with ideal gas relationships and normalized variables, the result is,

$$\left[1+\frac{(\gamma-1)f_\kappa}{1+\varepsilon_s}\right]\mathbf{P}+\left[(1+f_v)+\frac{1}{2}(1+\beta_r)(f_v+\tanh^2\eta_o-1)-\frac{f_\kappa-f_v}{(1-\sigma)(1+\varepsilon_s)}\right]\frac{d\mathbf{T}}{d\mathbf{X}}\frac{d\mathbf{P}}{d\mathbf{X}}+(1-f_v)\mathbf{T}\frac{d^2\mathbf{P}}{d\mathbf{X}^2}=0$$

where $\eta_o = (1+i)(y_o/\delta_v)$, and β_r is defined by Rott so as to express the temperature dependence of the dynamic viscosity μ as $\mu = \mu_I \mathbf{T}^{\beta r}$.

The 2nd order enthalpy (H_2) or energy equation developed by Rott and Swift is,

$$\dot{H}_2 = \frac{\Pi y_0}{2\omega\rho_m}\mathrm{Im}\left[\frac{d\tilde{p}_1}{dx}p_1\left(1-\tilde{f}_v-\frac{T_m\beta(f_\kappa-\tilde{f}_v)}{(1+\varepsilon_s)(1+\sigma)}\right)\right]$$
$$+\frac{\Pi y_o c_p}{2\omega^3\rho_m(1-\sigma)}\frac{dT_m}{dx}\frac{dp_1}{dx}\frac{d\tilde{p}_1}{dx}$$
$$\times\mathrm{Im}\left[\tilde{f}_v+\frac{(f_\kappa-\tilde{f}_v)(1+\varepsilon_s f_v/f_\kappa)}{(1+\varepsilon_s)(1+\sigma)}\right]$$
$$-\Pi(y_0K+lK_s)\frac{dT_m}{dx}$$

(~ denotes complex conjugate)

The energy flow H_2 in a thermally insulated stack is a constant for a steady state solution. This energy constant must either specified, or guessed and solved in the model. The last local state variable for which we need an equation, is the temperature. So the energy equation can be rearranged to express the temperature derivative in terms the energy constant and acoustical variables, instead of the above form. If ideal gas identities and normalized state variables are also used, then resulting equation becomes,

$$\frac{d\mathbf{T}}{d\mathbf{X}}=\frac{\mathbf{T}\,\mathrm{Im}\left[\frac{d\tilde{\mathbf{P}}}{d\mathbf{X}}\mathbf{P}\left(1-\tilde{f}_v-\frac{f_\kappa-\tilde{f}_v}{(1+\varepsilon_s)(1+\sigma)}\right)\right]-\mathbf{H}_2}{\frac{\mathbf{T}}{(\gamma-1)(1-\sigma)}\left|\frac{d\mathbf{P}}{d\mathbf{X}}\right|^2\mathrm{Im}\left[1-\tilde{f}_v-\frac{(f_\kappa-\tilde{f}_v)(1+\varepsilon_s f_v/f_\kappa)}{(1+\varepsilon_s)(1+\sigma)}\right]+\mathbf{K}}$$

Tube Equations

Note: The derivative of the tube radius function may be discontinuous (i.e. the slope or angle of the tube bore), the tube radius function may NOT be discontinuous.

The dimensionless equations are two first order complex equations. (**P** is normalized acoustic pressure & **X** = $k_I x$.)

$$\mathbf{P}'_1 = Q(x)/f_2 \qquad Q' = -f_1 \mathbf{P}_1 ,$$

where $Q(x) = f_2 (d\mathbf{P}_1/d\mathbf{X})$, $f_1 \equiv \mathbf{R}^2_w[1 + (\gamma\text{-}1)f^*]$, $f_2 \equiv \mathbf{R}^2_w(1-f)$, $\mathbf{R}_w \equiv r_w/r_{wI}$,

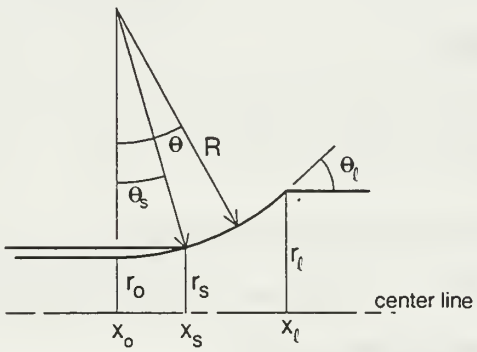
$f = f(\eta_w) = \dfrac{2J_1(i\eta_w)}{i\eta_w J_0(i\eta_w)}$, $f^* = f(\sqrt{\sigma}\,\eta_w)$. Also $\eta_w \equiv (1+i)(r_w/\delta_v)$ and $\sqrt{\sigma}\,\eta_w = (1+i)(r_w/\delta_\kappa)$.

The primes denote derivatives with respect to normalized position. The quantity r_w is the radius at the (inner) tube wall & r_{wI} is the initial or nominal tube radius. $J_n(z)$ are Bessel functions of complex argument. $U_1(x) = (i/\gamma)(\pi r^2_{wI})(p_o/p_m)(aQ)$, or

$$\mathbf{U}_1 = i\,Q.$$

This relates the normalized volume velocity to Q. The Prandtl number is σ, δ_v is the viscous penetration depth, and δ_κ is the thermal penetration depth.

Curved Section Tube Tapers



Now the subscript 's' stands for 'small' at the starting end, & 'ℓ' stands for 'large' at the finishing end, & and 'o' stands for a set of 'fictitious' coordinates that facilitate the solution.

$r(x) = r_o + R(1-\cos\theta)$ so that $r_s = r_o + R(1-\cos\theta_s)$ & $r_\ell = r_o + R(1-\cos\theta_\ell)$.

Subtracting the last two equations we obtain,

$R = (r_\ell - r_s)/(\cos\theta_s - \cos\theta_\ell)$ and $r_o = r_s - R(1-\cos\theta_s)$.

As before, the solution we want has the form

$r(x) = r_o + R\left(1 - \sqrt{1 - \frac{(x - x_o)^2}{R^2}}\right)$, but we don't know the value of x_o. Since

$x_s - x_o = R\sin\theta_s$, we can write $x - x_o = (x - x_s) + R\sin\theta_s$. Thus the solution becomes

$$r(x) = r_o + R\left(1 - \sqrt{\cos^2\theta_s - \frac{(x - x_s)^2}{R^2} - \frac{2\sin\theta_s(x - x_s)}{R}}\right).$$

We still need to know where the solution stops in the x coordinate.

$$x_\ell - x_s = R(\sin\theta_\ell - \sin\theta_s).$$

The final normalized versions are:

$$\mathbf{X}_\ell - \mathbf{X}_s = k_I R(\sin\theta_\ell - \sin\theta_s), \text{ and}$$

$$\mathbf{R}_w = \frac{r_o}{r_{wl}} + \frac{R}{r_{wl}}\left(1 - \sqrt{\cos^2\theta_s - \frac{(\mathbf{X} - \mathbf{X}_s)^2}{(k_l R)^2} - \frac{2\sin\theta_s(\mathbf{X} - \mathbf{X}_s)}{k_l R}}\right).$$

LUMPED ELEMENTS

Rigid Termination

In cases where the acoustic velocity is zero or very small, the thermal conduction at the interface of rigid stationary solid surface (with a large $K_s\rho_s c_s$,) gives an acoustic impedance of,

$$Z_A = \frac{-2}{\sqrt{2i}}\left(\frac{\gamma}{\gamma - 1}\right)\frac{p_m}{ak\delta_\kappa S}.$$

If we cast this as a normalized volume velocity, we obtain,

$$\mathbf{U} = -\tfrac{1}{2}(1+i)(\gamma - 1)\, k_I \delta_\kappa\, \mathbf{P}\, (S/A_T),$$

where S is the solid area exposed.

Small Volume

The impedance of an idealized, acoustically small gas volume is,

$Z_A = i\gamma p_m/\omega V$,

ignoring surface effects, where V is the volume. To include thermal conduction at the rigid surface we use the "rigid termination" solution. The answer is that the total volume velocity is the sum of the idealized volume velocity and the volume velocity at the rigid termination of the wall. Normalized this becomes,

$\mathbf{U}_{SV} = -i\, k_I\, \mathbf{P}\, (V/A_T) - \tfrac{1}{2}(1+i)(\gamma-1)\, k_I \delta_\kappa\, \mathbf{P}\, (S/A_T)$

Capillary

Assume that a given acoustic pressure drives one end of a capillary, and that the dynamic pressure is zero at the opposite end. Also, assume that the length ℓ is very short relative to a wavelength. Then the impedance of the capillary is,

$$Z_A = \frac{\gamma p_m}{\omega}\left(\frac{ik^2\ell}{S}\right)\left[1-\frac{2}{i\eta_o}\frac{J_1(i\eta_o)}{J_0(i\eta_o)}\right]^{-1},$$

where S is the area of the bore of the capillary and $\eta_o = (1+i)(r_o/\delta_v)$ with r_o being the capillary radius. The normalized volume velocity can be expressed as,

$$\mathbf{U} = \frac{i}{k_I \ell}\mathbf{P}\left[1-\frac{2}{i\eta_o}\frac{J_1(i\eta_o)}{J_0(i\eta_o)}\right]\frac{S}{A_T}\mathbf{T}$$

where the temperature dependence has been made explicit. The capillary can be combined with other lumped elements by adding its volume velocity with that of the other component, as was done previously with the small volume combined with its surface conduction effects.

Radiation Impedance

For a vibrating unflanged rigid piston, the radiation impedance (mechanical) is given by,

$$Z_m = \rho_m a\, S\, [¼(kr)^2 + 0.6\, i\, kr], \qquad \text{[Ref. 10: Chap. 9]}$$

where k is the wavenumber, r is the piston radius and S is the piston area. The acoustic impedance in an ideal gas is then,

$$Z_A = Z_m/S^2 = (\gamma p_m/aS)\, [¼(kr)^2 + 0.6\, i\, kr].$$

Since the sound speed and wavenumber are temperature dependent, normalization requires that this dependence be made explicit. The normalized form of the radiation impedance is then,

$$\mathbf{Z_A} = ¼(k_I r_I)^2\, \mathbf{T}^{-3/2} + 0.6\, i\, (k_I r_I/\mathbf{R})\, \mathbf{T}^{-1}.$$

LIST OF REFERENCES

1. Swift, G.W., "Thermoacoustic Engines and Refrigerators," *Physics Today,* pp.22-28, July 1995.

2. Swift, G.W., "Thermoacoustic Engines," *Journal of the Acoustical Society of America*, v.84, p. 1145-1179, 1988.

3. Wheatley, J.C., Swift, G.W., and Migliori, A., "The Natural Heat Engine," *Los Alamos Science,* number 14, pp. 2-29, Fall 1996.

4. Wheatley, J.C., Hofler, T.J., Swift, G.W., and Migliori, A., "Understanding some simple phenomena in thermoacoustics with applications to acoustical heat engines," *American Journal of Physics,* v. 65, No. 2, pp 147-162, February 1995.

5. Wong, K., "Solving Ordinary Differential Equations with Runge-Kutta Methods", June 1996, [http://www.geog.ubc.ca/numeric/labs/lab4/lab4/lab4.html].

6. Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P., *Numerical Recipies in C*, Cambridge University Press, Massachusetts, 1992.

7. Allen, R.C., Avery, P., and Wallace, J.Y., "Lower/Upper Triangular (LU) Decomposition", *Computational Science Textbook*, , Sandia Corporation, 1998, [http://ais.cs.sandia.gov/AiS/textbook/textbook.html].

8. Deitel, H.M., and Deitel, P.J., *How to Program C++*, Prentice Hall, New Jersey, 1998.

9. Pozo, R., MV++ Matrix/Vector Library, Mathematical and Computational Sciences Division, National Institute of Standards and Technology, [http://math.nist.gov/mv++/].

10. Kinsler, John W., Frey, A.R., Coppens, A.B., and Sanders, J.V., *Fundamentals of Acoustics*, 3d ed., John Wiley and Sons, 1982.

INITIAL DISTRIBUTION LIST

No. of copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Dr. Logan E. Hargrove ONR 331 2
   Office of Naval Research
   800 North Quincy Street
   Arlington, VA 22217-5560

4. Professor Thomas J. Hofler, Code PH/Hf 4
   Naval Postgraduate School
   Monterey, CA 93943-5100

5. Dr. Gregory W. Swift 1
   Los Alamos National Laboratory
   MS K764
   Los Alamos, NM 87545

6. Dr. David L. Gardner 1
   Los Alamos National Laboratory
   MS K764
   Los Alamos, NM 87545

7. Dr. Robert Wong 1
   Naval Postgraduate School
   Physics Department
   Monterey, CA 93943

8. Dr. Henry E. Bass 1
   National Center for Physical Acoustics
   2001 NCPA
   University, MS 38677

9. LT Eric Purdy.....1
Strike Figher Squadron 125
Naval Air Station Lemoore, CA 93246

6 483NPG TH 2823
10/99 22527-200 MILE

DUDLEY KNOX LIBRARY
3 2768 00366646 2