SYMBOLIC MANIPULATION AND SUPERSONIC TRANSITION

Gordon Erlebacher
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Hampton, VA 23665

ABSTRACT

Progress in the understanding of the physical mechanisms that underlie supersonic and hypersonic transition phenomena is greatly impeded by the nonlinearity of the Navier-Stokes equations. The convective terms in the momentum equations are cubic nonlinearities, and the temperature dependent viscosity and thermal conductivity greatly complicate the structure of the viscous stress and heat flux terms. When performing series expansions of the flow variables, these nonlinearities quickly render manual computations impractical, particularly if many such calculations become necessary.

In this paper, we describe the use of the Mathematica symbolic manipulation software to efficiently compute matrix elements which arise in weak nonlinear theory. Only the nonlinear terms are treated herein. The algorithm is efficient in that intermediate terms which are usually computed and carried along are never encountered, thus speeding the algorithm by over an order of magnitude. The symbolic code handles arbitrary systems of equations with the restriction that its coefficients be polynomial in the independent variables. Fully nonlinear terms must be Taylor expanded to the appropriate order before the equations are input to the code.

We also present an alternate package for derivatives and a package to generate useful Fortran code from Mathematica in a readable form.

INTRODUCTION

Progress in the understanding of the physical mechanisms that underlie supersonic and hypersonic transition phenomena is greatly impeded by the nonlinearity of the Navier-Stokes equations. The convective terms in the momentum equations are cubic nonlinearities, and the temperature dependent viscosity and thermal conductivity greatly complicate the structure of the viscous stress and heat flux terms. When performing series expansions of the flow variables, these nonlinearities quickly render manual computations impractical, particularly if many such calculations become necessary. This could occur if one wishes to experiment with different coordinate systems, or different sets of equations (for analysis purposes). Examples where computations become extremely tedious include secondary instability and weakly nonlinear theory. In the latter theory, the primitive variables are expanded in a Fourier series whose coefficients are themselves expansions in the (supposedly small) fundamental wave amplitude.

Although symbolic algebra can systematically reduce the number of errors incurred by manual computation, the problems alluded to above still require several hours of CPU time to complete. This amount of time is still impractical for serious experimentation. However, the actual number of interaction terms which appear in the final equations are often a small subset of the total number of terms generated during the intermediate calculations. For example, if we multiply two third-order polynomials together, the result is a sixth-order polynomial, although we might only wish to keep terms to third-order. A substantial fraction of the computation is redundant.

In this paper, we describe a tensorial approach, implemented in Mathematica which eliminates the unnecessary intermediate terms. The symbolic code handles arbitrary systems of equations with the restriction that its coefficients be polynomial in the independent variables. Fully nonlinear terms must be Taylor expanded to the appropriate order before the equations are input to the code. The technique is applied to the study of weakly nonlinear processes in supersonic boundary-layer flows. These are governed by a system of five nonlinear equations and five dependent variables. The solution process involves the determination of Landau constants which govern the time evolution of the fundamental wave amplitude. The technique presented here reduces the time required to determine the discrete equation matrix elements by almost two orders of magnitude, compared to the expense of a straightforward expansion procedure. Moreover, the algorithm makes no assumption about the order of the expansion.

The purpose of this paper is not to discuss the physics of transition, but rather to describe some of the symbolic manipulation techniques which have led to the efficient implementation of the weakly non-linear theory applied to the fully compressible Navier-Stokes equations.

Papers on symbolic manipulation which relate to practical problems rarely discuss the actual symbolic procedures which led to the solution of the problem at hand, but rather simply discuss the solution technique and state that the problem was solved symbolically. However, once the analytical algorithm has been established, there are a myriad of ways to actually implement it. Often, a straightforward implementation is much more expensive and tedious than an approach which exploits properties of both the physical problem and of the symbolic manipulation package. We describe the Mathematica routines which solve the weakly nonlinear problem in transition. To this effect the paper is divided into several sections which will lead the reader from the simplest setting to the most general framework we are considering. In the second section the problem is described in the context of compressible transition. The justification of the formulation is kept brief, and relevant references are given for further details. In the third section, the problem is reduced to the study of the weakly non-linear evolution of a single wave which arises from a system of evolution equations. The base equation will be a nonlinear equation with quadratic linearities of a single variable. From the numerous Mathematica code fragments in the text, it should prove easy to generalize the concepts to higher spatial dimensions, multiple waves, and more complicated systems of equations.

Because the system of equations that are handled are rather complex, and the resulting discrete system is very large, it is beneficial to allow Mathematica to generate the fortran routines that calculate these matrix elements. We, therefore, list in the appendix a general package which can output the contents of an arbitrary-sized matrix, along with several other useful options. This code is listed in Appendix B, along with the derivative package used for this work (Appendix A).

FORMULATION

The study of transition is usually based on the solution to the compressible Navier-Stokes equations. In non-dimensional form these are

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0, \\ \frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) &= -\frac{1}{\gamma M_R^2}\nabla P + \frac{1}{Re}\nabla \cdot \vec{\boldsymbol{\sigma}}, \\ \frac{\partial P}{\partial t} + \mathbf{u}\cdot \nabla P + \gamma P \nabla \cdot \mathbf{u} &= \frac{\gamma}{Re Pr}\nabla \cdot \kappa \nabla T + \frac{\gamma(\gamma-1)M_R^2}{Re}\,\Phi, \end{aligned} \tag{1}$$

where

$$\vec{\boldsymbol{\sigma}} = 2\mu\left[\frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T) - \frac{1}{3}(\nabla \cdot \mathbf{u})\,\vec{\mathbf{I}}\right] \tag{2}$$

is the viscous stress tensor (with bulk viscosity assumed zero), $\vec{\mathbf{I}}$ is the identity tensor, and

$$\Phi = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T) : \vec{\boldsymbol{\sigma}} \tag{3}$$

is the viscous dissipation function. The equation of state is

$$P = \rho T. \tag{4}$$

The density, velocity, temperature and pressure are, respectively

$$\begin{aligned} \rho &= \frac{\rho^*}{\rho_R}, \\ \mathbf{u} &= \frac{\mathbf{u}^*}{U_R}, \\ T &= \frac{T^*}{T_R}, \\ P &= \frac{p^*}{R_g \rho_R T_R}. \end{aligned}$$

where the dimensional reference values are denoted by the subscript R and R_g is the universal gas constant. Dimensional variables have an asterisk superscript. Distance and time are scaled, respectively, with respect to L_R and $t_R = L_R/U_R$. In the text, $\mathbf{x}$ refers to the Cartesian position vector.

The Prandtl number $Pr = (\mu_R C_p)/\kappa_R$, the Reynolds number $Re = (\rho_R U_R L_R)/\mu_R$; γ is the ratio of specific heats. Viscosity and conductivity are denoted by μ and κ, respectively, and are scaled with respect to μ_R and κ_R. In general, viscosity and temperature are nonlinear functions of the temperature. Usually Sutherland's law

$$\mu(T) = T^{1.5}\,\frac{1 + C/T_R}{T + C/T_R} \tag{5}$$

is adopted for the viscosity (where $C = 199^\circ R$). The Prandtl number Pr is usually fixed at 0.7 for air. In this paper, we assume the viscosity and Prandtl numbers to be arbitrary analytic functions of temperature in the temperature range of interest.

Much work has been devoted to the study of the eigenvalues and eigenfunctions of the Navier-Stokes equations, linearized about some suitable mean flow: see Ref. 3 for linear stability and Ref. 2 for secondary instability formulations. For flat plates, the mean flow is often the solution to the similarity equations. If similarity equations do not exist, boundary-layer or parabolized Navier-Stokes codes can often provide the required steady flow.

Usually, neglect of the normal mean velocity still provides accurate stability characteristics if spanwise curvature and crossflow effects are not considered. These velocities are usually taken to be functions of only the normal coordinate (y). (For the flat plate, this coordinate is normal to the plate). The x coordinate is reserved for the direction of the mean flow, and the third direction (spanwise) is labeled z. Let

$$\mathbf{u} = (u, v, w)^T \tag{6}$$

be the velocity vector, and

$$\mathbf{Q} = (\rho, \mathbf{u}, T)^T \tag{7}$$

be the solution vector to the compressible Navier-Stokes equations. Linear eigenfunctions $\mathbf{q}(\mathbf{x})$ can be obtained by substituting

$$\mathbf{Q}(\mathbf{x}) = \mathbf{Q}_m(y) + A\mathbf{q}(y)e^{\imath(\alpha x - \omega t)} \tag{8}$$

into the Navier-Stokes equations, and linearizing the equations with respect to A. The resulting system is homogeneous with homogeneous boundary conditions

$$\mathbf{u} = T = 0, \qquad y = 0 \tag{9}$$

$$\mathbf{u}, T \to 0, \qquad y \to 0. \tag{10}$$

If a linearized solution (primary or secondary) is not sufficient, the next approximation seeks nonlinear corrections to these solutions in the form of a nonlinear evolution equation (given by a Landau equation) for the primary wave amplitude. This Landau equation can indicate whether the wave is subcritical or supercritical, and in cases where validity of the amplitude expansions can be ascertained, the equation can calculate the presence or absence of saturated states. Weakly nonlinear theory also allows for the interaction of multiple waves at the expense of increased complexity in the computation. The final result is then a set of couple Landau equations which, for small amplitudes, simply returns the linearized growth rates of each individual wave. A brief description of the method is now given.

Symbolically, we wish to solve a nonlinear system of equations

$$\mathbf{N}(\mathbf{Q}) = 0 \tag{11}$$

where $\mathbf{N}$ represents a system of nonlinear operators. If $\mathbf{LQ}$ is the linear system which results after linearization about the basic state $\mathbf{Q}_m(y)$, one has

$$\mathbf{N}(\mathbf{Q}) = \mathbf{LQ} - \mathbf{R}(\mathbf{Q}) = 0. \tag{12}$$

The system of equations $\mathbf{LQ} = 0$ admits a spectrum of eigenvalues and their associated eigenfunctions can be computed. These eigenfunctions are called primary waves. Because the problem is now linear, the amplitude A of the primary wave is indeterminate. However, it is often desirable to know what happens when this wave is allowed to interact with itself within the constraints of the original system of equations. Thus, by inserting $\mathbf{Q}$ into $\mathbf{R}(\mathbf{Q})$ produces quadratic and higher-order nonlinearities. In addition, correction terms to the mean flow appear due to the interaction of the primary wave with its complex conjugate. It is therefore useful to expand the solution vector $\mathbf{Q}$ in the series

$$\mathbf{Q} = \mathbf{Q}_M(y) + \sum_{n=-\infty}^{\infty} \mathbf{Q}_n(y,t)\, e^{\imath n(\alpha x - \zeta(t))}. \tag{13}$$

The system of partial differential equations obtained after substitution in the Navier-Stokes equations is fully coupled and extremely difficult to solve efficiently. Weak non-linear theory is concerned with solutions of this system of equations which depart only slightly from linearity. The theory is fully described in Refs. 5, 6 and 1. Details and justification of various expansions, restrictions and approximations are omitted here. The interested reader is referred to the aforementionned references. To make the system of equations, Eqs. 13 tractable, an ordering must be established. If we assume that the higher-order terms in the expansion are solely due to interactions of the primary wave with itself and its harmonics, it is natural to rewrite $\mathbf{Q}$ as

$$\mathbf{Q}_n(y,t) = A(t)^n \mathbf{q}_n(y,t) \tag{14}$$

where $A(t)$ is the time-dependent real amplitude of the fundamental wave. The growth of the primary wave is completely incorporated in its amplitude function $A(t)$. Its phase is $\zeta(t)$. Further simplification is possible by expanding $\mathbf{q}(y,t)$ as a power series in $A(t)$:

$$\mathbf{q}_n(y,t) = \sum_{m=0}^{\infty} A(t)^{2m} \mathbf{q}_{n,m}(y) \tag{15}$$

under the assumption that $\mathbf{q}(y,t)$ departs only slightly from its linear shape (in the limit $A \rightarrow 0$). It can be shown that all terms with odd powers of A vanish. Time derivatives of $\mathbf{Q}(y,t)$ require the evaluation of $dA(t)/dt$ and $d\zeta(t)/dt$. These are also defined as a power series in $A(t)$:

$$\begin{aligned} \frac{1}{A(t)}\frac{dA(t)}{dt} &= \sum_{n=0}^{\infty} A^{2n} a_n \\ \frac{d\zeta(t)}{dt} &= \sum_{n=0}^{\infty} A(t)^{2n} g_n . \end{aligned}$$

The term $\mathbf{q}_{0,0}$ is set to zero since it represents the mean flow already taken into account by $\mathbf{Q}_M(y)$. The coefficients a_n and g_n are called Landau coefficients. In practice we are concerned with the first Landau coefficients a_1 and g_1; more particularly, the former which provides information on the nonlinear growth of the wave. Substituting E for $e^{\iota(\alpha x - \zeta(t))}$ one finally obtains

$$\mathbf{Q} = \mathbf{Q}_M(y) + \sum_{n=-\infty}^{\infty} \sum_{m=0}^{\infty} \mathbf{q}_{nm}(y)\, A(t)^{|n|+2m} E^n \tag{16}$$

for the solution vector. If $A(t) << 1$, one obtains a hierarchy of equations by grouping terms of order $A^k E^l$ together and setting these groupings to zero. Here, $k \in [0,\infty]$ and $l \in [-\infty,\infty]$. At each level, the system reduces to the form

$$\mathbf{L}_{kl}\mathbf{Q}_{kl} = \mathbf{R}_{kl} \tag{17}$$

which is simply the coefficient of $A^k E^l$ in the system of equations. The objective of this paper is to efficiently determine the right-hand side $\mathbf{R}_{kl}$. The linear operator $\mathbf{L}_{kl}$ is not considered here because its evaluation is cheap and straightforward.

We now describe the symbolic algorithm which is based on Mathematica. It is assumed that the reader has a working knowledge of the system since some advanced features of Mathematica are used. Only features and concepts not normally used in simple symbolic programs will be explained. Details of Mathematica can be found in Ref. 7.

IMPLEMENTATION

Consider the equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = u_{xx} + u_{yy}. \tag{18}$$

This equation contains all the elements essential to the comprehension of the symbolic algorithm. After presentation of the algorithm, various generalizations will be introduced.

Before introducing the actual symbolic algorithm, consider a method only slightly better than brute force. If only the first Landau coefficients are required, only terms to order A^3 must be retained in the expansions. Therefore, $n \in [-2,2]$ and $m \in [0,1]$ and the series which defines $\mathbf{Q}(\mathbf{x},t)$ has in general nine non-zero terms. Quadratic interactions lead to series of 81 terms, while cubic interactions (present in the compressible Navier-Stokes equations) lead to 729 terms. It is clear, however, that most of these terms play no role in the determination of the Landau coefficients. This is easily seen for the quadratic case. If u^2 is rewritten as a power series in A, the exponent of A ranges from 0 to 6, while only terms to order A^3 are required. It is immediately evident that the computation of the coefficients in the amplitude series of u^2 is more expensive than is required. In reality, the extra cost is much more severe since quadratic terms might occur several times in the system of equations studied, and the series recomputed every time these nonlinearities occur. A better approach is to recognize that all quadratic terms have the same structure in terms of their representative power series. If the expense of computing the required terms is incurred only once, a substantial gain in CPU time is achieved. This will be accomplished through the use of a Mathematica function definition. Further gains in time are achieved in the evaluation of this function. If the summation limits on the power representation of u^2 are such that only the non-zero terms are included in the sum, the extraneous terms will not have to be dealt with at a later stage of the algorithm. Further issues to be considered are the spatial (x) and time (t) derivatives.

Recognizing that Mathematica's simplification routines are very intensive in the amount of CPU time they consume, it is a rule of thumb that simplifications should mostly be done in a controlled fashion by the user to keep the symbolic expressions as small as possible at each step of the algorithm. To illustrate the point, consider the computation of the x derivative of u. Computing

$$\frac{\partial u(\mathbf{x})}{\partial x} = \frac{\partial}{\partial x} \sum u_n(y) e^{\iota(\alpha x - \omega t)} \tag{19}$$

is more expensive than the evaluation of

$$\frac{\partial u(\mathbf{x})}{\partial x} = \frac{\partial}{\partial x} \sum u_n(y) E^n \tag{20}$$

with the auxiliary rule

$$\frac{\partial E}{\partial x} = -\iota \alpha E. \tag{21}$$

First there are many less terms to compute (this becomes more evident as the number of terms in the series increases, and when the expressions for the coefficients of E becomes more complex). Second, Eq. 21 will have a simpler final form than Eq. 20. Although several simplification steps have been stated, there is one more which deserves mention and which will form the crux of the algorithm. Why retain the exponential term E and the amplitude power terms (A^n) if they are to be eliminated in the final stages of the calculation? It would be far better if a method could be devised which dispensed with them from the beginning. Although this is not always possible, it so happens that this particular problem allows it.

The implementation of the symbolic algorithm is presented through the use of actual Mathematica code fragments. This will hopefully aid researchers in similar lines of research. It is the author's experience that papers on symbolic methods often concentrate on the physics of the problem rather than on the symbolic algorithms themselves. Such papers are best published in physics-oriented journals.

Rather than specialize the method to the Navier-Stokes equations, the objective is to treat the weak nonlinear problem in a more general context. Therefore, the equation description file is isolated from the remainder of the program. The weakly nonlinear module is separate and pretty much independent of the system of equations solved. To this end, general notation must be established. After writing out the equations in mnemonic form and storing them in an array called `eqlst`, the mnemonic primitive flow variables `u,v,rho,T` are replaced by

```
u = vtot[1];
v = vtot[2];
rho = vtot[3];
T = vtot[4];
```
(22)

For simplicity, we only consider the two-dimensional problem, although the extension to three-dimensions is straightforward. To insure that derivatives are calculated properly, we state the required dependencies with respect to the independent variables:

```
depends[{utot[_]},{x,y,t}];
depends[{umean[_]},{y}];
depends[{upert[_]},{x,y,t}];
depends[{A},{t}];
```
(23)

(see Appendix A for the dependency routine package). Once the substitution, Eq. 22, is made, the equations are solely a function of `vtot` and its spatial and temporal derivatives. The total number of equations and variables are given by

```
nequ = Length[eqlst];
nvar = nequ;
```
(24)

Next, `vtot[i_` is decomposed into mean and fluctuating components according to

```
utot[i_] := umean[0,i] + upert[0,0,i];
eqlst = Expand[eqlst];
```
(25)

and the system of equations expanded. Here, `umean[n,i]` refers to the n^{th} derivative of the mean flow with respect to y, and

$$\texttt{upert}[\texttt{n},\texttt{m},\texttt{i}] = \frac{\partial^n \partial^m}{\partial x^n \partial y^m}\,\texttt{upert}[0,0,\texttt{i}] \tag{26}$$

for the general case of three-dimensional flows. Next, the terms which involve the derivative operator `DD` are removed in anticipation of the Fortran code produced at the final stages of the algorithm. (DD is the first order derivative package written by the author. The package which defines it is listed in Appendix A.) These substitution rules are

```
rulesDD = {
    DD[umean[n_,i_],y]     :> umean[n+1,i],
    DD[upert[n_,m_,i_],x]  :> upert[n+1,m,i],
    DD[upert[n_,m_,i_],y]  :> upert[n,m+1,i],
    DD[upert[n_,m_,i_],t]  :> ut[n,m,i]
};
```
(27)

This set of rules allows for derivatives with respect to x, y, z of arbitrary order of the mean flow and perturbation variables. Of course, these derivatives appear in the constitutive equations under study. Also note that the actual dependence of `upert` on the independent variables is still unspecified. The time derivatives are defined slightly differently. Only single time derivatives are considered, which explains why a special name is reserved (`ut`). Spatial derivatives of the time derivative terms are allowed, although this feature is not used in the study of the Navier-Stokes equations in the primitive variable formulation. It is always a good idea to simplify expressions as the calculation proceeds instead of waiting to the end. This results in a sizeable savings in compute resources (both memory and CPU time). In this case, the simplifications take the form of simpler notation. The rule set `rulesDD` is next applied recursively to the equations to simplify second- and higher-order derivatives:

```
eqlst = eqlst //.  rulesDD;                                        (28)
```

As explained earlier, only the nonlinear terms that are ultimately required should be computed. We now show how this is accomplished. Consider the product of two perturbation quantities $\tilde{u}$ and $\tilde{v}$:

$$\tilde{u}\tilde{v} = A^{|n|+|p|+2m+2q} E^{n+p} u_{n,m}\, v_{p,q}. \tag{29}$$

where the repeated summation is assumed. Letting

$$\begin{aligned} r &= n+p \\ s &= |n|+|p|+2m+2q, \end{aligned}$$

$\tilde{u}\tilde{v}$ becomes

$$\tilde{u}\tilde{v} = u_{n,m}\, v_{r-n,(s-|n|-|r-n|)/2-m}\ A^s\, E^r \tag{30}$$

which leads to the simpler notation

$$uv = (uv)_{s,r}\, A^s\, E^r. \tag{31}$$

Note that the coefficient of $A^s\, E^r$ in Eq. 30 is a convolution of u and v over the appropriate range (see definition of the `list` operator below). To simplify notation further, this convolution is denoted by `convol` which takes an arbitrary number of arguments. With this notation,

$$\begin{aligned} \tilde{u} &= \mathtt{convol}[\tilde{u}][s,r]\ A^s\, E^r \\ \tilde{u}\tilde{v} &= \mathtt{convol}[\tilde{u},\tilde{v}][s,r]\ A^s\, E^r \\ \tilde{u}\tilde{v}\tilde{w} &= \mathtt{convol}[\tilde{u},\tilde{v},\tilde{w}][s,r]\ A^s\, E^r. \end{aligned} \tag{32}$$

Next, all perturbation variables are replaced by the convolution operator with the rule

```
rule1 = { upert[i_] :> convol[u[i]] };                             (33)
```

The equations now only contain sums of products of convolutions which must be reduced to sums of single convolutions according to Eq. 32.

The rules that define the `convol` operator are

```
rulesconvoldef = {
   Power[a_convol, n_Integer?positive] := Block[{i},
     Apply[Times, Table[a,{i,1,n}]]],                              (34)
   Times[convol[x_], convol[y_]] := convol[x,y]
};
```

The first function transforms powers of the convolution operator by multiplication, while the second function explicitly implements the rules of Eq. 32. The system of equations eqlst is now in a useful form. For example,

$$\mathrm{eqlst}[[1]] = \sum_{r,s} cst_i \mathtt{convol}[\cdots][r,s] A^r E^s. \tag{35}$$

Since the factor $A^r E^s$ appears in every term, it is never introduced in the symbolic code.

So far, the spatial dependence of the perturbation variables has not been taken into account. This is now done by remembering that

$$E = e^{\iota(\alpha x - \omega t)} \tag{36}$$

so that partial derivatives with respect to x are calculated according to the rule

```
rulex = {
    upert[n_,m_,i_][r_,s_] :>
       alpha^(n r) I^(n r) u[m,i][r,s]
};
```
(37)

Note the use of the wave number arguments $[r, s]$ which are now necessary because the derivative depends on them. Always implicit is the presence of the $A^r E^s$ factor.

The time derivative of

$$u = A^{|n|+2m} E^n u_{nm} \tag{38}$$

is

$$\frac{\partial u}{\partial t} = A^{|n|+2m} E^n u_{nm}[(|n| + 2m)a_k A^{2k} - \iota n w_k A^{2k}]. \tag{39}$$

The objective is to rewrite this expression with $A^{|n|+2m} E^n$ as the leading factor. Substituting $l = 2m + 2k$ leads to

$$\frac{\partial u}{\partial t} = A^{|n|+2l} E^n u_{n,l-k}[(|n| + 2(l - k))a_k - \iota n \omega_k] \tag{40}$$

where $k \in [0, \infty]$. To the lowest order in the Landau expansion, terms of $O(A^4)$ are neglected, so that in practice, $k \in [0, 1]$. Note that no range is required for n and l since we are only interested in the coefficients pertaining to a particular (n, l) wave. In Mathematica, Eq. 40 translates to

```
ruleut={
    ut[i_][n_,m_] :>
       Block[{},
          Sum[((Abs[n]+2(m-jamp)) ac[1,jamp] - I n gc[1,jamp]),
          {jamp,0,3}] /.  rules11
       ]
    }
};
```
(41)

where `rules11` is a set of rules designed to continuously eliminate zero terms as they occur. These rules are

```
rules11=u[x_][n_,m_] :> 0 /; m < 0 || m > 3 || Abs[n]+2m > 3;
```
(42)

The third condition insures that all terms of order $o(A^3)$ are removed.

The rules that pertain to the `convol` operator are

```
rulesconvol= {
    convol[a__][r_,s_] :> 0 /; Length[{a}] > r,
    convol[a_][p__] :> a[p],
    convol[a_,b_][p__] :> lists[p][2][a,b]
};
```
(43)

Some explanations are necessary at this stage. The wave numbers $[r, s]$ now refer to the coefficient of $A^r E^s$. Therefore, the first rule states that the order of convolution operator (i.e., the number of arguments in the first slot of `convol`) cannot be greater than the exponent of A in the wave we are treating. The second rule is an identity since the convolution of a single variable is the variable itself. The third rule treats quadratic nonlinearities. Rather than recomputing the convolution sum every time it is required, it is far more efficient to recognize that this sum is the same whatever the arguments for `convol`, and compute this sum once. The arguments of the function `lists` are, respectively, the wave numbers (`p`), the order of the nonlinearity (two for quadratic), and the variables to convolve (`a` and `b`). We only list the quadratic case herein. The definition of `lists` is

```
list[r_,s_][2] := Block[{'i,'j},
    Release[
      Sum[
        #1[i,j] #2[s-i,(r-Abs[i]-Abs[s-i])/2-j,
        {i,-r,r},
        {j,0,(Abs[r]-Abs[i]-Abs[s-i])/2}] /.  rInteract
      ]&
    ];
];
```
(44)

The rule [rInteract] sets the mean flow components and any variable with a rational argument (integer fraction) to zero:

```
rInteract={
    #1[0,0]->0, #2[0,0]->0,
    #1[___,_Rational,___] :> 0,
    #2[___,_Rational,___] :> 0,
};
```
(45)

Several advanced features of Mathematica have been used. First, in `rInteract`, note the triple underscores (zero or more arguments). Their presence before and after the `_Rational` arguments guarantees that if a rational argument appears in any position, the variable is set to zero. Returning to `list`, one notes the absence of the third set of arguments (the variable names) in its definition (compare with `rulesconvol`). The function `list` is actually a pure function of two arguments which are the perturbation variables `upert`. The `Release` operator forces the `Sum` operator to be expanded inside the definition of `list`, rather than each time it is used. The function `list` is defined with the `:=` operator which indicates it is reevaluated each time it is called. However, it only differs when the wave numbers are modified. Therefore, we define a wave number list which in this problem is

```
wavelist={{1,0}, {0,1}, {2,0}, {1,1}};
```
(46)

(The component `{r,s}` refers to the wave characterized by $A^r E^s$.) A new operator, `lists`, is now defined for each wave that is considered:

```
Do [
    lists @@ wavelist[[i]])[2] = (list @@ wavelist[[i]])[2],
    {i,Length[wavelist]}
];
```
(47)

Because of the `=` sign, `lists` is not a function definition, but is in fact the actual expanded summation defined in `list`. This process explains why it was necessary to define the arguments of list as three separate sets rather than put them all in one set. The latter alternative would have prevented the definition of an operator which takes the wavenumbers as arguments.

We now compute the required right hand side. First, one must extract from the original system of equations the nonlinear terms. This is done by

```
eqlst = eqlst /.  vtot[i_]:> eps vtot[i];
eqlst = -Sum[Coefficient[eqlst,eps^i],{i,2,3}] //Expand;
```
(48)

if one assumes that there are only quadratic and cubic nonlinearities. Next, one applies the sequence of predefined rules according to

```
eqlst = eqlst /.  rule1 //.  rulesconvoldef;
```
(49)

The set of equations now has terms of the form

```
convol[r,s][upert[n,m,i],upert[p,q,j]];
```
(50)

We now specialize to specific wave numbers by defining a variable `rrhs[i]`:

```
Do [
   rrhs[i] = rhs /.  convol[y__] :> (convol[y] @@ wavelist[[i]]);
   rrhs[i] = rrhs[i] /.  rulesconvol /.  rulex /.  ruleut,
{i,Length[wavelist]}];
```
(51)

Finally, the Fortran code is generated with

```
Do [
   wrr[rrhs[i] /.rulesvar, "arhs",
     StringJoin["rhs", ToString[i],"_wnl.h"], lowestIndex->"j,"],
{i,Length[wavelist]}];
```
(52)

where

```
rulesvar = {u[n_,i_][r_,s] :> v[j,i,n,r,s]};
```
(53)

is necessary to insure proper translation into Fortran (the translator cannot recognize structure of the form u[i][j]). The `StringJoin` operator creates files with names "rhs1_wnl.h, rhs2_wnl.h, ..." which, respectively, contain the Fortran source for waves numbers `wavelist[[i]]`, where `i` ranges from 1 to `Length[wavelist]`. These files are included in a skeleton Fortran code using the C language preprocessor "cpp". This is done within the Mathematica code by escaping to the shell. The `wrr` operator is defined in Appendix B. Its second argument is the left-hand side of the Fortran statement, while the last argument specifies the j^{th} node in the vertical direction in the numerical solution procedure (not discussed here). More precisely, although `rhs` is a one dimensional array, the Fortran statements are of the form

```
arhs(j,1) = ...
arhs(j,2) = ...
```
(54)

CONCLUSIONS

In summary, we have presented one possible implementation (using Mathematica) of a weak nonlinear formulation of a general system of equations. This work was motivated by the increased complexity of the fully compressible Navier-Stokes equations relative to its incompressible counterpart. The paper concentrated mostly on the actual Mathematica implementation rather than on the physics (which has been described elsewhere for incompressible flows).

Actual numerical results pertaining to supersonic transition are not yet available, and will be the subject of a future report.

ACKNOWLEDGEMENTS

The author wishes to thank Bart Singer for numerous discussions during the implementation phase of this algorithm.

REFERENCES

1. Herbert, T., "On Perturbation Methods in Nonlinear Stability Theory," Journal of Fluid Mechanics, Vol. 126, 1983, pp. 167-186.

2. Herbert, T., "Three-Dimensional Phenomena in the Transitional Flat-Plate Boundary Layer," AIAA paper AIAA-85-0489 presented at the AIAA 23rd Aerospace Sciences Meeting, Reno, Nevada, 1985.

3. Mack, L. M., "Boundary-Layer Stability Theory," Document 900-277, Rev. A, Jet Propulsion Lab., Pasadena, CA, 1969.

4. Maeder, R., Programming in Mathematica, Addison-Wesley, New York, 1989.

5. Stuart, J. T., "On the Nonlinear Mechanics of Wave Disturbances in Stable and Unstable Parallel Flows, Part 1 - The Basic Behavior in Plane Poiseuille Flow," Journal of Fluid Mechanics, Vol. 9, 1960, pp. 353-370.

6. Watson, J., "On the Nonlinear Mechanics of Wave Disturbances in Stable and Unstable Parallel Flows. Part 2 - The Development of a Solution for Plane Poiseuille Flow and for Plane Couette Flow," Journal of Fluid Mechanics, Vol. 9, 1960, pp. 371-389.

7. Wolfram, S., Mathematica: A System for Doing Mathematics by Computer, Addison Wesley, New York, 1988.

APPENDIX A: DERIVATIVE PACKAGE

```
(*
 *  Mimic dependency function of Macsyma
 *)

dependP=True;

Apply[BeginPackage,Prepend[$ContextPath,"Depend`"]];
(*BeginPackage["Depend`"];*)

depend::usage = "depend[a_,b_] returns True if a depends on b, False otherwise"
depends::usage = "depends[a_List,b_List] defines dependencies of a[[i]] with
    respect to variables b[[j]]"
undepends::usage = "undepends[a_List,b_List] removes dependencies of a[[i]]
with respect to variables b[[j]]"
```

```
Begin["`private`"];

depend[a_,b_]=False

undepends[a_List,b_List]:=Block[{i,j,la,lb,tmp},
   la=Length[a];
   lb=Length[b];
   Do [
  Do[
 depend[a[[i]],b[[j]]]=False;
  ,{i,1,la}];
   ,{j,1,lb}];
];

depends[a_List,b_List]:=Block[{i,j,la,lb,tmp},
   la=Length[a];
   lb=Length[b];
   Do [
  Do[
 depend[a[[i]],b[[j]]]=True;
  ,{i,1,la}];
   ,{j,1,lb}];
];
End[];
EndPackage[];

derivativeP=True;
```

```
(*
 *  It is assumed that x is an atomic variable.  The package does
 *  not check that this is so (inexperience of the author).
 *)

Apply[BeginPackage,Join[{"Derivative`","Depend`"},$ContextPath]];
(*BeginPackage["Derivative`","Depend`"];*)
DD::usage = "DD[a_,x_] is the derivative of a with respect to x";

SetAttributes[DD,Listable];

DD[x_,x_]:=1;
DD[x_^n_,x_]:=n x^(n-1)  /; FreeQ[n,x];
DD[a_+b_,x_]:=DD[a,x]+DD[b,x];
DD[a_ b_,x_]:=a DD[b,x] + b DD[a,x];
DD[a_^n_,x_]:=n a^(n-1) DD[a,x];
DD[a_,x_]:=0 /; Not[depend[a,x]] && FreeQ[a,x] && AtomQ[a]

Unprotect[Sin,Cos,Tan];
```

```
Sin/:DD[Sin[a_],x_]:= Cos[a] DD[a,x];
Cos/:DD[Cos[a_],x_]:=-Sin[a] DD[a,x];
Tan/:DD[Tan[a_],x_]:=(1/Cos[a]^2) DD[a,x];
Protect[Sin,Cos,Tan];

DD[sinh[a_],x_] := cosh[a] DD[a,x];
DD[cosh[a_],x_] := sinh[a] DD[a,x];
DD[tanh[a_],x_] := (1/cosh[a]^2) DD[a,x];

EndPackage[];
```

APPENDIX B: FORTRAN GENERATION

```
BeginPackage["Fortran`"]
wrr::usage=""
lowerLimit::usage="lower array bounds for array to be Fortran'ed"
lowestIndex::usage="additional ind-ex/ices prior to first array index"
highestIndex::usage="additional ind-ex/ices after last array index"
sublist::usage=

Begin["`private`"]

Unprotect[Flatten];
Flatten[x_,0] := x;
Protect[Flatten];

Options[wrr] = {lowerLimit->0, lowestIndex->"", highestIndex->""};

wrr[x_List,xx_String,out_String,opts___Rule]:=Block[
   {lg,lowIndex,highIndex,xtmp,xflat,prompts,nbelem,index,i,j,dim},

   index=Flatten[
       Outer[List,Sequence @@ Range[Dimensions[x]]],
       TensorRank[x]-1 ];
   nbelem=Times @@ Dimensions[x];
(*
 *  stringjoin necessary.  If ToExpression appears, xx may get replaced
 *  by its value defined before
 *)
   highIndex=highestIndex /. {opts} /. Options[wrr];
    lowIndex= lowestIndex /. {opts} /. Options[wrr] ;
   prompts=Map[ StringJoin[
      xx,"(",lowIndex,
      StringJoin @@ Take[#-(lowerLimit /. {opts} /. Options[wrr]) //
      ToString//Characters,{2,-2}],highIndex,")"
                 ]&,
           index];
```

```
   (*xflat=Map[(#//FortranForm//ToString)&, Flatten[x]];*)
   xflat=Map[#&, Flatten[x]];

   OpenWrite[out, FormatType->OutputForm];
   i=.;
   Do [
      If [Head[xflat[[i]]] === sublist,
         lg=Length[xflat[[i]]]-1;
         forOut[StringJoin[prompts[[i]],"=",
            xflat[[i,1]]//FortranForm//ToString],out];
         lg=Length[xflat[[i]]]-1;
         If [lg > 0,
            Do [
               forOut[StringJoin[prompts[[i]],"=",prompts[[i]]," + ",
                  xflat[[i,j+1]]//FortranForm//ToString],out]
            ,{j,1,lg}]
         ],
 If [xflat[[i]] =!= 0 && xflat[[i]] =!= 0.,
            forOut[StringJoin[prompts[[i]],"=",
               xflat[[i]]//FortranForm//ToString],out]
 ]
  ]
   ,{i,nbelem}];
]

forOut[str_,file_]:=Block[
   {i,beg=1,end=0,mult,prompt="      ",strlg=StringLength[str],
    strlst=Characters[str]},

(* extract positions of all multiplies  *)

   end = Min[64,strlg];
   While [strlg > 64,
      For[end, strlst[[end]] != " " && strlst[[end]] != "*" ,
          end--, Continue[]];
      Write[file,StringJoin[prompt,StringJoin @@ Take[strlst,{beg,end}]] ];
      prompt="     >  ";
      strlg -= end-beg+1;
      beg=end+1;
      end += Min[64,strlg]
   ]
   Write[file,StringJoin[prompt,StringJoin @@ Take[strlst,{beg,end}]] ]
];

End[]
EndPackage[]
```