The Use of Swirl to Clean Nuclear Rocket Plumes

by

David Younghee Oh

S.B. Aeronautics and Astronautics, Massachusetts Institute of Technology
S.B. Music, Massachusetts Institute of Technology

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

Aeronautics and Astronautics

at the

Massachusetts Institute of Technology

May 1993

Signature of Author _____
Department of Aeronautics and Astronautics
May 5, 1993

Certified by _____
Professor Daniel Hastings
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by _____
Professor Harold Y. Wachman
Department Graduate Committee

THE USE OF SWIRL TO CLEAN NUCLEAR ROCKET PLUMES

by

David Younghee Oh

Submitted to the Department of Aeronautical and Astronautical Engineering
on May 5, 1993 in partial fulfillment of the requirements for the Degree of Master of Science in Aeronautics and Astronautics

Abstract

This paper contains a description and detailed computational analysis of a vortex cleaning system designed to remove radioactive material from the plumes of nuclear rockets. The proposed system is designed to remove both particulates and radioactive gaseous material from the plume. A two part computational model is used to examine the system's ability to remove particulates, and the results indicate that under some conditions, the system can remove over 99% of the particles in the flow. However, the effectiveness of the system depends heavily on the size and density of the particles the flow. This paper identifies two critical parameters which govern the effectiveness of the system and provides the information necessary to estimate cleaning efficiencies for particles of known sizes and densities.

A simple steady state analytical solution is also developed to examine the system's ability to remove gaseous radioactive material. This analysis, while inconclusive, suggests that the swirl rates necessary to achieve useful efficiencies are too high to be achieved in any practical manner. Therefore, this system is probably not suitable for use with gaseous radioactive material.

This paper also concludes that the system can cause negligible specific impulse losses, though there may be a substantial mass penalty associated with its use.

Acknowledgements

I wish to thank many people, and despite the fact that there's no page limit on this thesis, it's not really practical to list them all. But you all know who you are, and I thank you. I must give special thanks to my advisor, Professor Daniel Hastings, for putting up with my petulance, randomness, and downright stubbornness in good spirits and for guiding me through my Master's degree and through this thesis. Also, I thank him for putting up with the bad jokes. It's a bad habit, I admit.

I also wish to thank Bryn, for doing all of the above during my doctoral qualifying exam. It's my hope that I'll get to thank them both again three years from now when I get my Ph.D.

This research was funded by the Air Force Office of Scientific Research.

Why?

"The known is finite, the unknown infinite; intellectually we stand on an isle in the midst of an illimitable ocean of inexplicability. Our business in every generation is to reclaim a little more land, to add something to the extent and solidity of our possessions."

-Thomas Huxley, on the Reception of the *Origin of Species* (1887)

This thesis is dedicated to the future and to the hope that this research will be a part of it.

Table of Contents

List of Figures and Tables

Nomenclature

a = Radius of Particulate

c = Local Speed of Sound

$d_{2\eta}$, $d_{2\xi}$ = Second Order Damping Coefficient

$d_{4\eta}$, $d_{4\xi}$ = Fourth Order Damping Coefficient

l_c = Characteristic Length of gas in a Solid Body Rotation Flow

m = Mass of Particulate

m_a = Molecular Mass

m_{ab} = Reduced Mass

n = Number Density

p = Local Pressure

r = Radial Position

s = Entropy

t = Separation Time

s_r = Skimming Ratio

U, u = Magnitude of Fluid Velocity

u_r = Flow Radial Velocity

u_z = Flow Axial Velocity

u_θ = Fluid Tangential Velocity

v_c = Critical Coupling Velocity

z = Axial Position

A = Cross Sectional Area

$\overline{C}$ = Average Thermal Velocity

$\overline{C}_{ab}$ = Average Intermolecular Approach Velocity

C_d = Drag Coefficient

C_p = Specific Heat at Constant Pressure

C_v = Specific Heat at Constant Volume

D, E, F = Flux Vectors

Fc = Centrifugal Force on Particulate

F_D = Drag Force on Particulate

F^D_r, F^D_z, F^D_q, = Components of Drag on Particulate

F^P_r = Radial Pressure Force on Particulate

G = Source Vector

H = Total Enthalpy

I_{sp} = Specific Impulse

J_+, J_- = Riemann Invariants

N = Number of Molecules

P_c = Chamber Pressure

R^* = Ratio of Chamber Radius to Characteristic Length of Gas

Re = Reynolds's Number

R_o = Radius of Separation Chamber

R_s = Distance from Centerline to Skimmer

T = Temperature

T_c = Chamber Temperature

U = Fluid State Vector

$\mathbf{V}$ = Fluid Velocity Vector

Ω = Vorticity Vector

Ω_z = Axial Component of Vorticity

$\Omega^{(2,2)}$ = Average Effective Collision Integral

ψ = Flow Turning Angle

γ = Radio of Specific Heats

μ = Viscosity

ν = Dimensionless Frequency

ν_{ab} = Collision Frequency

η = Cleaning Efficiency

η' = Radius Based Cleaning Efficiency

η'_o = Linear Efficiency when $s_r = 0$

θ = Angular Position

ρ = Density of Fluid

ρ_p = Density of Particle

σ = Ratio of Material removed to Hydrogen Removed

ω = Fluid Rotation Rate

Dots above variables indicate time derivatives.

1.0 Introduction

1.1 General Background

Nuclear thermal rockets have existed as a concept since the 1950's, and offer a combination of high thrust and high specific impulse that make them ideal systems for the transportation of large payloads in a timely manner. In particular, in recent years, the Space Exploration Initiative has brought a renewed interest in these rockets for use in manned interplanetary missions. The basic concept behind nuclear thermal propulsion is simple: a critical mass of Uranium is used to heat hydrogen gas, which is then accelerated out a nozzle at the rear of the vehicle. The overall performance of the system is largely determined by the configuration of the Uranium. If the core consists of solid columns of radioactive material, the system is referred to as a solid core system. If the core is constructed from small spheres of Uranium with diameters less than 0.5 mm, the system is referred to as a particle bed system. Both of these systems are expected to have specific impulses in a range from 500 to 1000 sec. A more radical approach proposes the use of gaseous uranium to form a core of radioactive gas. These gas core systems may have specific impulses over 2000 sec.

Over the past two decades, a great deal of theoretical and experimental work has been done on these systems and, given the need, solid core rockets could be built and flown within the decade. However, the presence of radioactive material in these rockets creates very serious safety and contamination issues if these devices are to be ground tested or used in the near earth environment. In particular, during the NERVA tests of the 1960's, it was observed that even solid core rockets can emit highly radioactive hydrogen plumes. These plumes not only make it difficult and expensive to test these systems, but may contaminate the area on and around the spacecraft with radioactive material. Unfortunately, there is very little experimental data available on the composition of these plumes, but judging from what is available, it seems reasonable to assume that both gaseous and solid radioactive material will be present in the exhaust plume.[1] This paper describes a system which would use artificially induced swirl to remove both types of material from the exhaust plume before it leaves the rocket nozzle. It outlines the proposed system, describes the analytical and computational models used to analyze it, and discusses the parameters which govern its overall effectiveness. This paper includes a detailed description of the system, a list of significant system parameters, a series of charts showing its theoretical effectiveness, and a brief discussion of the penalties associated with its use.

1.2 Description of Proposed Vortex Separation System

At the present time, there is very little theoretical or experimental work available on the composition of nuclear rocket plumes. However, based on the available literature, it seems reasonable to assume that this material will take on the following forms in solid core nuclear rockets.

- Gaseous Fission Products, particularly noble gasses like Xenon and Krypton.[1]
- Solid particulates from condensed metallic fission products.
- Solid particulates resulting from fuel element corrosion (i.e. Carbon or zirconium carbide).[1]

In addition, open cycle gas core reactor plumes may contain another major source of radioactivity in the form of

- Gaseous Uranium fuel which has escaped from the core of the reactor.

In general, it is known that fission reactions produce a variety of products with atomic masses from 75 to 165 AMU's and scattered in bimodal distribution. The distribution function is shown in figure 1.1 below.

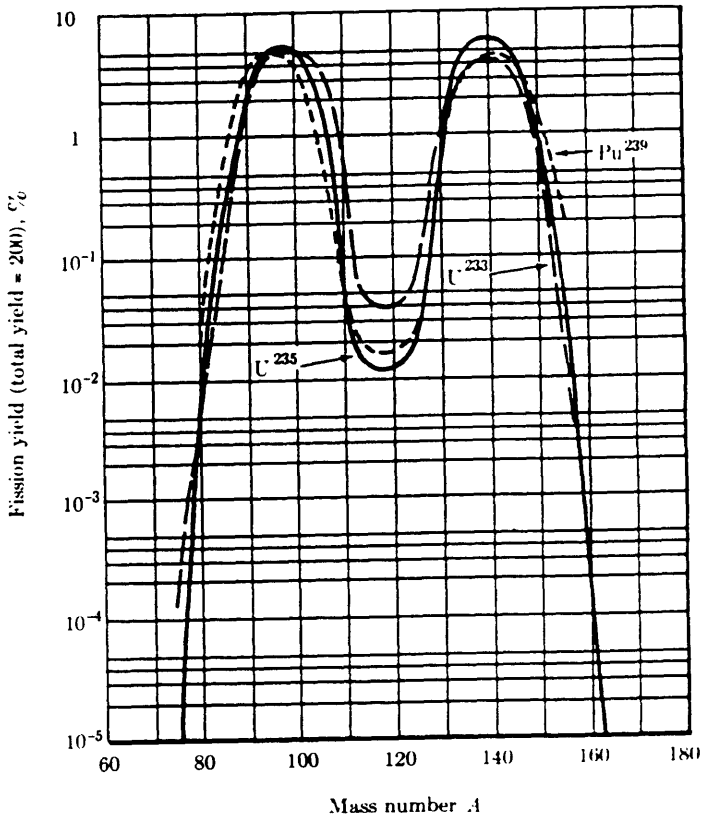


Figure 1.1: Distribution of Fission Products from Uranium Fission Reactions[2]

While it is probable that some fission products will escape the core and contaminate the plume, the amount which will escape is unknown. At the present time, only very limited data is available on the distribution (i.e. type and quantity) of material in nuclear rocket plumes. Some experimental data is available for gaseous material in solid core plumes, but none is available on solid particulates and, as of this writing, gas core reactors are still hypothetical devices. Because of the lack of relevant data, this paper will present models which are applicable across a wide range of materials, but will not make any general conclusions about what portion of the total radioactivity that can be removed from the plume.

The basic principle behind swirl based particle separation is a simple one: since the radioactive material in the plume has a much higher density than hydrogen gas, centrifugal forces can be used to force it to the outside of the flow where it can be "skimmed" off before it passes through the nozzle. Such forces can be created by passing the flow through a fixed vane swirler and creating a vortex in a separation chamber. A diagram of the proposed system in shown in figure 1.2.

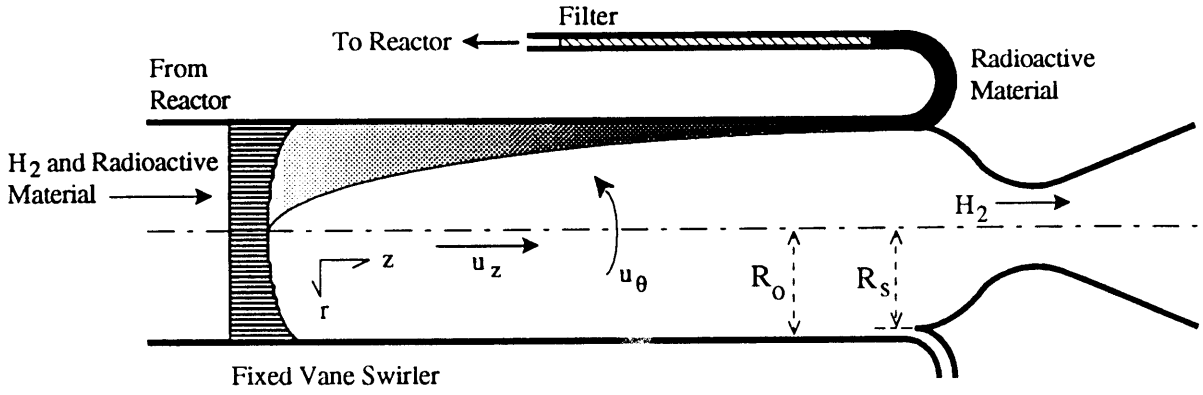


Figure 1.2: Proposed Swirl Based Cleaning System

Centrifugal separation systems have has been used for decades in chemical processing and uranium enrichment systems. This particular system, however, is unique because the flow velocity places severe constraints on the separation time. Under typical conditions, the hydrogen might leave the combustion chamber at velocities near 1100 m/s (M_Z = 0.3). Therefore, if the separation chamber is a meter in length, the separation time must be on the order of a millisecond or less if the system is to work effectively.

Once the flow has passed through the separation chamber, its outer portion can skimmed off and filtered to remove any radioactive material. The non-radioactive portion of the flow can then be returned to the reactor and reused as a propellant. The result is a regenerative cleaning scenario which greatly lowers the engine's I_{sp} losses. It should be noted, however, that the lost mass flow still lowers the thrust of the system. There are

additional performance penalties associated with the mass of the separation chamber, the filtering system, and their associated cooling systems, but this paper does not examine these issues. Rather, it deals with the vortex separation process and the ability of this system to keep radioactive material from escaping out the rocket nozzle. The specifics of the filtering and cooling process are left for future work.*

While vortex separation could theoretically be used with both solid and gas core rockets, for this study, it is assumed that the system is coupled to a NERVA class solid core rocket with a nominal thrust of 300,000 N, a chamber temperature of 2500K, and a chamber pressure of 29 atm. The system is assumed to be operating in vacuum and to have a throat radius of 0.119 m and a chamber radius of 0.148 m. The result is an axial velocity of approximately 1100 m/s ($M_z = 0.3$) in the separation chamber.

1.3 Definition of Parameters

It is convenient at this point to define several variables which characterize the overall performance of the system. The effectiveness of a particle removal system can be expressed in terms of two figures of merit. The overall efficiency of the system, η, is defined to be

$$\eta = \frac{n_p}{N_p} \tag{1.3-1}$$

where n_p is the number of particles removed from the flow and N_p is the total number of particles initially present in the flow. Thus, this parameter represents the fraction of the radioactive material removed from the flow. It is also possible to define a radius based efficiency as follows

$$\eta' = \frac{R_o - R_\eta}{R_o} \tag{1.3-2}$$

where R_o is the radius of the cleaning channel and R_η is the initial radius of the inner-most particle which is skimmed off the flow. All particles which enter the channel with an initial radius greater than or equal to R_η are skimmed off and all particles with an initial radius less than R_η continue out the end of the nozzle. If the particles enter the channel with a uniform distribution, η and η' are related as follows.

---

* Information on one filtering system designed for use on the ground is available in reference (1), but the system discussed is not directly applicable to this work.

$$\eta = 1 - (1 - \eta')^2 \tag{1.3-3}$$

Another useful design parameter is the skimming ratio, s_r, which represents the portion of the total flow radius removed by the particle skimmer. It is defined as

$$s_r = \frac{R_o - R_s}{R_o} \tag{1.3-4}$$

where R_o is the outer radius of the channel and R_s is the radius of the channel after the outer material has been "skimmed" off (as shown in Figure 1.2). It should be noted that the skimming ratio is related but not identical to the fraction of the total flow mass removed by the skimmer.

Finally, because the axial velocity in the separation chamber varies with rotation rate, it is often useful to express data in terms of separation time rather than directly in terms of the chamber length. The separation time, t, is obtained by dividing the chamber length by the mean axial velocity of the flow. It represents the time necessary to obtain a given degree of particle-flow separation.

2.0 Computational Particle Model

2.1 Overview

As was stated above, it is theoretically possible to use vortex separation to remove both gaseous and solid radioactive material from the plume before it enters the nozzle. This paper addresses both these situations, though with different degrees of detail. Solid particle removal is studied using results from a computational simulation of fluid-particle interactions in the separation chamber. The gas separation process is examined using a separate analytical model which is discussed in section four.

In order to model the particle separation process, a two part computational simulation has been developed to model the motion of particles in a swirling flow. The first part consists of an invicid flow solver to model the hydrogen in the channel, and the second of a particle tracking code to track the motion of particles in that flow. The simulation was run on the 115x30 computational domain shown in figure 2.1 which simulates a constant radius channel with a choked exit.

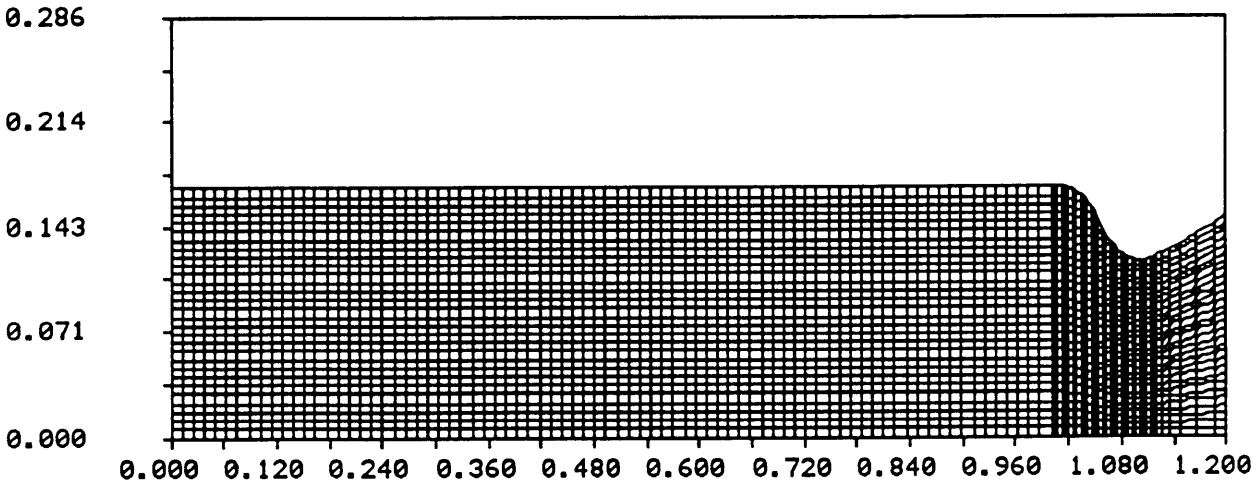


Figure 2.1: Computational Grid

2.2 Governing Fluid Equations

The computation flow model was based on the governing equations for a steady, invicid, axisymmetric flow field. These can be written in conservative form as

$$\frac{\partial U}{\partial t}+\frac{\partial E}{\partial z}+\frac{\partial F}{\partial r}+G=0 \qquad (2.2\text{-}1)$$

Where U, E, F, and G are given by the following expressions:

$$U = \begin{bmatrix} \rho \\ \rho u_z \\ \rho u_r \\ \rho u_\theta \\ \frac{1}{\gamma-1} p + \frac{1}{2}\rho |V|^2 \end{bmatrix} \tag{2.2-2}$$

$$E = \begin{bmatrix} \rho u_z \\ p + \rho u_z^2 \\ \rho u_z u_r \\ \rho u_z u_\theta \\ \frac{\gamma}{\gamma-1} p u_z + \frac{1}{2}\rho u_z |V|^2 \end{bmatrix} \tag{2.2-3}$$

$$F = \begin{bmatrix} \rho u_r \\ \rho u_z u_r \\ p + \rho u_r^2 \\ \rho u_r u_\theta \\ \frac{\gamma}{\gamma-1} p u_r + \frac{1}{2}\rho u_r |V|^2 \end{bmatrix} \tag{2.2-4}$$

$$G = \begin{bmatrix} \frac{\rho u_r}{r} \\ \frac{u_z u_r \rho}{r} \\ -\frac{\rho u_\theta^2}{r} + \frac{\rho u_r^2}{r} \\ \frac{2\rho u_r u_\theta}{r} \\ \frac{\frac{\gamma}{\gamma-1} p u_r + \frac{1}{2}\rho u_r |V|^2}{r} \end{bmatrix} \tag{2.2-5}$$

The first entry in each matrix represents the continuity equation, the next three represent momentum (Euler) equations, and the last the energy equation for a non-conducting ideal gas. In order to solve the flow field, these equations were transformed from physical coordinates (x, y) to computational space (ξ, η) and solved using MacCormick's predictor-corrector method. The coordinate transform is carried out using the following procedure. In steady state, equation (2.2-1) can be written as

$$\frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} = -G \tag{2.2-6}$$

Using the chain rule, the differential terms can be written

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial E}{\partial \eta}\frac{\partial \eta}{\partial x}$$
$$\frac{\partial F}{\partial y} = \frac{\partial F}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial F}{\partial \eta}\frac{\partial \eta}{\partial y} \qquad (2.2\text{-}7)$$

Substituting (2.2-7) into (2.2-6) and rearranging gives

$$\frac{\partial}{\partial \xi}\left(E\frac{\partial \xi}{\partial x} + F\frac{\partial \xi}{\partial y}\right) + \frac{\partial}{\partial \eta}\left(E\frac{\partial \eta}{\partial x} + F\frac{\partial \eta}{\partial y}\right) = -G \qquad (2.2\text{-}8)$$

In order to implement this equation, it is necessary to calculate the ξ and η derivatives numerically. Although these values can not be directly calculated, the values of $\frac{\partial y}{\partial \eta}, \frac{\partial x}{\partial \eta}, \frac{\partial x}{\partial \xi}$ and $\frac{\partial y}{\partial \xi}$ can be calculated directly from the grid using finite differences. Once these values are known, it can be shown that [3]

$$\frac{\partial \xi}{\partial x} = \frac{1}{J}\frac{\partial y}{\partial \eta}$$
$$\frac{\partial \xi}{\partial y} = -\frac{1}{J}\frac{\partial x}{\partial \eta} \qquad (2.2\text{-}9)$$
$$\frac{\partial \eta}{\partial x} = -\frac{1}{J}\frac{\partial y}{\partial \xi}$$
$$\frac{\partial \eta}{\partial y} = \frac{1}{J}\frac{\partial x}{\partial \xi}$$

Where J is the jacobian, i.e.

$$J = \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} - \frac{\partial y}{\partial \xi}\frac{\partial x}{\partial \eta} \qquad (2.2\text{-}10)$$

Eq (2.2-8) was solved using an explicit, time accurate version of MacCormick's finite difference predictor-corrector scheme. When applied to equation (2.2-8), this method can be written as the following two step procedure

$$U_{j,k}^{(1)} = U_{j,k}^{n} - \frac{\Delta t}{\Delta \varsigma}\left(E'^{n}_{j+1,k} - E'^{n}_{j,k}\right) - \frac{\Delta t}{\Delta \eta}\left(F'^{n}_{j,k+1} - F'^{n}_{j,k}\right) - \Delta t G_{j,k} \qquad (2.2\text{-}11)$$
$$U_{j,k}^{n+1} = \frac{1}{2}\left[U_{j,k}^{n} + U_{j,k}^{1} - \frac{\Delta t}{\Delta \varsigma}\left(E'^{(1)}_{j,k} - E'^{(1)}_{j-1,k}\right)\right] - \frac{1}{2}\left[-\frac{\Delta t}{\Delta \eta}\left(F'^{(1)}_{j,k} - F'^{(1)}_{j,k-1}\right) - \Delta t G^{(1)}_{j,k} + D^{n}_{j,k}\right]$$

Where E' and F' are defined in equation (2.2-8), i.e.

$$E' = E\frac{\partial \xi}{\partial x} + F\frac{\partial \xi}{\partial y}$$
$$F' = E\frac{\partial \eta}{\partial x} + F\frac{\partial \eta}{\partial y} \qquad (2.2\text{-}12)$$

D is a smoothing term consisting of the sum of a second order damping term (D_2) and a fourth order damping term (D_4). These terms are given respectively by

$$D^n_{2(j,k)} = d_{2\zeta}\left[U^n_{j-1,k} + 2U^n_{j,k} + U^n_{j+1,k}\right] + d_{2\eta}\left[U^n_{j,k-1} + 2U^n_{j,k} + U^n_{j,k+1}\right]$$
$$D^n_{4(j,k)} = -d_{4\zeta}\left[U^n_{j+2,k} + U^n_{j-2,k} - 4\left(U^n_{j+1,k} + U^n_{j-1,k}\right) + 6U^n_{j,k}\right] \qquad (2.2\text{-}13)$$
$$-d_{4\eta}\left[U^n_{j,k+2} + U^n_{j,k-2} - 4\left(U^n_{j,k+1} + U^n_{j,k-1}\right) + 6U^n_{j,k}\right]$$

It should be noted that these damping terms are implemented in non-conservative form and do not take the grid metrics into account. Therefore, the use of damping leads to small, but noticeable, errors in mass and momentum conservation.

2.3 Fluid Boundary Conditions

The flow boundary conditions are specified separately along each of four flow boundaries. The upper and lower boundary conditions are specified by imposing flow tangency along the upper wall and imposing a no swirl condition at the centerline. The inlet conditions are specified using a modified Riemann invariant scheme and the outlet conditions are specified using a simple extrapolation from the interior of the flow. This section consists of a detailed description of these boundary conditions and the methods used to implement them.

The boundary conditions at the upper wall are based on the assumption that the flow is tangent to the wall and that the pressure gradient along the boundary is zero. Because of the nature of the numerics the actual magnitude of the velocity is relatively unimportant. The boundary conditions are imposed using a two step scheme which initially sets the axial and tangential components at the wall equal to the axial and tangential components of the velocity at the adjacent node, and then adjusts the radial component so that the velocity is vector is tangent to the wall. The turning angle is determined using the following expression

$$\Delta\theta = \pi/2 - \cos^{-1}\left(\frac{\mathbf{V}_z^{j-1}\cdot\mathbf{n}}{\left|\mathbf{V}_z^{j-1}\right|}\right) \tag{2.3-1}$$

Where $\mathbf{V}_z^{j-1}$ is the z component of the velocity at the node adjacent to the wall. The new wall velocity vector is then given by

$$\mathbf{V}_j = \begin{bmatrix} u_z \\ \tan(\Delta\theta)u_z \\ u_\theta \end{bmatrix} \tag{2.3-3}$$

Where $\mathbf{V}_j$ is the velocity vector at the wall node. To maintain zero pressure gradient, the pressure and density at the wall are set equal to the pressure and density at the cell adjacent to the wall, i.e.

$$\begin{aligned} P^j &= P^{j-1} \\ \rho^j &= \rho^{j-1} \\ T^j &= \frac{P^j}{R\rho^j} \end{aligned} \tag{2.3-4}$$

These values are then combined with equation (2.2-2) to construct the new state vector.

The centerline boundary condition is based on the assumption that there is no swirl at the centerline and that core of the flow is in solid body rotation. It is also assumed that the axial velocity gradient near the centerline is zero. The first two conditions are always true in the core of a swirling flow and the third condition is a necessary assumption. In general, the radial pressure gradient in a solid body rotation flow is given by

$$\frac{dP}{dr} = \frac{\rho u_\theta^2}{r} = \rho\omega^2 r \tag{2.3-5}$$

Where ω is the core's rotation rate in rad/sec. For small Δr, the pressure at the centerline is given by

$$P_0 = P_1 - \tfrac{1}{2}\omega^2 r_1^2 \tag{2.3-6}$$

Where P_1 and r_1 are the pressure and radial position of the node adjacent to the centerline. The tangential and radial velocity components are set to zero at the centerline and the axial velocity at the centerline is set equal to the axial velocity at the adjacent node. The density at the centerline is set using the following procedure. The enthalpy at the node adjacent to the centerline is given by

$$H_1 = \frac{\gamma-1}{\gamma}\frac{P_1}{\rho_1} + \tfrac{1}{2}(u_z^2 + u_r^2 + u_\theta^2) \tag{2.3-7}$$

Since the flow has a constant total enthalpy, the density at the centerline can be written as

$$\rho_o = \frac{P_o}{\left(H_1 - \tfrac{1}{2}u_z^2\right)\left(\frac{\gamma-1}{\gamma}\right)} \tag{2.3-8}$$

These values are then combined with equation (2.2-1) to construct the new state vector.

The outlet conditions are relatively simple because the presence of the throat assures that the flow at the outlet is always supersonic. The outlet boundary conditions are obtained using a weighted upwind averaging method which can be written as

$$\begin{aligned} \rho^i &= 2.0\rho^{i-1} - \rho^{i-2} \\ P^i &= 2.0P^{i-1} - P^{i-2} \\ u_z^i &= 2.0u_z^{i-1} - u_z^{i-2} \\ u_\theta^i &= 2.0u_\theta^{i-1} - u_\theta^{i-2} \\ u_r^i &= 2.0u_r^{i-1} - u_r^{i-2} \end{aligned} \tag{2.3-9}$$

Where the superscripts i, i-1, and i-2 indicate the node at the outlet, the node adjacent to the outlet in the upstream direction, and the node adjacent to that one, also in the upstream direction. These quantities are sufficient to calculate the new state vector at the outlet.

The inlet boundary conditions are mostly determined by conditions in the combustion chamber, but the governing physics is complicated by the presence of subsonic flow. The inlet flow is assumed to be a constant enthalpy flow with no radial velocity component and a predetermined swirl velocity profile. In general, conditions at the inlet were set using a modified form of the Riemann Invariants, J_+, J_-, u_θ, and s. Because the flow is subsonic, J_- is determined by downstream conditions and can be written as

$$J_- = u_z - \frac{2}{\gamma-1}a = u_z - \frac{2}{\gamma-1}\sqrt{\gamma RT} \tag{2.3-10}$$

The expression for J_+ is similar, but not directly useful since the upstream conditions are determined by chamber conditions and cannot be calculated directly. Instead, it is useful to look at the flow's total enthalpy, i.e.

$$H = C_p T + \tfrac{1}{2}|V|^2 \tag{2.3-11}$$

Since the flow is assumed to have no initial radial velocity, this expression can be written as

$$H = C_p T + \tfrac{1}{2}(u_z^2 + u_\theta^2) \tag{2.3-12}$$

The total enthalpy is determined by the chamber conditions, so

$$H = C_p T_c \tag{2.3-13}$$

Since the swirl velocity is a defined parameter, equations (2.3-10) and (2.3-12) can be combined to solve for the temperature and the axial velocity at the inlet. Solving equation (2.3-10) for T gives

$$T = \frac{(\gamma-1)^2}{4\gamma R}(u_z^2 - 2u_z J_- + J_-^{\,2}) \tag{2.3-14}$$

Substituting (2.3-14) into (2.3-12) results in a quadratic with the following solution

$$u_z = \frac{2J_- + \sqrt{4J_-^2 - 4\left(1+\frac{2}{\gamma-1}\right)\left(J_-^{\,2} - T_c\frac{4\gamma R}{(\gamma-1)^2} + \frac{2u_\theta^2}{\gamma-1}\right)}}{2+\frac{4}{\gamma-1}} \tag{2.3-15}$$

Empirically, it has been determined that the positive sign is the correct sign to use before the radical in this equation. The entropy of a swirling flow can be calculated using Crocco's relation, which relates the total enthalpy, entropy, velocity and vorticity as [4]

$$\nabla H = T\nabla s + (\mathbf{u}\times\Omega) \tag{2.3-16}$$

For a constant enthalpy flow, this can be expressed as

$$\frac{a^2}{\gamma R}\nabla s = -\mathbf{u}\times\Omega \tag{2.3-17}$$

Using equation (2.3-12), this can be written as follows for a constant enthalpy flow.

$$\nabla s = -C_p\frac{1}{H - \tfrac{1}{2}(u_z^2 + u_\theta^2)}(\mathbf{u}\times\Omega) \tag{2.3-18}$$

Since the flow field is cylindrical, the vorticity has no radial or tangential component and is given by

$$\omega_z = \frac{1}{r}\frac{d}{dr}(ru_\theta) \tag{2.3-19}$$

As a result, the cross product $(\mathbf{u}\times\Omega)$ is entirely radial, and the entropy gradient can be simplified to

$$\frac{ds}{dr} = \frac{C_p\Omega_z u_\theta}{H_\infty - \tfrac{1}{2}(u_z^2 + u_\theta^2)} \tag{2.3-20}$$

This integral can be evaluated numerically to give s as a function of radius.

Once s has been calculated, the thermodynamic definition of entropy can be combined with the equation of state to determine the flow's pressure and density. In general, the entropy change in an ideal gas is given by

$$s_2 - s_1 = C_v\left\{\ln\left[\left(\frac{P_2}{P_1}\right)\left(\frac{\rho_1}{\rho_2}\right)^\gamma\right]\right\} \tag{2.3-21}$$

Where s_1, P_1, and ρ_1 represent reference values. If the entropy at the centerline is designated as the reference value and assumed to be zero, solving this equation results in the following expression for the density as a function of entropy.

$$\rho = \left[\left(\frac{\rho_o^\gamma}{P_o}\right)\left(\frac{\gamma}{\gamma-1}\right)C_p T e^{-\frac{s}{C_v}}\right]^{1/\gamma-1} \tag{2.3-22}$$

Once the density and temperature are known, the pressure can be calculated using the ideal gas law. The result is a set of four equations, (2.3-14), (2.3-15), (2.3-20), and (2.3-22), which fully specify the inlet boundary conditions.

The complex nature of the inlet conditions merit some discussion. While these equations are physically accurate, they also define a fully reflecting boundary condition, so even small disturbances are reflected back into the flow. In general, this prevents the numerical scheme from converging in reasonable periods of time because pressure disturbances in the subsonic section are reflected back and forth between the (choked) throat and the inlet. Presumably, in a real system, these waves dissipate over time, leaving a steady flow in the separation chamber. However, in this invicid simulation, these waves propagate back and forth forever, preventing convergence of the solution. In order to speed

convergence, the first 2000 time steps are calculated using relatively large amounts of second order damping to dissipate out disturbances. Once the largest disturbances have disappeared, the second order damping is removed and the solution is allowed to converge with only fourth order damping present. Even with this procedure, some waves persist in the solution, and over time, a point is reached where further iterations fail to produce further convergence. However, because these waves appear to be physical phenomenon and because they are small, it was decided that their presence did not seriously affect the overall accuracy of the solution. Therefore, the flows used to compute the final particle separation results contained small pressure disturbances. These disturbances were small enough that they should not have affected the particle separation results in any substantial manner.

The final simulation was written in C and run on a Silicon Graphics Iris Indigo with a IP12 processor running at 33 MHz. Each simulation generally took a four or five hours to complete, and resulted in flow results that fit general theory. Figures 2.2 and 2.3 show typical results for a solid body rotation flow with a rotation rate of 10000 rad/sec.

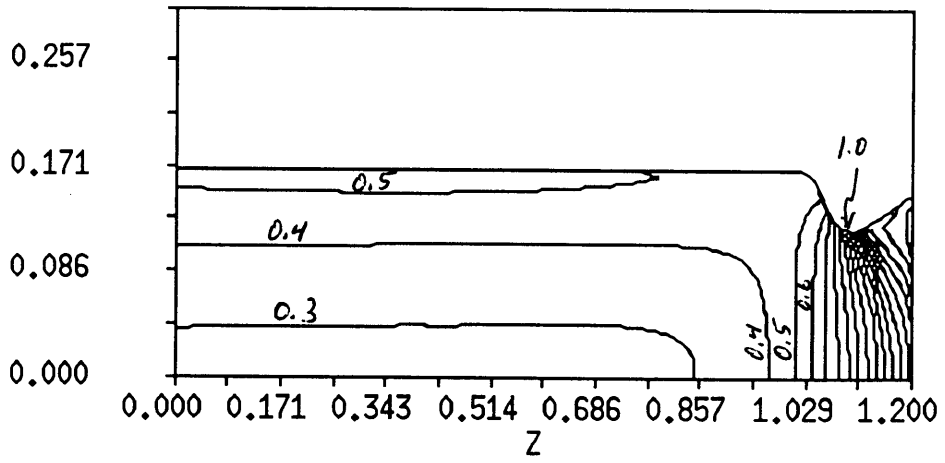
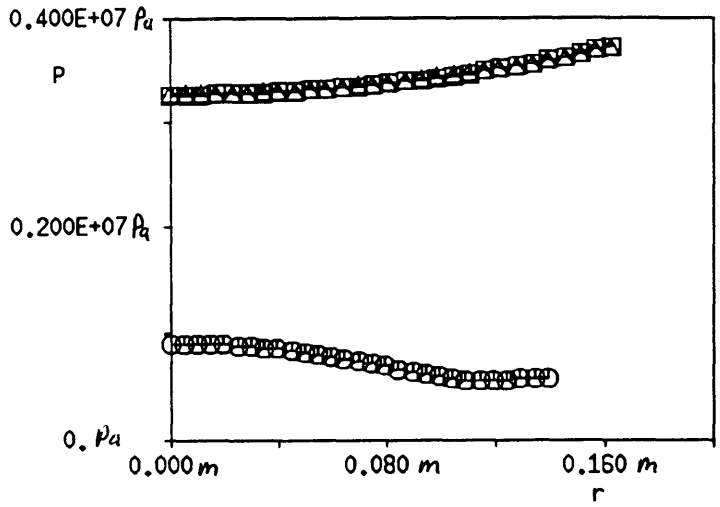


Figure 2.2: Mach Number Contour Plot of Solid Body Rotation Plot



Squares represent pressure at the inlet.
Circles represent pressure at the outlet.

Figure 2.3: Radial Plot of Pressure vs. Radius for Solid Body Rotation Flow

As one would expect, figure 2.2 shows that the flow has a high swirl velocity near the wall and properly chokes at the throat. Figure 2.3 shows the effect of swirl on the radial pressure gradient. As one would expect, in the inlet area, there is an exponential rise in the pressure as the radius increases. This occurs because the pressure forces must offset the centrifugal forces acting on the flow. At the outlet, the pressure drops dramatically and the radial pressure gradient is radically deformed. This occurs because of the choking condition causes large variations in the flow's axial velocity profile. These results are all consistent with simple nozzle theory. In all the results used for this study, mass flow is conserved to within 3%. The remaining variations are due to the use of non conservative damping and the errors introduced by the reflective inlet boundary condition.

2.4 Governing Particle Equations

The second part of the simulation consists of a particle tracker that is used to post process the solutions developed by the MacCormick solver. Due to the lack of relevant experimental data, it is necessary to make several assumptions about the composition of the radioactive material in the flow. Throughout this study, this material is assumed to consist of solid, spherical particles distributed uniformly across the flow. These particles are assumed to be composed of solid graphite or uranium with diameters ranging from 2 to 200 μm; a range of distributions which corresponds roughly to the distribution of dust particles in a solid rocket booster.[5] This distribution has no particular significance, but seemed a reasonable assumption. It is also assumed that any particle which reaches the outer wall of the channel remains in the boundary layer and does not rebound back into the center of the

flow. This implies that any particle caught by a skimmer placed at a given position in the channel will also be caught by a skimmer placed downstream of that position.

The particle tracker developed for this study simulates the release of a uniform distribution of spherical particles at the inlet of the chamber and tracks the particles individually to determine which are caught by the skimmer and which simply pass through the nozzle. This tracker did not actually model the skimmer itself, but instead took a series of cross sections at various points in the separation chamber and assumed that any particles with a radial position, r, greater than the R_S specified by the skimming ratio was caught by the skimmer. The governing equations for spherical particles in an axisymmetric swirling flow are

$$\ddot{r} - r\dot{\theta}^2 = \frac{F_r^D + F_r^P}{m} \tag{2.4-1}$$

$$r\ddot{\theta} + 2\dot{r}\dot{\theta} = \frac{F_\theta^D}{m} \tag{2.4-2}$$

$$\ddot{z} = \frac{F_z^D}{m} \tag{2.4-3}$$

Where F^D is the drag force on the particle and F^P is the force on the particle due to the presence of pressure gradients. In general, F^P is obtained by integrating the pressure across the surface of each particle. However, using the divergence theorem, it can be shown that for small particles this integral is approximately given by:

$$F_p = V\nabla P = \tfrac{4}{3}\pi a^3 \nabla P \tag{2.4-4}$$

Where V is the volume of the particle and ∇P is the pressure gradient across it. In a constant area axisymmetric channel, the pressure gradient is entirely radial and is given by

$$\frac{dP}{dr} = \frac{\rho u_\theta^2}{r} \tag{2.4-5}$$

Combining (2.4-4) and (2.4-5) gives

$$F_p \equiv \tfrac{4}{3}\pi a^3 \frac{u_\theta^2}{r}\rho \tag{2.4-6}$$

This expression is valid for small particles like the ones modeled in this simulation. An idea of the relative importance of this term can been obtained by comparing it to the centrifugal forces on the particle. In general, these centrifugal forces are given by

$$F_c = \frac{m u_\theta'^2}{r} = \frac{\frac{4}{3}\pi a^3 \rho_p u_\theta'^2}{r} \tag{2.4-7}$$

Where F_c is the magnitude of the centrifugal force and u_θ' is the particle's tangential velocity. Dividing (2.4-6) by (2.4-7) shows that the ratio of these forces is given by

$$\frac{F_p}{F_c} = \frac{\rho}{\rho_p} \frac{u_\theta'^2}{u_\theta^2} \tag{2.4-8}$$

In general, because the particles consist of solid material, ρ_p is five or six orders of magnitude larger than ρ. As a result, centrifugal forces will dominate the particle's motion in any situation where $u_\theta' / u_\theta > 0.01$. Therefore, under most conditions, it is possible to neglect the pressure forces acting on the particle. Nevertheless, for sake of completeness, this pressure term was included in the computational pressure model.

The drag force on the particle is calculated using the usual expression

$$F^D = \frac{1}{2}\rho u^2 A C_d \tag{2.4-9}$$

Where the cross sectional area, A, is given by

$$A = \pi a^2 \tag{2.4-10}$$

and C_d is the drag coefficient for a fully immersed sphere. For $0 < Re < 2.5 \times 10^5$, the drag coefficient is given by the following formula.[4]

$$C_d = \frac{24}{Re} + \frac{6}{1+\sqrt{Re}} + 0.4 \tag{2.4-11}$$

Where Re is the diameter based Reynolds Number. The values given in this expression is always within 10% of the experimental, as is shown in figure 2.4 below.

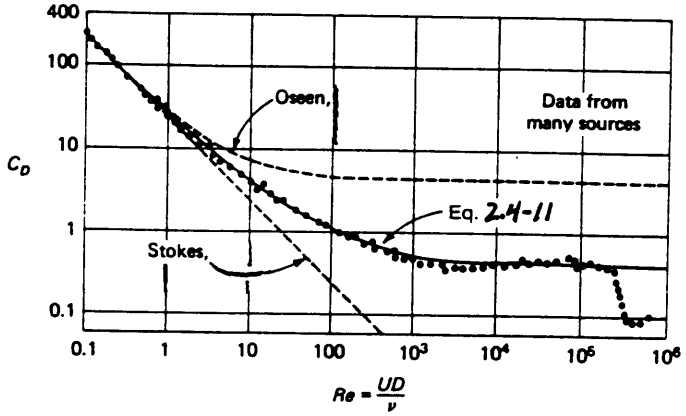


Figure 2.4: C_d vs. Re for a Fully Immersed Sphere[6]

The viscosity of H_2 is determined using a low order approximation for the viscosity of a pure gas. The following equation gives the viscosity in units of kg/m/s.[7]

$$\mu = 2.6693 \times 10^{-26} \frac{(MT_g)^{\frac{1}{2}}}{\Omega^{(2,2)}} \qquad (2.4\text{-}12)$$

Where $\Omega^{(2,2)}$ is the average effective collision integral. This quantity is temperature dependent and has been recorded for H_2 by Vanderslice et al.[8] A table of these values is included in Appendix A. Typical values for μ at the temperatures encountered in the simulation range from 2.5 x 10^{-5} to 3.8 x 10^{-5} kg/m/s. Overall, for particles with radii under 100μm, this model should be valid at relative velocities from 0 m/s to over 10000 m/s and at temperatures up to 15,000 K.

Each run consists of the release of 200 particles at evenly spaced distances from the centerline (i.e. the first particle if released at r=0, the last one at r=R_O.) In all runs, it is assumed that the particles enter the chamber with an axial velocity equal to the flow's axial velocity, but that the particle's tangential velocity is zero.

3.0 Particle Removal Analysis and Results

3.1 Overview

The computational simulation described in section two was used to examine the effectiveness of two types of fixed vane swirlers over a variety of operating conditions. The two types of swirlers examined were constant angle and solid body rotation swirlers. In constant angle swirlers, the swirl velocity is given by

$$u_\theta = u_z \tan \psi \tag{3.1-1}$$

where ψ represents the flow turning angle. This relation holds everywhere except close to the centerline, where the swirl velocity is restricted by a no-swirl boundary condition. In solid body rotation swirlers, the governing equation is

$$u_\theta = \Omega r \tag{3.1-2}$$

where Ω represents the flow's rotation rate in radians/sec. Figure 3.1 shows plots of u_θ vs. r for both types of swirlers.

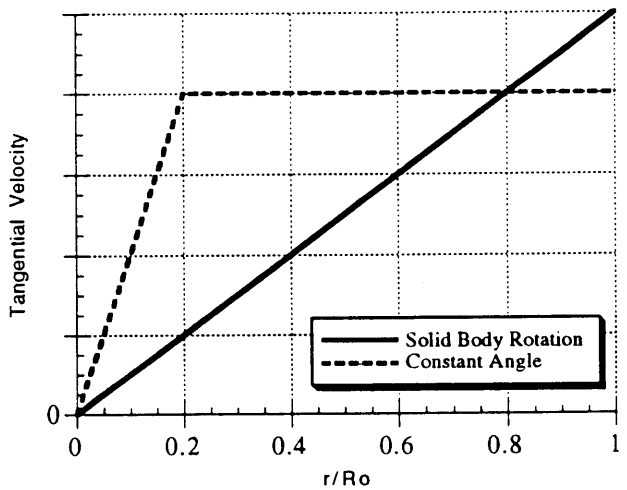


Figure 3.1: Tangential Velocity Profiles for Swirlers Examined in Study

The results discussed in section three are based on solid body rotation swirl results. Constant angle swirlers are discussed below in the section on alternative flow profiles.

3.2 Solid Body Rotation Results

A total of five parameters were varied during the solid body rotation swirl runs: the chamber length, the particle density, the particle radius, the skimming ratio, and the flow rotation rate. The particle density was simply varied between two values. These values corresponded to the density of graphite, 1800 kg/m^3, and Uranium, 18700 kg/m^3. The other four parameters were varied as follows:

- Chamber Length: 0 to 1 meter.
- Particle Radius: 1 to 100 μm.
- Rotation Rate: 0 to 10000 rad/sec.
- Skimming Ratio: 0 to 0.2

3.2.1 Relationship between Efficiency and Skimming Ratio

The solid body rotation results show several trends which greatly simplify analysis of the overall system. Figure 3.2 shows a graph of the system's linear efficiency vs. skimming ratio for a series of different separation times. It shows simulated data for 40 μm graphite particles in a flow with a rotation rate of 10000 rad/sec, and each point on the graph represents a simulated data point.

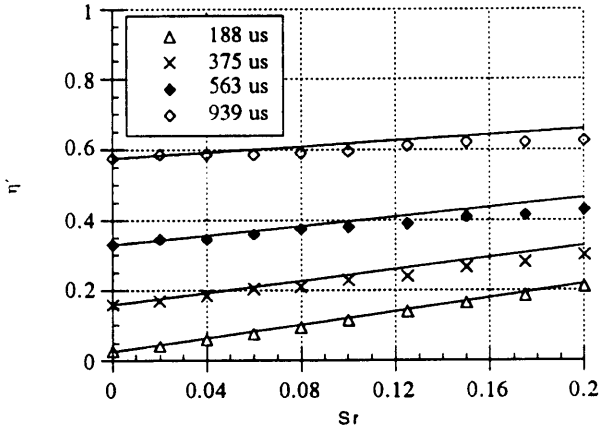


Figure 3.2: η' vs. Sr at 10000 rad/second

Figure 3.2 clearly shows the presence of a linear relationship between the skimming ratio and the linear efficiency. In fact, the linear efficiency results can be approximated using the following equation

$$\eta' = \eta'_o + (1 - \eta'_o)s_r \qquad (3.2.1\text{-}1)$$

where η'_0 is the linear efficiency for a skimming ratio of zero. The solid lines on the graph are based on this approximation. Figure 3.3 show a similar graph for 10 μm particles with a flow rotation rate of 8000 rad/sec. Again, the points represent simulated data and the solid lines indicate curve fits calculated from equation (3.2.1-1).

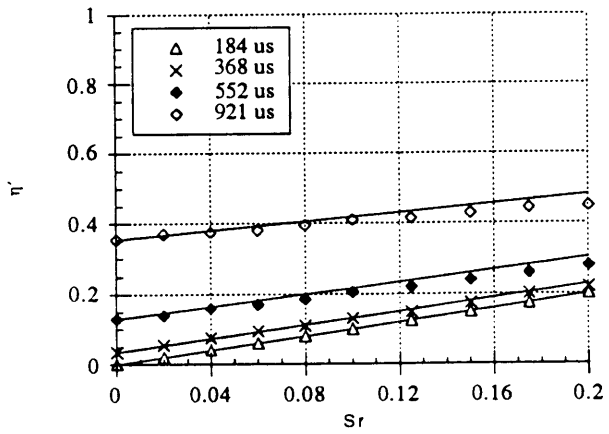


Figure 3.3: η' vs. Skimming Ratio at 8000 rad/sec.

Although the values of η' have changed, the basic relationship between the linear efficiency and skimming ratio is preserved, and equation. (3.2.1-1) is still a close approximation to the simulated data. In fact, an examination of the all the simulated data shows that equation (3.2.1-1) is valid to within 10% across the entire range of simulated operating conditions and is generally within 5% of the simulated results. Therefore, equation (3.2.1-1) and a table of reference linear efficiencies provide all the information necessary to calculate the efficiency of this system at any skimming ratio. It should be noted that as the overall efficiency rises, changing the skimming ratio has less and less effect on the system. Therefore, in the operating ranges of interest to the designer, the choice of skimming ratio has relatively little effect on the overall efficiency of the system.

3.2.2 Relationship between Efficiency and ν

The relationship of greatest interest to the designer is that between the efficiency and the flow rotation rate. Figure 3.4 shown as graph of efficiency vs. rotation rate for 40 μm particles at a skimming ratio of zero and with chamber lengths from 0.2 m to 1 m.

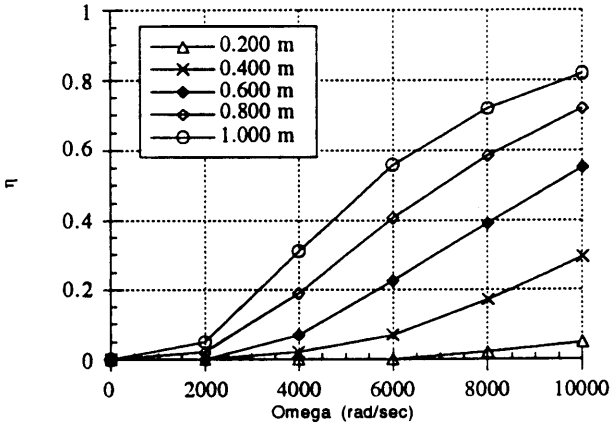


Figure 3.4: η vs. ω and Chamber Length

As one would expect, figure 3.4 shows that as the rotation rate increases, a shorter chamber is required to obtain equivalent levels of cleaning efficiency. The exact relationship between chamber length and efficiency is obscured by the fact that the axial velocity varies slightly with rotation rate. It is also useful to plot the efficiency vs. a dimensionless frequency ν, where

$$\nu = \omega t \tag{3.2.2-1}$$

Figure 3.5 shows the same simulated data presented in figure 3.4, but plotted against this parameter.

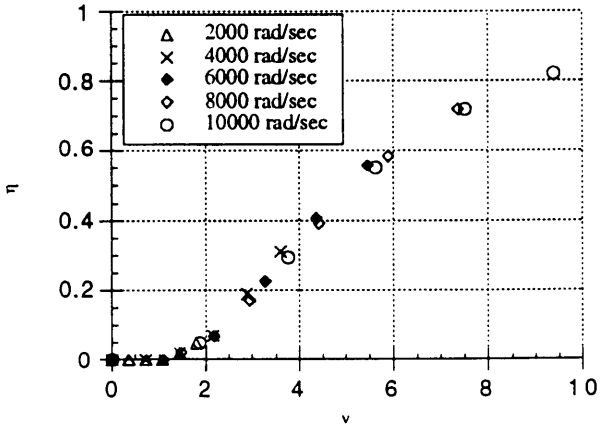


Figure 3.5: η vs. Dimensionless Frequency

Figure 3.5 is very interesting because it shows that the data from an entire series of runs collapses down to a single curve when plotted against ν. This not only allows one to summarize the data in a concise plot, but also indicates that ν may be one of the critical dimensionless parameters in this system. Using equation (3.2.1-1) and figure 3.5, a designer can determine the effectiveness of this system for any skimming ratio, rotation rate, or chamber length. Therefore, this plot contains all of the information necessary to carry out design studies and construct an initial design of a vortex cleaning system. Using a family of these plots, it is possible to summarize the performance of this system across a range of performance parameters. Figure 3.6 (see next page) shows such a family of plots spanning the entire range of simulated data for solid body rotation swirlers.

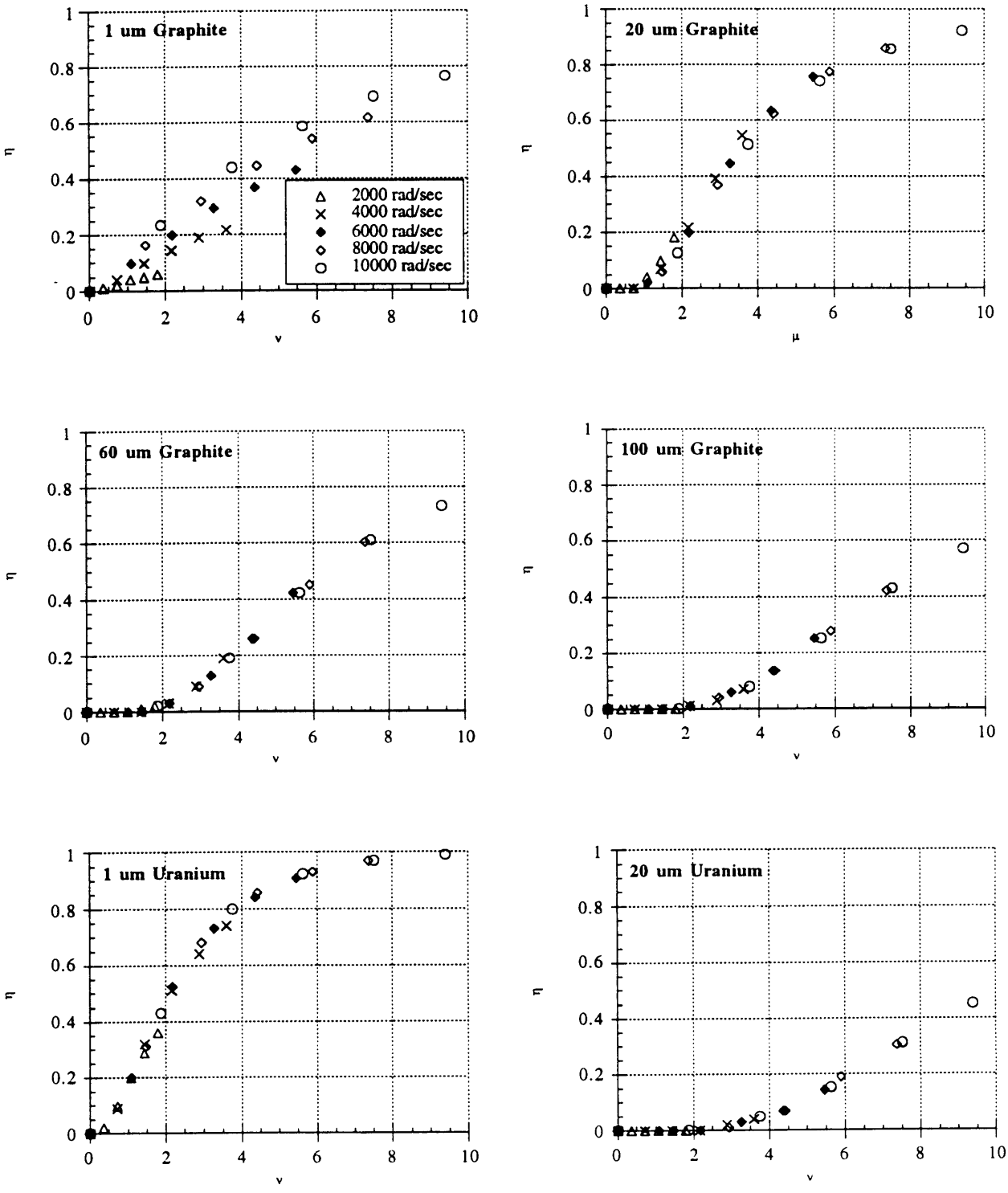


Figure 3.6: Simulated Solid Body Rotation Swirl Data

Figure 3.6 illustrates the fact that the efficiency curves collapse when plotted vs. v for particles spanning a wide range of sizes and densities. In fact, the only simulated case in which this collapse does not occur is with the 1 μm graphite particles, which are "coupled" to the flow; a condition discussed below in section 3.3.2. Figure 3.6 also indicates that the effectiveness of the system depends strongly on the composition of the particles in the flow. For instance, while a 1 m separation chamber operating at 10000 rad/sec would remove almost 99% of the 1 μm Uranium particles, it would remove at most 50% of the 20 μm Uranium particles and becomes even less efficient with larger Uranium particles.

In summary, using results from the computational simulation outlined in section 2, we have succeeded in identifying two relationships which are of interest to the designer. The first is the relationship between linear efficiency and skimming ratio defined by equation (3.2.1-1). This equation not only identifies the linear nature of this relationship, but also shows that in the high efficiency ranges which are of interest to a designer, the choice of skimming ratio has relatively little effect on the system efficiency. The second is the relationship between linear efficiency and skimming ratio. The nature of this relationship is less clearly defined, but figure 3.6 clearly shows that a relationship exists and that the simulated data from a variety of conditions will collapse down to a single curve when plotted against this parameter. This parameter is of critical interest to the designer because it determines the relationship between the flow rotation rate, the length of the separation chamber, and the efficiency of the system.

3.3 Analytical Particle Model and Fundamental Physics

3.3.1 Governing Equations

Since the actual distribution of particles in a nuclear rocket will probably span a wider range of sizes and densities than those examined in section 3.2, it is beneficial to develop a more general understanding of the relationship between the system efficiency and the size and composition of the particles in the flow. One way of doing this is to examine a simple analytical model of the vortex separation process. In general, the presence of drag makes it impossible to the solve particle equations of motions using analytical methods. However, if it is assumed that when the particle enters the channel, it is moving at the same axial and tangential velocity as the fluid, the centrifugal forces on the particle are much larger than the forces due to particle-fluid interaction. Particles traveling in this regime are said to be "uncoupled" from the flow. This is in contrast to a particle's whose motion is dominated by drag and pressure forces which is said to be "coupled" to the flow. Neglecting fluid forces reduces equations (2.4-1)-(2.4-3) to the following form

$$\ddot{r} - r\dot{\theta}^2 = 0 \tag{3.3.1-1}$$

$$r\ddot{\theta} + 2\dot{r}\dot{\theta} = 0 \tag{3.3.1-2}$$

$$\ddot{z} = 0 \tag{3.3.1-3}$$

Solving equations (3.3.1-1) and (3.3.1-2) analytically results in the following solution

$$t = \frac{r}{\omega}\sqrt{\left(1/r_o\right)^2 - \left(1/r\right)^2} \tag{3.3-1-4}$$

where r_O is the particle's initial position and ω is its initial angular velocity. Setting $r = R_S$, makes this an equation for the time it takes for a particle to move from an initial position r_O to a position where it is caught by the skimmer. t represents the particle separation time and approaches infinity as r_O approaches zero, indicating that particles traveling near the centerline will never be caught by the skimmer.

3.3.2 Discussion

Equation (3.3.1-4) can also written in terms of the design parameters η' and s_r as

$$\eta' = \frac{1 - s_r}{\sqrt{\nu^2 + 1}} \tag{3.3.2-1}$$

Equation (3.3.2-1) shows that in an uncoupled flow, there are two fundamental parameters which determine the system's efficiency: the skimming ratio and the dimensionless frequency ν. For a given value of ν, s_r and η' are linearly related with the efficiency ratio approaching unity as s_r approaches unity. This matches the behavior seen in the simulated data, in particular in equation (3.2.1-1) above. The relationship between ν and η' is less clear. Figure 3.7 shows a graph of this relationship for various values of s_r.

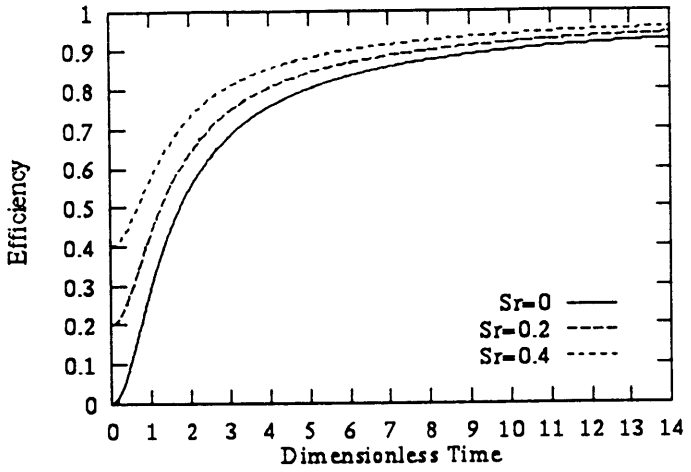


Figure 3.7: Analytical Graph of Efficiency vs. ν and s_r

Although the shape of these curves is not the same of those seen in Figure 3.6, it is still clear that higher rotation rates and higher separation times result in higher efficiencies, and that the parameter governing the effectiveness of the system is the dimensionless frequency, ν. This implies that in some sense, particles in the full simulation act as though they are uncoupled from the flow. However, since particles in the full simulation enter the chamber with a tangential velocity of zero, it is also clear that drag forces must play an important role in determining their motion. The behavior of the actual particles, then, must depend on whether the particles are coupled or uncoupled from the flow.

Because the pressure forces on the particle tend to be small, the critical parameter which determines whether a particle is "coupled" or "uncoupled" from the flow is the ratio of the drag forces to the inertial forces acting on a particle. The magnitude of the centrifugal forces on a particle are given by the following expression

$$F_c = \frac{m u_\theta^2}{r} = \frac{\frac{4}{3}\pi a^3 \rho_p u_\theta^2}{r} \qquad (3.3.2\text{-}2)$$

Using this and expression (2.4-9), it can be shown that the ratio of the centrifugal to the drag force on a particle in a solid body flow is approximately is given by

$$\frac{F_c}{F_d} = \frac{2}{3}\frac{1}{C_d}\frac{a}{r}\frac{\rho_p}{\rho_f}\frac{u_\theta^2}{(r\omega - u_\theta)^2} \qquad (3.3.2\text{-}3)$$

Where ω is the fluid rotation rate. When u_θ approaches 0, F_c/F_d approaches 0, drag forces dominate over any inertial forces, and the particle is strongly coupled to the fluid. However,

as u_θ approaches $r\omega$, F_c/F_d approaches infinity, and the particle is uncoupled from the flow. In between these extremes, there is a critical velocity, v_c, where the ratio $F_c/F_d = 1$. When u_θ is much less than v_c, the particle's motion is dominated by drag forces and it is coupled to the flow. Similarly, when u_θ is much greater than v_c, a particle is uncoupled from the flow and its motion is independent of the fluid. Every particle enters the flow initially coupled to the fluid, will eventually uncouple as it experiences tangential acceleration.

Figure 3.8 shows the approximate value of the force ratio F_c/F_d vs. u_θ for graphite and uranium particles under conditions similar to those used in the computer simulations. In this calculation, it is assumed that the fluid density is 0.3 kg/m^3, the fluid rotation rate is 10000 rad/sec, r = 0.1 m, and m = 2.4 x 10^{-5} kg/m/s. Each particle's critical velocity is indicated by the point at which the curve crosses the solid line.

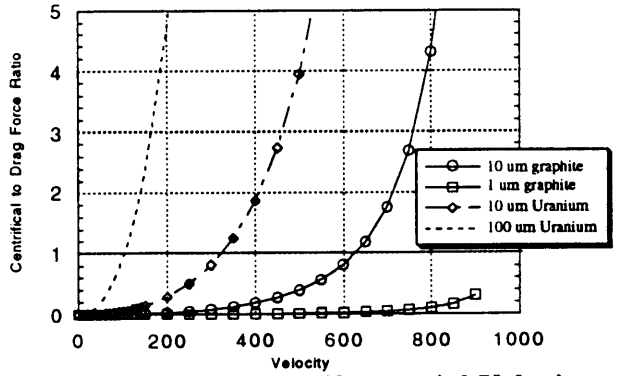


Figure 3.8: Force Ratio vs. Tangential Velocity

As one would expect, figure 3.8 shows that F_c/F_d is zero when $u_\theta = 0$ and approaches infinity as u_θ approaches $r\omega$. At intermediate velocities, the relative importance of the two forces depends on the size and density of the particle being analyzed. Comparing this graph with figure 3.6 shows that particles with critical velocities in an intermediate range (between 200 and 800 m/s) have relatively high recovery rates, while those at the high and low end of the range have relatively low ones. The reasons for this correlation are evident in Figure 3.9, which shows the path followed by three graphite particles which begin at the same position, but have radii of 1, 10, and 100 μm respectively.

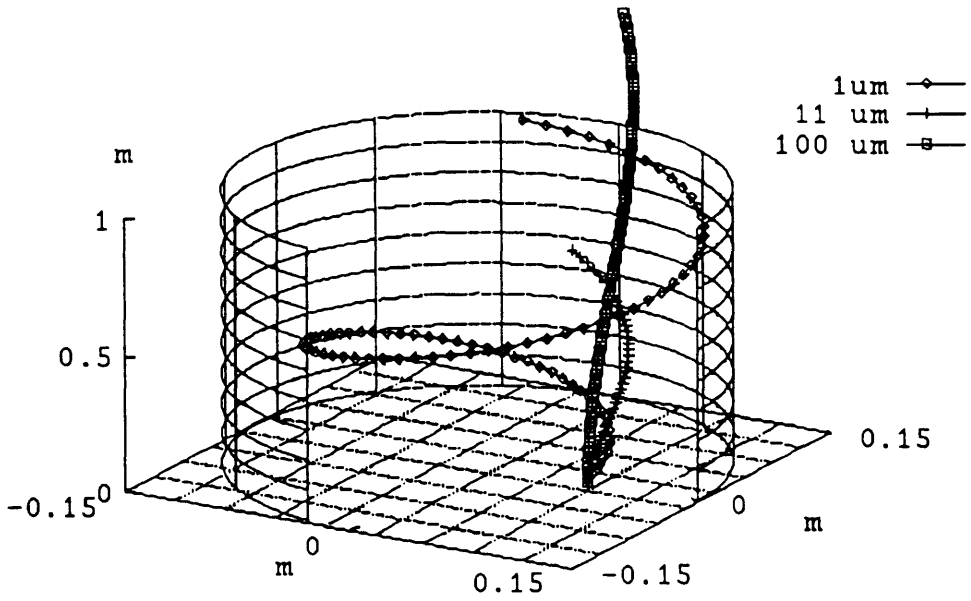


Figure 3.9: Particle Tracking Data

Figure 3.9 shows that particles with different critical velocities follow radically different paths in the separation chamber. The 1 μm graphite particle, which has a high critical velocity, remains coupled to the flow down most of the length of the channel and follows a long, spiral path towards the outside wall. The 100 μm particle, on the other hand, has a low critical velocity and uncouples from the flow very quickly. As a result, it travels directly down the channel, ignoring the influence of the fluid all together. Neither particle moves to the outside of the flow in a particularly efficient manner. In the first case, although the particle is quickly accelerated to a high tangential velocity, drag forces dominate its motion and prevent it from gaining too high a radial velocity. In the second case, the particle uncouples too quickly and practically never accelerates at all. As a result, the particles which are separated most efficiently are those with intermediate critical velocities, such that the particles remain coupled to the fluid long enough to reach a high tangential velocity, and then uncouple as they travel towards the outside of the channel.

Equation (3.3.2-3), then, defines another parameter which is of critical importance to the designer. The critical velocity determines the effectiveness of the vortex separation system and determines which types of particles can and can not be removed by a given separation geometry. Given specific data on the distribution of particles in a nuclear rocket plume, equation (3.3.2-3) can be used to determine what portion of the radioactive material can be removed from the flow. At the present time, however, such data is not available, and all that can be said with certainty is that vortex separation systems will be most effective on particles with critical velocities in an intermediate range.

3.4 Constant Angle Swirl Results

Although solid body rotation swirlers are convenient to characterize and study, they distribute swirl energy in a relatively inefficient manner. In particular, because they create higher swirl velocities at the outside of the flow, they tend to give the greatest acceleration to those particles which are already near the outside of the flow. This is inefficient because it is those particles near the center of the flow which need the greatest acceleration, not those at the outside. Constant angle swirlers, on the other hand, distribute energy evenly across the flow, and are therefore better suited for particle removal applications. To simulate constant angle swirlers, a series of runs were conducted using the constant angle swirl profiles shown in figure 3.1. Note that there is a core of solid body rotation flow at the center of the vortex. This core is necessary to meet the centerline no swirl condition. The simulated results were produced for swirlers with turning angles from 0.0 to 45.0 degrees.

Figure 3.10 is a plot of constant angle rotation data. This data is presented as a plot of linear efficiency vs. skimming ratio and separation time. This simulated data is for 45 μm graphite particles at a turning angle of 45 degrees. The solid lines indicate curve fits made using equation (3.2.1-1).

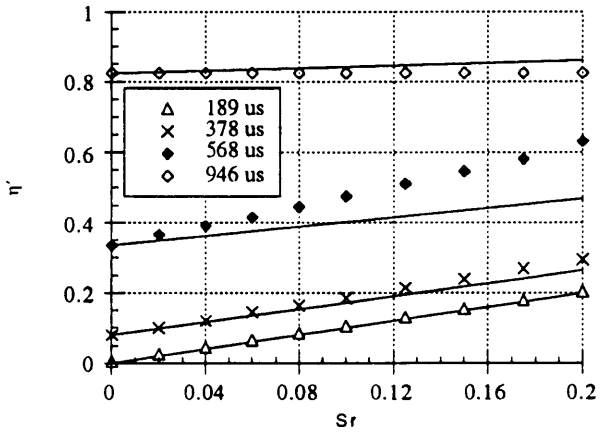


Figure 3.10: η' vs. Sr at 45 degrees turning angle

A comparison of this plot to the equivalent data in Figure 3.2 quickly reveals two interesting points. The first is that the efficiencies are quite a bit higher than those in Figure 3.2, even though the maximum flow turning angle represented in Figure 3.2 is over 50 degrees (this is the turning angle at the outer wall when the rotation rate is 10000 rad/sec). The second is that equation (3.2.1-1) no longer fits the simulated data with any great degree of accuracy. As a result, a table of reference skimming ratios is no longer sufficient to calculate the

system efficiency at any skimming ratio. To fully characterize the system, it is now necessary to collect data at each individual skimming ratio. Fortunately, however, some simplification of the data is possible. Figure 3.11 shows six graphs of simulated constant angle data vs. a dimensionless frequency. In this case, however, the flow rotation rate is not that of the entire flow, but only of the core solid body rotation region. Notice that although the system is no longer a solid body rotation system, the basic relationship between η and ν is still present; data from a variety of runs still collapses down to a single curve. It should be noted that although the values of ν differ, this data was taken over a range of conditions similar to that covered in figure 3.5 and 3.6.

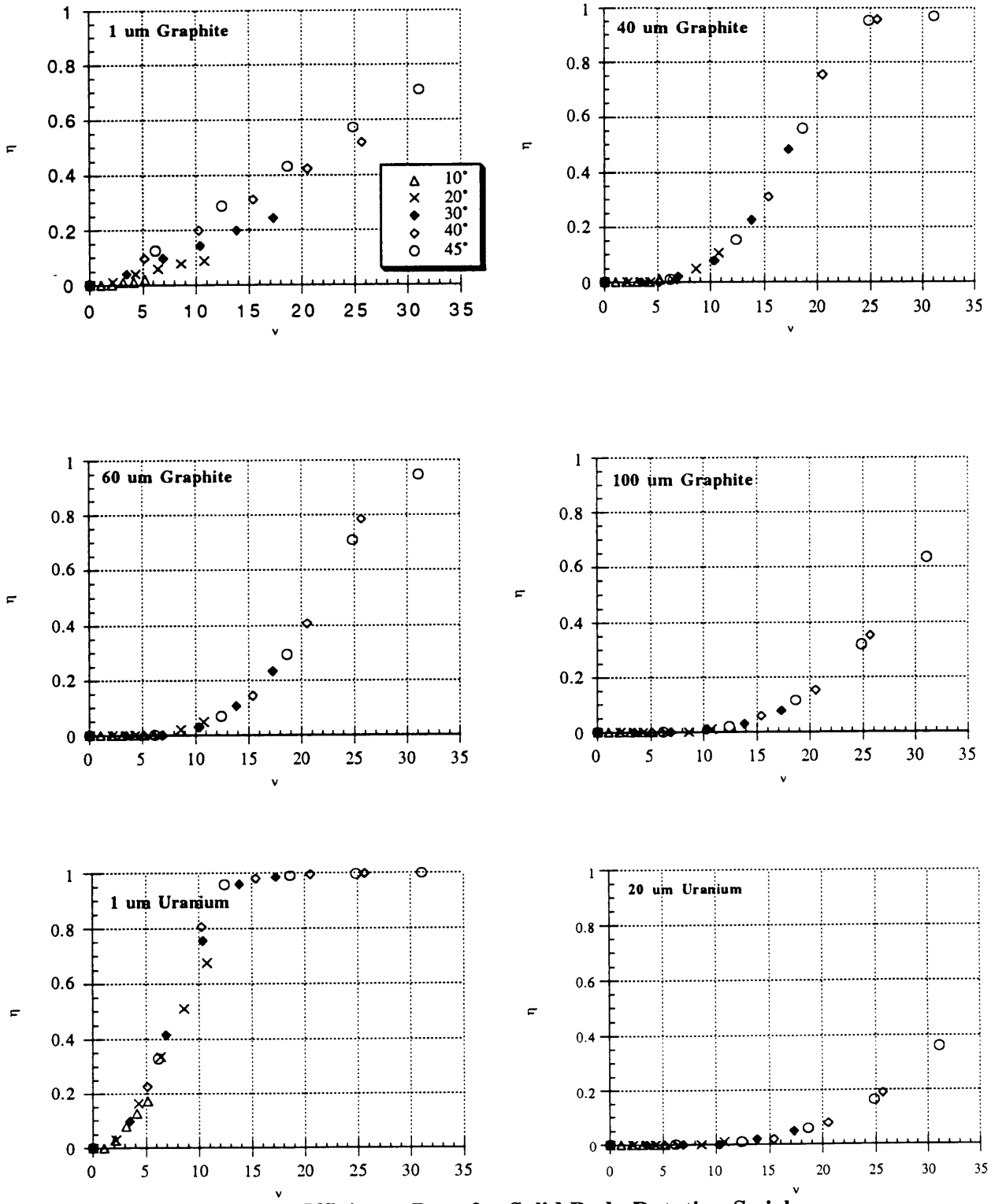


Figure 3.11: Efficiency Data for Solid Body Rotation Swirlers

Figure 3.11 shows that under most conditions, constant angles swirlers are considerably more effective than the equivalent solid body rotation swirlers shown in figures 3.5 and 3.6. Under some conditions, constant angle swirlers can easily remove 99% of the material in the flow, even at moderate turning angles. In addition, although the system is still most efficiency for particles with intermediate critical velocities, constant angle swirlers are able to remove a wider distribution of material from the flow than their solid body rotation equivalents. Therefore, any practical cleaning system is almost certain to be designed around a constant angle swirl profile or something similar to it. Given further work, it may be possible to develop more effective flow profiles which will distribute swirl energy more efficiency across the flow and will enhance the system's efficiency over an even wider distribution of particle sizes and densities.

4.0 Heavy Gas Removal Analysis

The analysis carried out in section three is designed to examine the effectiveness of vortex separation systems for the removal of particulates the plume. In principle, however, this system can also be used to remove gaseous fission products and other heavy gasses from the plume. This section describes the governing equations for an invicid, axisymmetric flow of two mixed gasses in steady state, and then examines an analytical solution to these equations based on Dalton's partial pressure law. It also discusses some of the trade offs involved in designing gas removal systems, and considers whether they can actually be of practical value for removing radioactivity from nuclear rocket plumes.

4.1 Governing Equations

The invicid, steady state equations for an axisymmetric flow of two gasses consists of two sets of five equations. One set of equations is associated with each species, and each set of equations can be written in non-conservative form as follows [9]

Continuity:

$$\frac{\partial}{\partial r}(\rho u_r)+\frac{\partial}{\partial z}(\rho u_z)=-\frac{\rho u_r}{r} \tag{4.1-1}$$

Axial Momentum:

$$\frac{\partial(p+\rho u_z^2)}{\partial z}+\frac{\partial(\rho u_r u_z)}{\partial r}=-\frac{\rho u_r u_z}{r}+K_{ab}(u_z'-u_z) \tag{4.1-2}$$

Radial Momentum:

$$\frac{\partial(\rho u_r u_z)}{\partial z}+\frac{\partial(p+\rho u_r^2)}{\partial r}=-\frac{\rho u_r^2}{r}+\frac{\rho u_\theta^2}{r}+K_{ab}(u_r'-u_r) \tag{4.1-3}$$

Tangential Momentum:

$$\frac{\partial(\rho u_z u_\theta)}{\partial z}+\frac{\partial(\rho u_r u_\theta)}{\partial r}=-\frac{2\rho u_r u_\theta}{r}+K_{ab}(u_\theta'-u_\theta) \tag{4.1-4}$$

General Energy Conservation:

$$\frac{\partial(\frac{\gamma}{\gamma-1}pu_z+\tfrac{1}{2}\rho u_z|V|^2)}{\partial z}+\frac{\partial(\frac{\gamma}{\gamma-1}pu_r+\tfrac{1}{2}u_r|V|^2)}{\partial r}=\frac{\frac{\gamma}{\gamma-1}pu_r+\tfrac{1}{2}\rho u_r|V|^2}{r}+M \tag{4.1-5}$$

Where u' represents the velocity of the other gas (i.e. for the group of equations modeling the Hydrogen gas, u' represents the velocity of the fission products). It should be noted that these equations are very similar to equations (2.2-1) through (2.2-5). The only significant differences are the extra collision flux term in the momentum equations and the extra collision flux term in the energy equation. The $K(\Delta u)$ terms represent the change in momentum density due to collisions between the two gasses. K_{ab} is given by

$$K_{ab} = n_a m_{ab} \nu_{ab} \tag{4.1-6}$$

Where the subscript "a" refers to the first gas, and "b" refers to the second gas (i.e. for the group of equation modeling the hydrogen gas, n_a represent the number density of the hydrogen and n_b represents of the fission products). m_{ab} is the reduced mass, i.e.

$$m_{ab}^{-1} = m_a^{-1} + m_b^{-1} \tag{4.1-7}$$

ν_{ab} is the collision cross section and is given by

$$\nu_{ab} = n_b \overline{C}_{ab} Q_{ab} \tag{4.1-8}$$

And C_{ab} is the thermal approach velocity. Its value is discussed below. As one would expect, these terms are symmetric, so the momentum flux out of one gas is the same as the momentum flux into the other one.

M represents the rate of energy density change due to collisions, and is given by

$$M = K_{ab}\left[\left(\frac{3k}{m+m'}\right)(T'-T) + \frac{m'}{m+m'}\left[(u_r'-u_r)^2 + (u_z'-u_z)^2 + (u_\theta'-u_\theta)^2\right] + u_z(u_z'-u_z) + u_r(u_r'-u_r) + u_\theta(u_\theta'-u_\theta)\right] \tag{4.1-9}$$

This term is also symmetric.

The thermal approach velocity, $\overline{C}_{ab}$, is generally derived from the average thermal velocities of the two component gasses. The average thermal velocity of each component gas is given by

$$\overline{C} = \sqrt{\frac{8kT}{\pi m}} \tag{4.1-10}$$

Since the radioactive material in the flow has a molecular mass much larger that of the hydrogen, the thermal approach velocity is approximately equal to the average thermal velocity of the hydrogen alone. Therefore, equation (4.1-8) can be written as

$$\nu_{ab} = n_b \overline{C}_h Q_{ab} \tag{4.1-11}$$

4.2 Steady State Solution to Two-Fluid Equations

In general, it is not possible solve equations (4.1-1) through (4.1-5) analytically. However, in steady state, it can be shown that the radial momentum equation reduces to a form of Dalton's partial pressure law. Consider the radial momentum equation, (4.1-3). In situations where the centrifugal force term dominates over the collision flux term, i.e.

$$\frac{\rho\, u_\theta^2}{r} >> K_{ab}\left(u_r' - u_r\right) \tag{4.2-1}$$

it is possible to neglect the collision flux term. For the fission products, this term can be written as

$$\frac{\rho_u u_\theta^2}{r} >> K_{ab}\left(\Delta u_r\right) \tag{4.2-2}$$

Where ρ_u represents the density of the heavy gas. Using equation (4.1-6), this expression can be written as

$$\frac{m_u n_u u_\theta^2}{r} >> n_u m_{uh} \nu_{uh} \Delta u_r \tag{4.2-3}$$

From equation (4.1-7), it can be shown that if $m_u >> m_h$ the reduced mass is approximately equal to m_h. Therefore, (4.2-3) can be written as

$$\frac{m_u}{m_h}\frac{1}{\Delta u_r} >> \frac{r\nu_{uh}}{u_\theta^2} \tag{4.2-4}$$

This expression merits some discussion. If the separation chamber were of infinite length, as the flow traveled down the chamber, Δu_r would gradually approach zero as z approaches infinity. Eventually, the flow would reach a steady state, where bulk radial velocity of the two gasses would be equal and the net momentum flux between them would be zero.

Expression (4.2-4) determines how small Δu_r must be before the gas reaches this steady state condition. At the pressures and temperatures present in the separation chamber, ν_{uh} is on the order of 1 x 10^{11} sec^{-1}. Therefore, using equation (4.2-4), it can be shown that in a solid body rotation flow with ω = 10000 rad/sec, the momentum flux term can be neglected if

$$\frac{m_u}{m_h}\frac{1}{\Delta u_r} >> 1\times 10^7 \text{ sec/ m} \tag{4.2-5}$$

Even when the heavy to light gas mass ratio is of order 100, Δu_r must be of the order of 10^{-5} m/s before it is technically correct to neglect the collision flux term. Therefore, in a separation chamber of finite length, it is unlikely that a two gas flow would reach a steady state. Nevertheless, it is useful to examine the steady state solution because it represents a limiting case and reveals some of the fundamental tradeoffs involved in building a swirl based gas removal system.

Under the conditions defined by equation (4.2-5), the radial momentum equation for each component of a multi gas system is the same as the equation for a single gas, i.e.

$$\frac{\partial(\rho u_r)}{\partial t}+\frac{\partial(\rho u_z u_r)}{\partial z}+\frac{\partial(\rho u_r^2+p)}{\partial r}+\frac{\rho u_r^2-\rho u_\theta^2}{r}=0 \tag{4.2-6}$$

As long as the channel is of constant radius, there are no velocity variations in the z direction. Therefore, there is no direct coupling between the axial and radial velocity. In addition, in a steady state, the radial velocity will be zero, so equation (4.2-6) can be reduced to

$$\frac{\partial p}{\partial r}=\frac{\rho v_\theta^2}{r} \tag{4.2-7}$$

Which is simply the equation for the hydrostatic equilibrium of a fluid in centrifugal force field. Using the fundamental gas dynamics relations

$$p = nkT \qquad\qquad \rho = mn \tag{4.2-8}$$

This equation can be written as

$$\frac{dn_i}{n_i}=\frac{m_i}{kT}\frac{v_\theta^s}{r}dr \tag{4.2-9}$$

Where *i* is an index designating each species of gas. It should be noted that this equation is a form of Dalton's partial pressure law, which states that in equilibrium, each of the components of a system of several gasses acts as though the other components are not present.

In order to integrate this equation, It is necessary to make an assumption about the flow's tangential velocity distribution. If the flow is in solid body rotation, equation. (4.2-9) can be written as

$$\frac{dn_i}{n_i} = \frac{m_i}{kT}\Omega^2 r \tag{4.2-10}$$

This can be integrated to give

$$n_i = n_{i_o} \exp\left(\frac{m_i}{kT}\frac{\omega^2}{2}r^2\right) \tag{4.2-11}$$

It is interesting to note that it is now possible to define a characteristic length, l_c, as

$$l_c = \sqrt{\frac{2kT}{m_i}}\frac{1}{\omega} \tag{4.2-12}$$

Such that

$$n_i = n_{i_o} \exp\left(\left(\frac{r}{l_c}\right)^2\right) \tag{4.2-13}$$

l_c represents the portion of the outer radius that contains the bulk of the gas in the flow. For instance, if l_c = 0.05 m, around 60% of the gas is concentrated in the outer 0.05 m of the chamber. This result merits some discussion. According to equation (4.2-13), if (l_c <<R), the bulk of the gas will be concentrated near the outer wall of the cylinder in a steep exponential distribution. If, on the other hand, (l_c >>R), the gas will basically be uniformly distributed throughout the body of the cylinder. A gas's characteristic length is determined by its molecular mass, its temperature, and its angular rotation rate, but since the flow temperature is determined by the reactor, the flow rotation rate, ω, is the only true parameter in this expression.

In order to maximize the separation of the two gasses, it is desirably to design a channel whose radius is much less than the characteristic length of the fission products (l_p) and much greater than the characteristic length of the hydrogen gas (l_h). Fortunately, the ratio of the atomic masses is high enough that these lengths differ considerably. At temperatures typical of those found in a solid core nuclear rockets, the hydrogen remains in diatomic form and has an atomic weight of 2 AMU's. As shown in figure 1.1, fission products consist of a range of material scattered in a bimodal distribution, and the peaks of this distribution are at 95 and 140 AMU's. Based on these values, it can be shown that the ratio of l_p/l_h is 6.89 at the first peak and 8.37 at the second peak. These ratios are sufficiently high that is theoretically possible to achieve a high degree of separation between the hydrogen and the gaseous fission products in the flow.

Equation (4.2-13) can be further integrated to give the number of molecules in any section of the cylinder. In cylindrical coordinates, the integral can be written as

$$N = \int_0^{2\pi} \int_0^{r} n_{i_o} \exp\left(r/l_c\right)^2 r dr d\theta \tag{4.2-14}$$

and can be integrated to give

$$N = \pi n_{i_o} \left[\exp\left(R/l_c\right)^2 - 1 \right] \tag{4.2-15}$$

Equation (4.2-15) can now be used to calculate the amount of fissile material and hydrogen removed from the flow. For a fixed skimming ratio, the removal efficiency is given by

$$\eta = \frac{\exp\left(R_s/l_c\right)^2 - 1}{\exp\left(R_o/l_c\right)^2 - 1} \tag{4.2-16}$$

Using equation (4.2-16), it is possible to calculate the removal efficiency, η, as a function skimming ratio and channel length. Figure 4.1 shown a graph of the efficiency vs. skimming ratio and R*, where R* is defined as the ratio of the chamber length to the characteristic length, i.e.

$$R^* = \frac{R_o}{l_c} \tag{4.2-17}$$

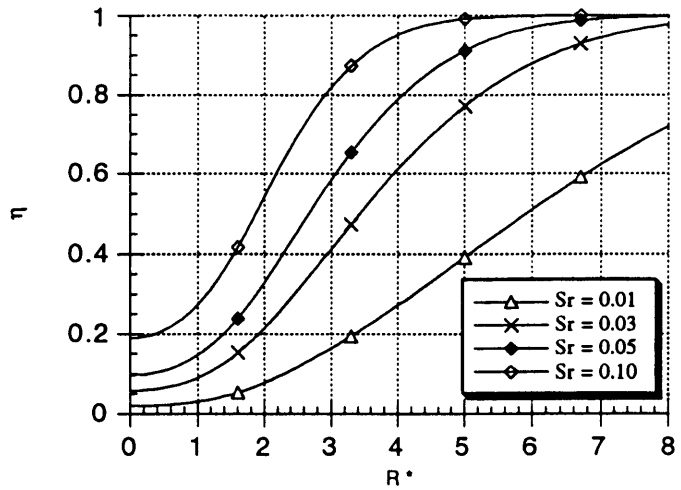


Figure 4.1: Heavy Gas Removal Efficiency vs. Length and Skimming Ratio

As one would expect, figure 4.1 shows that higher values of R* (i.e. higher rotation rates) produce higher removal efficiencies. It should be noted, however, that these curves apply to all components of the flow, including the hydrogen. Therefore, while larger values of R^* do result in the removal of more fissile material, they also result in the removal of more hydrogen propellant. To examine this effect, it is convenient to define another parameter, the material to propellant removal ratio, as follows:

$$\sigma = \frac{\eta_p}{\eta_h} \tag{4.2-18}$$

Where η_p is the removal efficiency for the fission product and η_h is the removal efficiency for the hydrogen propellant. Figure 4.2 shows a graph of this removal ratio vs. skimming ratio and R^* for a fission product with a molecular weight of 95gm/mole.

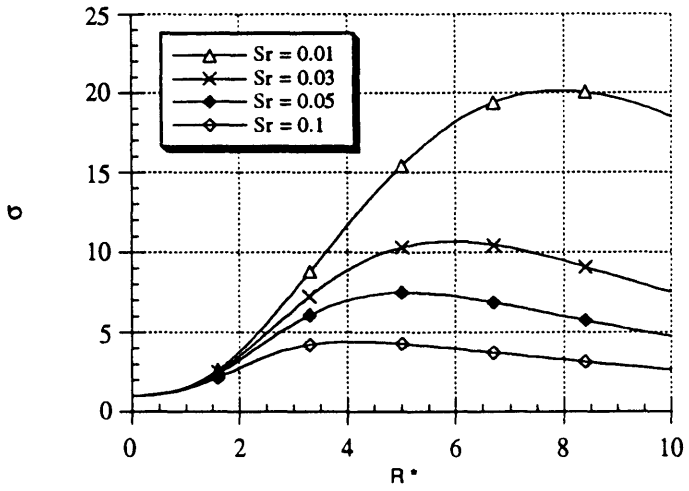


Figure 4.2: Gas Removal Ratio vs. Length and Skimming Ratio

Figure 4.2 shows for any given skimming ratio, there is a maximum achievable value of σ. Although the removal ratio initially increases with R^*, σ actually declines as R^* approaches infinity. This occurs because swirl pushes both the fission product and the hydrogen propellant to the outside of the flow. Initially, increasing the swirl velocity increases the amount of heavy material collected by the skimmer, but as the rotation rate continues to increase, there is also an increase in the amount of hydrogen collected by the skimmer. As a result, the removal ratio decreases as l_h approaches R_o. It should also be noted that in general, σ increases as the skimming ratio decreases. This occurs because the fissile material's distribution curve described by equation (4.2-16) becomes much steeper as R_s approaches R_o. However, in contrast to this result, figure 4.1 shows that for a given value of R^*, as the skimming ratio decreases, η_f decreases also. Therefore, there is a tradeoff between the amount of fissile material removed from the flow and propellant lost in removing that material. Low skimming ratios will limit propellant loss, but remove relatively low amounts of fissile material, while high skimming ratios remove large amounts of material with a relatively high performance penalty.

Using equation (4.2-12), it is also possible to estimate the swirl velocity necessary to achieve a given value of R^* for a chamber of a given size. Table 4.1 shows a series of these results for a separation chamber with the geometry shown in figure 1.2.

Ω (rad/sec)	lc (meters)	R*
100	44.66	0.003
500	8.93	0.016
1000	4.47	0.031
2000	2.23	0.063
4000	1.12	0.125
6000	0.74	0.188
10000	0.45	0.313
15000	0.30	0.470
20000	0.22	0.627
30000	0.15	0.940
40000	0.11	1.254
50000	0.09	1.567
60000	0.07	1.881
70000	0.06	2.194
80000	0.06	2.508
90000	0.05	2.821
100000	0.04	3.135

Table 4.3: Length Ratio vs. Rotation Rate for Gas Removal

Table 4.3 shows that it is impossible to obtain useful values of R^* without using unreasonably high rotation rates. The conditions which determine the exact limits on the rotation rate are discussed in section five, but it should be noted that at the relatively high rotation rate of 10000 rad/sec, a system with a skimming ratio of 0.1 would have an efficiency of only 20% and would remove almost as much hydrogen as it did fissile material. To obtain useful values of R^*, it is necessary to use rotation rates almost an order of magnitude larger, in a range where the flow's tangential velocity would exceed its axial velocity by an order of magnitude or more.

Although Table 4.3 applies only to steady state flows, the magnitude of its results imply that vortex separation will not be a practical means of removing gaseous radioactive material. Since the steady state case represents the limiting case as z approaches infinity, it is reasonable to assume that the efficiencies seen in the separation chamber will be less than or equal to those shown in figure 4.1, and there is certainly no reason to believe that the flow's characteristic length will shrink by an order of magnitude. In order to achieve reasonable steady state efficiencies, it is necessary to exceed reasonable rotation rates by an order of magnitude; a condition which is clearly not practical, at least not for the system studied in this paper. In principle, however, if a high enough swirl velocity could be achieved, this system could be used to remove material from the flow. This topic is discussed in greater detail in section five below.

In summary, in a steady state, the invicid radial momentum equation for a mixture of gas reduces to the form of Dalton's partial pressure law shown in equation (4.2-9). The conditions under which this assumption is valid are defined by equation (4.2-5) and, in general, the conditions in the separation chamber do not justify this assumption. Nevertheless, it is possible to solve equation (4.2-9) analytically, and the results indicate that in steady state, this system is governed by a characteristic length and that for any given skimming ratio, there is an optimum length ratio R^* which removes the greatest amount of radioactive material relative to the amount of hydrogen gas removed from the flow. Unfortunately, the only parameter under the designer's direct control is the swirl velocity ω, and table 4.3 indicates that any practical device would require swirl velocities well beyond those which can be achieved in a practical device. Finally, although these results are based on a steady state solution and are not technically valid in the separation chamber, they still cast a great deal of doubt on the use of vortex separation to remove gaseous radioactive material from the flow.

5.0 Limits on System Efficiency

From the results presented to this point, it is clear that higher swirl rates lead to lower separation times and thus to shorter separation chambers. Therefore, it is generally desirable to make the swirl rate as high as possible. However, there are practical limits to the amount of swirl which can be imparted to the flow without causing performance losses. In particular, an absolute limit is set by the need to avoid vortex breakdown. Vortex breakdown is a phenomena in which the structure of the swirling flow is suddenly destroyed by a spontaneous and pronounced retardation of the flow and a divergence of stream surfaces near the core of the vortex.[10] If breakdown were to occur in the separation chamber, it would destroy the structure of the flow and cause a large drop in its velocity. Breakdown is generally found in highly swirling flows and in the presence of adverse pressure gradients, though boundary layer effects can produce breakdown in parallel geometries like the one proposed for the separation chamber. Although the mechanism by which breakdown occurs is poorly understood, experimental data indicates that the possibility of vortex breakdown exists in situations where the flow turning angle is greater than 40 degrees.[11] Therefore, for a chamber with the geometry shown in figure 1.2, the threat of vortex breakdown precludes the use of solid body rotation rates greater than 6500 rad/sec. and of constant angle turning rates greater than 40 degrees.

Another practical limit is set by the need to minimize I_{sp} losses. Because energy is diverted from the axial to the tangential components of the velocity, adding swirl to the flow lowers its axial velocity. However, as the flow expands down the nozzle, the conservation of angular momentum causes the transfer of energy back form the swirl velocity into the core of the flow. This tends to mitigate any losses due to the presence of swirl in the nozzle. To analyze these losses, the flow solver described in section three was used to model solid body rotation flows down a nozzle with a 50:1 area ratio. The results showed that even with an initial swirl rate of 10000 rad/sec, the I_{sp} loss at the exit was still only 0.1%. Since mass flow was generally preserved to within 2.5% in these simulations, these results show that the overall I_{sp} losses are on the order of 2% or less (i.e. too small to be resolved by the solver). This in turn indicates that at the swirl levels of interest, the system I_{sp} losses due to the presence of swirl are negligible.

Finally, it should be noted that although the filtered hydrogen is returned to the reactor, the mass flow removed by the skimmer still lowers the thrust of the system. The thrust loss is proportional to the mass flow removed and is therefore strongly related to the skimming ratio. The results presented in section three, however, are generally made using the non-physical skimming ratio of zero. In reality, the skimming ratio would be the smallest

practical value which could actually catch the particles. It is not clear, however, just how small one could make the skimming ratio, as the factors which would determine would entail detailed consideration of the boundary layer and are therefore beyond the scope of this study. It should be noted that equation (4.2-15) shows that for small values of s_r, the fraction of the mass flow removed is approximately twice the skimming ratio This means that a skimming ratio of 0.01 (a 1.4 mm slot for the geometry shown in figure 1.2) would cause a thrust loss of 2%. Therefore, a practical particle vortex separation system could easily end up causing significant thrust losses.

6.0 Conclusions

This paper contains a detailed analysis of a swirl based separation system designed to remove radioactive material from the plumes of nuclear rockets. This system is designed to remove radioactive material from the plume before it enters the nozzle, to filter out this material, and then return the cleaned hydrogen to the reactor where it can be used again as fuel. A computational analysis and simple analytical model are used to examine the ability of this system to particulates from the plume, and a steady state analytical solution is used to examine its ability to remove gaseous radioactive material.

The computation analysis carried out in section three indicates that vortex separation is able to remove some particles with efficiencies in excess of 99%, but that the effectiveness of the system depends heavily on the size and densities of the particles in the flow. Due to a lack of experimental data, it is impossible to simply state whether or not the system is suitable for use on real nuclear rockets. However, the computational and analytical work carried out in this paper identifies the critical velocity, v_c, as the parameter which determines the effectiveness with which the system removes a particle of given size and density. Once experimental data is available, it will be possible to use equation (3.3.2-3) to calculate the relevant critical velocities and determine the effectiveness of swirl based particle removal systems. The computational analysis also identified one other parameter of interest to the designer: the dimensionless frequency, ν. The dimensionless frequency is a function of the critical velocity and determines the relationship between the flow rotation rate and the length of the separation chamber, and the performance of both constant angle and solid body rotation systems are governed by these two critical parameters. In addition, there is one other parameter, the skimming ratio, which determines the thrust losses associated with this system, but has little direct effect on the system's efficiency. Using these three parameter, it should be possible to conduct basic design studies on swirl based particle separation systems.

In contrast to these results, section four describes a steady state solution to the invicid, axisymmetric two gas equations which indicates that vortex separation is probably not a practical system for removing gaseous material from nuclear rocket plumes. The results outlined in table 4.3 show that in the steady state, reasonably removal efficiencies can not be achieved without using swirl velocities an order of magnitude larger than the practical limit. While this steady state solution does not necessarily apply to this system, since it represents a limiting case, it is unlikely that considerably better performance can be achieved in the separation chamber. Therefore, it is unlikely that this system will prove effective for the removal of gaseous radioactive material from the plume.

In conclusion, the vortex separation system proposed in this paper shows some promise. While it probably can not be used to remove gaseous material from the plume, it should be able to remove substantial amounts of the particulates in the plume before they leave the nozzle. There may, however, be a substantial mass penalty associated with this system, particularly if it is necessary to remove particles with non-intermediate critical velocities. Further experimental work is required to verify these computational results and to determine the size and composition of the radioactive material in actual nuclear rocket plumes. Once such data is available, the material presented in this paper should provides the tools necessary to estimate the effectiveness of vortex separation systems.

7.0 References

[1] See for instance W.L. Kirk. "Nuclear Furnace-1 Test Report." N-Division and CNC-Division, Los Alamos Scientific Laboratory, Los Alamos NM, Rept. LA-5189-MS, March 1973, pp. 62-74.

[2] R.W. Bussard and R.D. DeLauer. *Fundamentals of Nuclear Flight.* McGraw-Hill Book Company, New York, 1965, pg. 265.

[3] Dale A. Anderson, John C. Tannehill, and Richard H. Pletcher. *Computational Fluid Mechanics and Heat Transfer.* Hemisphere Publishing Corporation, New York, 1984, pg. 523.

[4] G.K. Bachelor. *An Introduction to Fluid Dynamics.* Cambridge University Press, Cambridge, England, 1967, pg. 160.

[5] J.C. Traineau, P. Kuentzmann, M. Prévost, P. Tarrin, and A. Delfour. "Particle Size Distribution Measurements in a Subscale Motor for the Ariane 5 Solid Rocket Booster," AIAA Paper 92-3049, AIAA/SAE/ASME/ASEE 28th Joint Propulsion Conference, Nashville TN, July 1992.

[6] Frank White. *Viscous Fluid Flow. Second Edition,* New York, 1974, pg. 182.

[7] J.O. Hirschfelder, C.F. Curtiss and R.B. Bird. *Molecular Theory of Gases and Liquids.* John Wiley & Sons, Inc., New York, 1954.

[8] J.T. Vanderslice, S. Weissman, E.A. Mason, and R.J. Fallon, "High-Temperature Transport Properties of Dissociating Hydrogen." *Physics of Fluid,* Vol. 5, No. 2, Feb. 1962, pp. 155-64.

[9] These equations are partially based on work done by Eli Niewood (Ph.D., 1993), but the general procedure for deriving them is outlined in: J.M. Burgers. *Flow Equations for Composite Gasses.* Academic Press, New York, 1969.

[10] M.G. Hall. "Vortex Breakdown." *Annual Review of Fluid Mechanics,* Vol. 4, 1972, pg. 195.

[11] Ibid, pg. 198.

Appendix A: Tabulated Effective Collision Integral for H_2

This appendix contains the effective collision integral for H_2-H_2 collisions, as tabulated by Vanderslice et al. in "High Temperature Transport Properties of Dissociating Hydrogen." *Physics of Fluid*, Vol. 5, No. 2, Feb. 1962, pp. 155-64. This shows the value of the integral in units of 10^{-20} m^2.

Temperature (K)	Ω (2,2)
300	7.34
500	6.75
1000	6.00
1500	5.59
2000	5.33
2500	5.02
3000	4.73
3500	4.50
4000	4.29
4500	4.12
5000	3.97
5500	3.83
6000	3.71
6500	3.60
7000	3.50
7500	3.41
8000	3.32
8500	3.24
9000	3.17
9500	3.10
10000	3.03
11000	2.91
12000	2.81
13000	2.71
14000	2.62
15000	2.54

Appendix B: Source Code

This Appendix B contains the source code of the computer program used to model particle-fluid interactions in the separation chamber. This model actually consists of two distinct parts: a fluid modeling code to solve the axisymmetric Euler Equations, and a particle tracking code which acts as a post processor. All codes are written in C and run on a Silicon Graphics Iris Indigo workstation. The first section is a listing of the fluid modeling code. This code consists of 9 modules and two header files. These files are named:

main.c: contains the main routine
boundary.c: calculates boundary conditions
damping.c: calculates damping coefficients
init.c: sets up initial flow conditions
integrate.c: calculates fluxes and carries out time integration
io.c: contains input/output routines
memory.c:allocates dynamic memory
metrics.c: calculates grid metrics
time.c: calculates timestep

And the two header files

nozzle.h: contains general header information
globals.h: contains a listing of all global variables

```
/* Main for Program Nozzle */
/* Nozzle is an Axisymmetric invicid Euler code based on the */
/* Macormick predictor-corrector method with fourth order and */
/* Second Order smoothing. */
/* All components are written in C, and the code is in dimensional form */
/* Written by David Oh (bamf) for his master's thesis */
/* August 6, 1992 */

/* This is file main.c */
#define ERRORPLOT 0
#include "nozzle.h"
#include <math.h>
#include <stdio.h>

/* GLOBAL DECLARATIONS */
/* For explaination of variables, see globals.h */
char predictor;
double ftheta,
       mdot,
       time,
       twall;
double dxi2, deta2, dxi4, deta4, dcenterxi2, dcentereta2;
int iter,
    nxi, neta;                  /* number of xi and eta grid cells */
double **d;                         /* Fourth Order damping term */
double **d2;                        /* Second Order damping term */
double *hgin, *ugin, *theta;
double **pg, **pgp, **pg0, **r,  **tg, **z;
/* For metrics */
double **dxidz, **dxidr, **detadz, **detadr, **xirr, **xizz, **etarr;
double **etazz, **jacobian, **drdeta, **drdxi, **dzdeta, **dzdxi;
vector5 **f1, **f2, **fg;
vector5 **g1, **g2, **gg;
vector5 **sg, **sxg, **syg, **ugp;
vector5 **ug;

/* And one for debugging */
FILE *debuggingoutput;

main()
{
  char fname[40];          /* input file name */
  char gridname[40];       /* Grid file name */
  char answer[5], continued; /* Answer to "continued?" question */
  double grms, normalizedgrms;
  int step;
  FILE *history;

  /* Get relevant information */
  printf("\nRun name: ");
  scanf ("%s", fname);
  printf ("\nGrid name: ");
  scanf ("%s", gridname);
  printf ("\nContinued run (y/n)? ");
  scanf ("%s", answer);
  if (answer[0] == 'y')
    continued = TRUE;
  else
    continued = FALSE;
  printf ("\nHow many integrations? ");
```



```
scanf ("%d", &step);
printf("\nDamping Coeff (one for defaults) %f %f %f %f",
                                        DXI4, DETA4, DXI2, DETA2);
printf("\n    Enter in order d4, d2, dcenter2" );
scanf("%lf", &dxi4);
if (dxi4 < 1.0+ EPSILON && dxi4 > 1.0-EPSILON) {
  dxi4 = DXI4;
  dxi2 = DXI2;
  deta4 = DETA4;
  deta4 = DETA4;
  dcenterxi2 = DXI2;
  dcentereta2 = DETA2;
}
else {
  scanf("%lf %lf %lf", &deta4, &dxi2, &deta2);
}

/* READ GRID */
/* Routine in io.c */
getgrid(gridname);

/* This routine in metrics.c */
calculatemetrics();

/* This routine in boundary.c */
calculateWallAngle();

/* Set up initial conditions */
/* This routine is in io.c */
if (continued == TRUE) {
  getflow (fname);
}
else {
  printf("\nInitializing New Flow.");
  initializeflow();
}

/* Some run information, just to help keep runs straight */
printf("\nInlet Swirl: %f", INLET_ROTATION_RATE);
printf("\n4th Order Damping: %g %g", dxi4, deta4);
printf("\n2nd Order Damping: %g %g", dxi2, deta2);
printf("\nCenterline 2nd Order Damping: %g %g", dcenterxi2, dcentereta2);

history = fopen ("history", "a");
/* Integrate the equations */
for ( ; step >0; --step) {

  /* This routine contained in integrate.c */
  printf("\nIteration: %d of %d ", iter, iter+step);
  fprintf(history, "\n%d ", iter);
  integrate();

  /* This routine in main.c */
  calculateError(&grms, &normalizedgrms);

  printf("\n        Error: %f, %f", grms, normalizedgrms);
  fprintf(history, "%f", normalizedgrms);
  iter = iter + 1;
}
```

```
  fclose(history);

  /* Write output Files */
  /* This routine is in io.c */
  writeflow (fname);
}

calculateError(grms, normalizedgrms)
double *grms, *normalizedgrms;
{
  int i,j;

  *grms = 0.0;
  *normalizedgrms = 0.0;

  for (j = 0; j < neta; ++j)
    for (i = 0; i < nxi; ++i) {
      *grms = *grms + SQ(pg[i][j] -pg0[i][j]);
      *normalizedgrms = *normalizedgrms + SQ(pg[i][j] - pg0[i][j])
/ SQ(pg0[i][j]);
    }

  *grms = sqrt (*grms / (double)(nxi * neta));
  *normalizedgrms = sqrt (*grms / (double)(nxi*neta));
}
```

```
/* boundary.c */
/* Uses REFLECTING boundary conditions */
/* Module for program nozzle */
/* Sets the boundary conditions */
/* Being Written 5/15/92 */

#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

/* These are C PREPROCESSOR flags that determine what boundary condition */
/* Is used.  Note that I've used the preprocessor is such a way that    */
/* Setting one of these flags to FALSE means that that entire section   */
/* Is not compiled at all.  This saves execution time, etc.             */
/* Setting a flag to true means that that boundary condition is applied */
/* C stands for "Corrector Step" and P stands for "Predictor Step" so   */
/* CENTERLINEC 1 means "apply the centerline boundary condition in the  */
/* corrector step */
#define CENTERLINEC 1
#define UPPERWALLC 1
#define OUTLETC 1
#define CENTERLINEP 1
#define UPPERWALLP 1
#define OUTLETP 1

/* These define statements set the inlet conditions */
#define INLETSUBSONICP 1
#define INLETSUPERSONICP 0
#define INLETSUBSONICC 1
#define INLETSUPERSONICC 0
#define INLETMACH 1.18
/* For non-solid body rotation */
#define INLET_TANGENTIAL_VELOCITY_RATIO 0.0

/* Ok;  Outlet subsonic conditions */
#define OUTLETSUBSONICP 0
#define OUTLETSUBSONICC 0

/*#define OUTLETTEMPERATURE 2440.21
#define OUTLETMACH 0.35  */

/*#define OUTLETTEMPERATURE 2452.8
#define OUTLETMACH 0.31*/

#define OUTLETMACH 0.3
#define OUTLETTEMPERATURE 2455.79

/*#define OUTLETMACH 0.28
#define OUTLETTEMPERATURE 2461.42 */
/*#define OUTLETMACH 0.4
#define OUTLETTEMPERATURE 2422.48 */

/* 1 = no relaxation; 0 = only the old value (no change) */
#define RELAX_FACTOR 1.0
#define INLET_RELAX_FACTOR 1.0

boundary()
/* Applies those invicid boundary conditions */
/* What fun */
{
  int i, j;
  double a, b, c;
  double ainlet;                    /* Speed of Sound in Inlet */
  double new_pressure, pressure;
  double jminus, jplus, outlettemperature, rho1, t1;
  double ur,                        /* Velocity Components */
         utheta,
         uz,
         uznew;                 /* Used for relaxed B.C. */
  double utot2;

  double delta_theta, theta2, dot;
  double alpha,                  /* Centerline ratio of p/rho^gamma */
         Cp,                  /* Specific heat at constant Pressure */
         Cv,                   /* Specific hear at constant Volume */
         htot,                           /* Total Enthalpy of Flow */
         mach,                         /* Local Axial Mach Number */
         omegaz,                    /* Axial component of vorticity */
         rho,                                    /* Local density */
         p2, rho2,
         s,                                    /* Entropy of Flow */
         dsdr,                               /* Entropy derivative */
         sum;               /* Running sum for trapizoid integrator */
  vector3 normal, tangential_velocity, velocity;
  double velocity_magnitude;
  double pnew;

  /* Couple of generally useful calculations */
  Cp = (GAMMA / (GAMMA-1.0)) * RGAS;
  Cv = (1.0 / (GAMMA-1.0)) * RGAS;
  htot = CHAMBER_TEMP * Cp;

  /* PREDICTOR BOUNDARY CONDITIONS */
  if (predictor) {

#if INLETSUBSONICP
    /* Inflow Boundary (left boundary) */
    /* This applies a constant total pressure and temperature requirement */
    /* Mass flow is set by the upstream choked throat */
    /* The nomenclature is switched around for convenience;  take it out */
    /* later if you need more speed */
    /* This must be executed before the Upper Wall conditions */

    for (j = 0; j < neta; ++j) {
      /* Calculate j-minus Riemann Invarient from upstream conditions */
      uz = ugp[1][j].b/ugp[1][j].a;
      rho1 = ugp[1][j].a;
      t1 = pgp[1][j] / (rho1*RGAS);
      jminus= uz-(2.0/(GAMMA-1.0))*sqrt(GAMMA*RGAS*t1);

      /* Get swirl profile data */
      swirlProfile(r[0][j], r[0][neta-1], &omegaz, &utheta);

      /* Now calculate the temperature and axial velocity in row 0 */
      a = 1.0 + (2.0 / (GAMMA - 1.0));
      b = - 2.0 * jminus;
      c = SQ(jminus) - (CHAMBER_TEMP*(4.0*GAMMA*RGAS)
/ SQ(GAMMA- 1.0)) + 2.0 * SQ(utheta)/(GAMMA-1.0);
```

```
      /* What sign should go before the sqrt is a little unclear. */
      uz = (-b + sqrt(b*b - 4.0*a*c))/(2.0*a);
      tg[0][j] = (SQ(GAMMA-1.0)/(4.0*GAMMA*RGAS)) *
(SQ(uz)-2.0*jminus*uz+SQ(jminus));

      /* Calculate the entropy of the flow */
      /* This is going to attempt to be a rather clever trapezoid solver */
      if (j == 0) {
mach = uz / sqrt (GAMMA * RGAS * tg[0][j]);
pgp[0][j]= CHAMBER_P/(pow (1.0+((GAMMA-1.0)/2.0) * SQ(mach),
   GAMMA/(GAMMA-1.0)));
rho = pgp[0][j] / (RGAS * tg[0][j]);
alpha = pgp[0][j] / pow(rho, GAMMA);
s = 0;
sum = 0;
      }
      else {
dsdr = -(Cp*omegaz*utheta) / (htot - 0.5*(SQ(uz)+SQ(utheta)));
s = (r[0][j]/(2.0*(double)(j))) * (2.0 * sum + dsdr);
sum += dsdr;
      }

      /* Get the pressure and density from the entropy */
      rho = pow ( (1.0/alpha)*(htot- 0.5*(SQ(uz)+SQ(utheta)))*
 ((GAMMA-1.0)/GAMMA) / exp(s/Cv), 1.0/(GAMMA-1.0));
      pgp[0][j] = RGAS * rho * tg[0][j];

      /* CHECK TO SEE IF MACH NUMBER IS INDEED LESS THAN 1 */
      /* IF THE MACH NUMBER IS GREATER THAN 0.9, SET IT TO 0.9 */
      ainlet = sqrt(GAMMA*RGAS*tg[0][j]);
      if ( (uz/ainlet) > 0.9) {
printf ("Inlet Mach exceeds 1\n");
uz=ainlet * 0.9;
tg[0][j]=CHAMBER_TEMP/(1.0+((GAMMA-1.0)/2.0)*SQ(0.9));
pgp[0][j]= CHAMBER_P/(pow (1.0+((GAMMA-1.0)/2.0) * SQ(0.9),
   GAMMA/(GAMMA-1.0)));
      }

      /* Calculate the state vector */
      if (j != 0) {
rho= pgp[0][j] / (RGAS*tg[0][j]);
ugp[0][j].a = rho;
ugp[0][j].b = rho*uz;
ugp[0][j].c = 0.0;
ugp[0][j].e = rho * utheta;
utot2 =
  (SQ(ugp[0][j].b)+SQ(ugp[0][j].c)+SQ(ugp[0][j].e))/SQ(ugp[0][j].a);
ugp[0][j].d = (1.0/(GAMMA-1.0))*pgp[0][j] + 0.5*ugp[0][j].a*utot2;
      }
    }
#endif

#if INLETSUPERSONICP
/* Predictor Inlet, Supersonic Conditions Only */
/* This sets supersonic inlet conditions which have nothing to do with */
/* Anything from the flow */
    for (j = 0; j < neta; ++j) {
      tg[0][j] = CHAMBER_TEMP / (1.0 + ((GAMMA-1.0)/2.0)*SQ(INLETMACH));
      pgp[0][j] = CHAMBER_P /
 pow((1.0 + ((GAMMA-1.0)/ 2.0)*SQ(INLETMACH)), GAMMA/(GAMMA-1.0));
```



```
      rhogb[j] = pgp[0][j] / (RGAS * tg[0][j]);
      if (((double) (j) / (double)(neta-1)) < 0.2)
ugp[0][j].e = rhogb[j]*INLETMACH*sqrt(GAMMA*RGAS*tg[0][j])*
  tan(INLET_TANGENTIAL_VELOCITY_RATIO)*
    5.0 * ((double)(j)/(double)(neta-1));
      else
ugp[0][j].e = rhogb[j]*INLETMACH*sqrt(GAMMA*RGAS*tg[0][j])*
  tan(INLET_TANGENTIAL_VELOCITY_RATIO);
      ugp[0][j].a = rhogb[j];
      if (j == 0)
pressure = 0.0;
      else
pressure = pressure + SQ(ugp[0][j].e)/(ugp[0][j].a*r[0][j])
  * (r[0][j] - r[0][j-1]);
      pgp[0][j] = pgp[0][j] + pressure;
      ugp[0][j].b = ugp[0][j].a * INLETMACH *  sqrt(GAMMA*RGAS*tg[0][j]);
      ugp[0][j].c = 0.0;
      ugp[0][j].d = (1.0/(GAMMA - 1.0)) * pgp[0][j] +
0.5 * SQ(ugp[0][j].b)/ ugp[0][j].a;
    }
#endif

#if CENTERLINEP
    /* Predictor Centerline (bottom) boundary */
    /* Applies a symmetry condition, with pressure calculated using */
    /* Solid body rotation assumptions */
    for (i = 0; i < nxi; ++i)  {
      /* Set 0 radial and tangential velocity */
      ugp[i][0].c = 0.0;
      ugp[i][0].e = 0.0;

      /* Get pressure from Solid Body Rotation Rules */
      pgp[i][0] = pgp[i][1] - 0.5 * ugp[i][1].a *
SQ(ugp[i][1].e/ugp[i][1].a);

      /* Assume constant energy */
      ugp[i][0].d = ugp[i][1].d;

      /* Set constant axial velocity, and calculate density */
      ugp[i][0].a = (2.0 / SQ(ugp[i][1].b / ugp[i][1].a)) *
(ugp[i][0].d - (1.0/(GAMMA-1.0)) * pgp[i][0]);
      ugp[i][0].b = ugp[i][0].a * (ugp[i][1].b / ugp[i][1].a);

      ugp[i][0].d = (1.0 / (GAMMA-1.0)) * pgp[i][0] +
0.5 * ugp[i][0].a * SQ(ugp[i][0].b/ugp[i][0].a);

      tg[i][0] = pgp[i][0] / (ugp[i][0].a * RGAS);
    }
#endif

#if UPPERWALLP
    /* Predictor Upper Wall Boundary        */
    /* A simple method designed to insure:  */
    /*    1) Zero Pressure Gradient         */
    /*    2) Tangential Velocity Vector     */
    /* Preserving the magnitude of the velocity vector doesn't seem  */
    /* To work, so I'm going to cut the normal component, as is      */
    /* conventional */

    /* The "x" direction is the axial */
```

```
    /* The "y" direction is radial */
    /* The "z" direction is tangential, out */

    for (i = 1; i < nxi-1; ++i) {
      velocity.x = ugp[i][neta-2].b / ugp[i][neta-2].a;
      velocity.y = 0.0;
      velocity.z = 0.0;

      normal.x = cos(theta[i]);
      normal.y = sin(theta[i]);
      normal.z = 0.0;

      /* Theta2 will is the angle by which the velocity needs to be */
      /* turned */
      dot = velocity.x * normal.x + velocity.y * normal.y
        + velocity.z * normal.z;

      velocity_magnitude = velocity.x;
      theta2 = acos(dot/velocity_magnitude);
      theta2 = PI/2.0 - theta2;

      velocity.y = tan(theta2) * velocity_magnitude;
      velocity.z = ugp[i][neta-2].e / ugp[i][neta-2].a;

      /* Construct new State Vector */
      ugp[i][neta-1].a = ugp[i][neta-2].a;
      ugp[i][neta-1].b = ugp[i][neta-1].a * velocity.x;
      ugp[i][neta-1].c = ugp[i][neta-1].a * velocity.y;
      ugp[i][neta-1].e = ugp[i][neta-1].a * velocity.z;
      /* relax pressure */
      new_pressure = pgp[i][neta-2] +
(SQ(ugp[i][neta-2].e)/(ugp[i][neta-2].a * r[i][neta-2]))
 *(r[i][neta-1]-r[i][neta-2]);
      pgp[i][neta-1] =
((1.0 - RELAX_FACTOR) * pgp[i][neta-1]
         + (RELAX_FACTOR * new_pressure)) / 1.0;
      ugp[i][neta-1].d = (1.0 / (GAMMA-1.0)) * pgp[i][neta-1] +
0.5 * ugp[i][neta-1].a * (SQ(velocity.x) + SQ(velocity.y) +
  SQ(velocity.z));
      tg[i][neta-1] = pgp[i][neta-1] / (RGAS * ugp[i][neta-1].a);
    }

#endif

#if OUTLETP
    /* Predictor Outlet Boundary */
    /* Conditions assume a supersonic outflow and extrapolate flow */
    /* variables values. */
    for (j = 0; j < neta; ++ j) {
      ugp[nxi-1][j].a = 2.0 * ugp[nxi-2][j].a - ugp[nxi-3][j].a;
      ugp[nxi-1][j].b = 2.0 * ugp[nxi-2][j].b - ugp[nxi-3][j].b;
      ugp[nxi-1][j].c = 2.0 * ugp[nxi-2][j].c - ugp[nxi-3][j].c;
      ugp[nxi-1][j].e = 2.0 * ugp[nxi-2][j].e - ugp[nxi-3][j].e;
      pgp[nxi-1][j] = 2.0 * pgp[nxi-2][j] - pgp[nxi-3][j];
      /* To keep very strange things from happening */
      if (pgp[nxi-1][j] < 0.0)
pgp[nxi-1][j] = 0.1;
      tg[nxi-1][j] = pgp[nxi-1][j] / (RGAS * ugp[nxi-1][j].a);
      ugp[nxi-1][j].d = (1.0 / (GAMMA-1.0)) * pgp[nxi-1][j] +
0.5 * ((SQ(ugp[nxi-1][j].b) + SQ(ugp[nxi-1][j].c) +
```



```
      SQ(ugp[nxi-1][j].e))/ugp[nxi-1][j].a);
    }

#endif

#if OUTLETSUBSONICP
    /* Predictor Outlet boundary for subsonic flow */
    for (j = 0; j < neta; ++j) {
      outlettemperature = CHAMBER_TEMP / (1.0 + (GAMMA-1.0)/2.0 *
  OUTLETMACH * OUTLETMACH);
      jplus = ugp[nxi-2][j].b / ugp[nxi-2][j].a + 2.0 *
sqrt(GAMMA * RGAS * tg[nxi-2][j]) / (GAMMA - 1.0);
      jminus = OUTLETMACH * sqrt(GAMMA*RGAS*outlettemperature) - 2.0 *
sqrt(GAMMA*RGAS*outlettemperature) / (GAMMA-1.0);
      s = log(pgp[nxi-2][j]) - GAMMA * log(ugp[nxi-2][j].a);
      ugp[nxi-1][j].a = pow((SQ(jplus - jminus)* (GAMMA-1.0) * (GAMMA-1.0)) /
                        (16.0 * GAMMA * exp(s)),
                              1.0 / (GAMMA-1.0));
      ugp[nxi-1][j].b = ugp[nxi-1][j].a * 0.5 * (jplus + jminus);
      ugp[nxi-1][j].c = ugp[nxi-1][j].a * ugp[nxi-2][j].c / ugp[nxi-2][j].a;
      ugp[nxi-1][j].e = ugp[nxi-1][j].a * ugp[nxi-2][j].e / ugp[nxi-2][j].a;
      ugp[nxi-1][j].d = ugp[nxi-1][j].a * SQ(jplus - jminus) * (GAMMA-1.0) /
(16.0 * GAMMA) + 0.5 * (SQ(ugp[nxi-1][j].b)
+ SQ(ugp[nxi-1][j].e)) / ugp[nxi-1][j].a;
      pgp[nxi-1][j] =  ugp[nxi-1][j].a * SQ(jplus - jminus) * (GAMMA-1.0)
 * (GAMMA-1.0)/ (16.0 * GAMMA);
    }
#endif

  } /* Ends "if (predictor)" */

/*************************************************************************/
/* Or, deal with the corrector step */
/*************************************************************************/

  else {
#if INLETSUBSONICC
    /* Inflow Boundary (left boundary) */
    /* This applies a constant total pressure and temperature requirement */
    /* Mass flow is set by the upstream choked throat */
    /* The nomenclature is switched around for convenience;  take it out */
    /* later if you need more speed */

    for (j = 0; j < neta; ++j) {
      /* Calculate j-minus Riemann Invarient from upstream conditions */
      uz = ug[1][j].b/ug[1][j].a;
      rho1 = ug[1][j].a;
      t1 = pg[1][j] / (rho1*RGAS);
      jminus= uz-(2.0/(GAMMA-1.0))*sqrt(GAMMA*RGAS*t1);

      /* Get swirl profile data */
      swirlProfile(r[0][j], r[0][neta-1], &omegaz, &utheta);

      /* Now calculate the temperature and axial velocity in row 0 */
      a = 1.0 + (2.0 / (GAMMA - 1.0));
      b = - 2.0 * jminus;
      c = SQ(jminus) - (CHAMBER_TEMP*(4.0*GAMMA*RGAS)
/ SQ(GAMMA- 1.0)) + 2.0 * SQ(utheta) / (GAMMA-1.0);

      /* What sign should go before the sqrt is a little unclear. */
```

```
      uz = (-b + sqrt(b*b - 4.0*a*c))/(2.0*a);
      tg[0][j] = (SQ(GAMMA-1.0)/(4.0*GAMMA*RGAS)) *
(SQ(uz)-2.0*jminus*uz+SQ(jminus));

      /* Calculate the entropy of the flow */
      /* This is going to attempt to be a rather clever trapezoid solver */
      if (j == 0) {
mach = uz / sqrt (GAMMA * RGAS * tg[0][j]);
pg[0][j]= CHAMBER_P/(pow (1.0+((GAMMA-1.0)/2.0) * SQ(mach),
   GAMMA/(GAMMA-1.0)));
rho = pg[0][j] / (RGAS * tg[0][j]);
alpha = pg[0][j] / pow(rho, GAMMA);
s = 0;
sum = 0;
      }
      else {
dsdr = -(Cp*omegaz*utheta) / (htot - 0.5*(SQ(uz)+SQ(utheta)));
s = (r[0][j] / (2.0*(double)(j))) * (2.0 * sum + dsdr);
sum += dsdr;
      }

      /* Get the pressure and density from the entropy */
      rho = pow ( (1.0/alpha)*(htot- 0.5*(SQ(uz)+SQ(utheta)))*
 ((GAMMA-1.0)/GAMMA) / exp(s/Cv), 1.0/(GAMMA-1.0));
      pg[0][j] = RGAS * rho * tg[0][j];

      /* CHECK TO SEE IF MACH NUMBER IS INDEED LESS THAN 1 */
      /* IF THE MACH NUMBER IS GREATER THAN 0.9, SET IT TO 0.9 */
      ainlet = sqrt(GAMMA*RGAS*tg[0][j]);
      if ( (uz/ainlet) > 0.9) {
printf ("Inlet Mach exceeds 1\n");
uz=ainlet * 0.9;
tg[0][j]=CHAMBER_TEMP/(1.0+((GAMMA-1.0)/2.0)*SQ(0.9));
pg[0][j]= CHAMBER_P/(pow (1.0+((GAMMA-1.0)/2.0) * SQ(0.9),
   GAMMA/(GAMMA-1.0)));
      }

      /* Calculate the state vector */
      if (j != 0) {
rho= pg[0][j] / (RGAS*tg[0][j]);
ug[0][j].a = rho;
ug[0][j].b = rho*uz;
ug[0][j].c = 0.0;
ug[0][j].e = rho * utheta;
utot2 =
  (SQ(ug[0][j].b)+SQ(ug[0][j].c)+SQ(ug[0][j].e))/SQ(ug[0][j].a);
ug[0][j].d = (1.0/(GAMMA-1.0))*pg[0][j] + 0.5*ug[0][j].a*utot2;
      }
    }

#endif


#if INLETSUPERSONICC
    /* Corrector Inlet, Supersonic Conditions Only */
    /* This sets supersonic inlet conditions which have nothing to do with */
    /* Anything from the flow */
    for (j = 0; j < neta; ++j) {
      tg[0][j] = CHAMBER_TEMP / (1.0 + ((GAMMA-1.0)/2.0)*SQ(INLETMACH));
      pg[0][j] = CHAMBER_P /
```



```
pow((1.0 + ((GAMMA-1.0)/ 2.0)*SQ(INLETMACH)), GAMMA/(GAMMA-1.0));
      rhogb[j] = pg[0][j] / (RGAS * tg[0][j]);
      ug[0][j].a = rhogb[j];
      ug[0][j].b = ug[0][j].a * INLETMACH *  sqrt(GAMMA*RGAS*tg[0][j]);
      ug[0][j].c = 0.0;
      ug[0][j].d = (1.0/(GAMMA - 1.0)) * pg[0][j] +
0.5 * SQ(ug[0][j].b)/ ug[0][j].a;
      if (((double) (j) / (double)(neta-1)) < 0.2)
ug[0][j].e = rhogb[j]*INLETMACH*sqrt(GAMMA*RGAS*tg[0][j])*
  tan(INLET_TANGENTIAL_VELOCITY_RATIO)* 5.0 *
    ((double)(j)/(double)(neta-1));
      else
ug[0][j].e = rhogb[j]*INLETMACH*sqrt(GAMMA*RGAS*tg[0][j])
  *tan(INLET_TANGENTIAL_VELOCITY_RATIO);
      if (j == 0)
pressure = 0.0;
      else
pressure = pressure + SQ(ug[0][j].e)/(ug[0][j].a * r[0][j]) *
  (r[0][j] - r[0][j-1]);
      pg[0][j] = pg[0][j] + pressure;
    }

#endif


#if CENTERLINEC
    /* Corrector Centerline Boundary */
    /* Calculate enthalpy at inlet */
    for (i = 0; i < nxi; ++i)  {
       /* Set 0 radial and tangential velocity */
      ug[i][0].c = 0.0;
      ug[i][0].e = 0.0;

      /* Get pressure from Solid Body Rotation Rules */
      /* Use a relaxation technique to cushon the blow */
      pg[i][0] = pg[i][1] - 0.5 * ug[i][1].a * SQ(ug[i][1].e/ug[i][1].a);

      /* Assume constant energy */
      ug[i][0].d = ug[i][1].d;

      /* Set constant axial velocity, and calculate density */
      ug[i][0].a = (2.0 / SQ(ug[i][1].b / ug[i][1].a)) *
(ug[i][0].d - (1.0/(GAMMA-1.0)) * pg[i][0]);

      ug[i][0].b = ug[i][0].a * (ug[i][1].b / ug[i][1].a);
      ug[i][0].d = (1.0 / (GAMMA-1.0)) * pg[i][0] +
0.5 * ug[i][0].a * SQ(ug[i][0].b/ug[i][0].a);
      tg[i][0] = pg[i][0] / (ug[i][0].a * RGAS);
    }
#endif

#if UPPERWALLC
    /* Corrector Upper Wall Boundary        */
    /* A simple method designed to insure:  */
    /*    1) Zero Pressure Gradient         */
    /*    2) Tangential Velocity Vector     */
    /* Since keeping the magnitude of the velocity vector doesn't */
    /* Seem to work, this code just takes the normal component and*/
    /* sets it to zero */
```

```
    /* The "x" direction is the axial */
    /* The "y" direction is radial */
    /* The "z" direction is tangential, out */

    for (i = 1; i < nxi-1; ++i) {

      velocity.x = ug[i][neta-2].b / ug[i][neta-2].a;
      velocity.y = 0.0;
      velocity.z = 0.0;

      normal.x = cos(theta[i]);
      normal.y = sin(theta[i]);
      normal.z = 0.0;

      dot = velocity.x * normal.x + velocity.y * normal.y
        + velocity.z * normal.z;

      velocity_magnitude = velocity.x;
      theta2 = acos(dot/velocity_magnitude);
      theta2 = PI/2.0 - theta2;

      velocity.y = tan(theta2) * velocity.x;
      velocity.z = ug[i][neta-2].e / ug[i][neta-2].a;

      /* Construct new State Vector */
      ug[i][neta-1].a = ug[i][neta-2].a;
      ug[i][neta-1].b = ug[i][neta-1].a * velocity.x;
      ug[i][neta-1].c = ug[i][neta-1].a * velocity.y;
      ug[i][neta-1].e = ug[i][neta-1].a * velocity.z;
      new_pressure = pg[i][neta-2] +
(SQ(ug[i][neta-2].e)/(ug[i][neta-2].a * r[i][neta-2]))
 *(r[i][neta-1]-r[i][neta-2]);
      /* Relax Pressure */
      pg[i][neta-1] = (((1.0 - RELAX_FACTOR) * pg[i][neta-1])
                + (RELAX_FACTOR * new_pressure));
      ug[i][neta-1].d = (1.0 / (GAMMA-1.0)) * pg[i][neta-1] +
0.5 * ug[i][neta-1].a * (SQ(velocity.x) + SQ(velocity.y) +
  SQ(velocity.z));
      tg[i][neta-1] = pg[i][neta-1] / (ug[i][neta-1].a * RGAS);
    }

#endif

#if OUTLETC
    /* Corrector Outlet Boundary */
    /* Conditions assume a supersonic outflow and extrapolate flow */
    /* variables values. */
    for (j = 0; j < neta; ++ j) {
      ug[nxi-1][j].a = 2.0 * ug[nxi-2][j].a - ug[nxi-3][j].a;
      ug[nxi-1][j].b = 2.0 * ug[nxi-2][j].b - ug[nxi-3][j].b;
      ug[nxi-1][j].c = 2.0 * ug[nxi-2][j].c - ug[nxi-3][j].c;
      ug[nxi-1][j].e = 2.0 * ug[nxi-2][j].e - ug[nxi-3][j].e;
      pg[nxi-1][j] = 2.0 * pg[nxi-2][j] - pg[nxi-3][j];
      /* To keep very strange things from happening */
      if (pg[nxi-1][j] < 0.0)
pg[nxi-1][j] = 0.1;
      tg[nxi-1][j] = pg[nxi-1][j] / (RGAS * ug[nxi-1][j].a);
      ug[nxi-1][j].d = (1.0 / (GAMMA-1.0)) * pg[nxi-1][j] +
0.5 * ((SQ(ug[nxi-1][j].b) + SQ(ug[nxi-1][j].c) +
       SQ(ug[nxi-1][j].e))/ug[nxi-1][j].a);
    }
#endif

#if OUTLETSUBSONICC
    /* Predictor Outlet boundary for subsonic flow */
    for (j = 0; j < neta; ++j) {
      outlettemperature =  CHAMBER_TEMP / (1.0 + (GAMMA-1.0)/2.0 *
  OUTLETMACH *OUTLETMACH);
      jplus = ug[nxi-2][j].b / ug[nxi-2][j].a + 2.0 *
sqrt(GAMMA * RGAS * tg[nxi-2][j]) / (GAMMA - 1.0);
      jminus = OUTLETMACH * sqrt(GAMMA*RGAS*outlettemperature) - 2.0 *
sqrt(GAMMA*RGAS*outlettemperature) / (GAMMA-1.0);
      s = log(pg[nxi-2][j]) - GAMMA * log(ug[nxi-2][j].a);
      ug[nxi-1][j].a = pow((SQ(jplus - jminus)* (GAMMA-1.0) * (GAMMA-1.0)) /
                    (16.0 * GAMMA * exp(s)),
                            1.0 / (GAMMA-1.0));
      ug[nxi-1][j].b = ug[nxi-1][j].a * 0.5 * (jplus + jminus);
      ug[nxi-1][j].c = ug[nxi-1][j].a * ug[nxi-2][j].c / ug[nxi-2][j].a;
      ug[nxi-1][j].e = ug[nxi-1][j].a * ug[nxi-2][j].e / ug[nxi-2][j].a;
      ug[nxi-1][j].d = ug[nxi-1][j].a * SQ(jplus - jminus) * (GAMMA-1.0) /
(16.0 * GAMMA) + 0.5 * (SQ(ug[nxi-1][j].b)
SQ(ug[nxi-1][j].e)) / ug[nxi-1][j].a;
      pg[nxi-1][j] =  ug[nxi-1][j].a * SQ(jplus - jminus) *
(GAMMA-1.0) * (GAMMA-1.0) / (16.0 * GAMMA);
    }
#endif
  }  /* ends else from if (predictor) */
}    /* Ends boundary() */

swirlProfile(r, rmax, omegaz, utheta)
/* Returns tangential velocity (utheta) and vorticity (omegaz) at inlet */
double *omegaz, r, rmax, *utheta;
{
  double utheta0;

  /* Uncomment for Solid Body Rotation */
/*
  *utheta = INLET_ROTATION_RATE * r;
  *omegaz = 2.0 * INLET_ROTATION_RATE;
*/
  /* Uncomment for Constant Angle Rotation */
  utheta0 = INLET_AXIAL_VELOCITY * tan(TURNING_ANGLE);
  if (r < 0.2 * rmax) {
    *utheta = utheta0 * r / (0.2 * rmax);
    *omegaz = 10.0 * utheta0;
  }
  else {
    *utheta = utheta0;
    *omegaz = utheta0 / r;
  }
}

calculateWallAngle()
{
  double delta_r, delta_z, theta1, theta2;
  int i;

  for (i = 1; i < nxi-1; ++i) {
    /* Calculate wall angle */
    delta_z = z[i][neta-1] - z[i-1][neta-1];
```

```
    delta_r = r[i][neta-1] - r[i-1][neta-1];
    theta1 = atan(delta_r / delta_z) - PI/2.0;
    delta_z = z[i+1][neta-1] - z[i][neta-1];
    delta_r = r[i+1][neta-1] - r[i][neta-1];
    theta2 = atan(delta_r / delta_z) - PI/2.0;
    theta[i] = (theta1 + theta2) / 2.0;
  }
}
```

```
/* damping.c */
/* This module contains any damping/smoothing/artificial vicosity needed */
/* for the terms */
#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

/* These flags will let you set some overdamping on the inlet area */
/* This is done in the hopes that it will kill any pressure waves as */
/* They enter the inlet */
#define OVERDAMPING_LENGTH 70
#define OVERDAMPING_FACTOR 15.0

calculateSecondOrderDamping()
/* It is not set up to use pressure switches yet */
/* It's not even set up yet, 10/30/92 */
{

  int i, j;
  if (dxi2 < EPSILON && deta2 < EPSILON)
    ;
  else {
    for (j = 0; j < neta; ++j) {
      for (i = 0; i < nxi; ++i) {
if (j < 1 || i < 1 || j > neta-2 || i > nxi-2) {
  d2[i][j].a = 0.0;
  d2[i][j].b = 0.0;
  d2[i][j].c = 0.0;
  d2[i][j].d = 0.0;
  d2[i][j].e = 0.0;
}
else {
  d2[i][j].a = dxi2 * (ug[i+1][j].a - 2.0 * ug[i][j].a + ug[i-1][j].a)
    + deta2 * (ug[i][j+1].a - 2.0 * ug[i][j].a + ug[i][j-1].a);
  d2[i][j].b = dxi2 * (ug[i+1][j].b - 2.0 * ug[i][j].b + ug[i-1][j].b)
    + deta2 * (ug[i][j+1].b - 2.0 * ug[i][j].b + ug[i][j-1].b);
  d2[i][j].c = dxi2 * (ug[i+1][j].c - 2.0 * ug[i][j].c + ug[i-1][j].c)
    + deta2 * (ug[i][j+1].c - 2.0 * ug[i][j].c + ug[i][j-1].c);
  d2[i][j].d = dxi2 * (ug[i+1][j].d - 2.0 * ug[i][j].d + ug[i-1][j].d)
    + deta2 * (ug[i][j+1].d - 2.0 * ug[i][j].d + ug[i][j-1].d);
  d2[i][j].e = dxi2 * (ug[i+1][j].e - 2.0 * ug[i][j].e + ug[i-1][j].e)
    + deta2 * (ug[i][j+1].e - 2.0 * ug[i][j].e + ug[i][j-1].e);
}
      }
    }
  }
}

calculateFourthOrderDamping()
{

  int i,j;

  for (j = 0; j < neta; ++j)  {
    for (i = 0; i < nxi; ++i)  {
      if (j < 2 || i < 2 || j > neta-3 || i > nxi-3) {
d[i][j].a = 0.0;
d[i][j].b = 0.0;
d[i][j].c = 0.0;
d[i][j].d = 0.0;
d[i][j].e = 0.0;
      }
      else if (i < OVERDAMPING_LENGTH) {
d[i][j].a = -deta4* OVERDAMPING_FACTOR * (ug[i+2][j].a + ug[i-2][j].a -
   4.0 * (ug[i+1][j].a + ug[i-1][j].a) + 6.0 * ug[i][j].a)
            - dxi4 * OVERDAMPING_FACTOR * (ug[i][j+2].a + ug[i][j-2].a -
   4.0 * (ug[i][j+1].a + ug[i][j-1].a) + 6.0 * ug[i][j].a);
d[i][j].b = -deta4* OVERDAMPING_FACTOR * (ug[i+2][j].b + ug[i-2][j].b -
   4.0 * (ug[i+1][j].b + ug[i-1][j].b) + 6.0 * ug[i][j].b)
            - dxi4 * OVERDAMPING_FACTOR * (ug[i][j+2].b + ug[i][j-2].b -
   4.0 * (ug[i][j+1].b + ug[i][j-1].b) + 6.0 * ug[i][j].b);
d[i][j].c = -deta4* OVERDAMPING_FACTOR * (ug[i+2][j].c + ug[i-2][j].c -
   4.0 * (ug[i+1][j].c + ug[i-1][j].c) + 6.0 * ug[i][j].c)
            - dxi4 * OVERDAMPING_FACTOR * (ug[i][j+2].c + ug[i][j-2].c -
   4.0 * (ug[i][j+1].c + ug[i][j-1].c) + 6.0 * ug[i][j].c);
d[i][j].d = -deta4* OVERDAMPING_FACTOR * (ug[i+2][j].d + ug[i-2][j].d -
   4.0 * (ug[i+1][j].d + ug[i-1][j].d) + 6.0 * ug[i][j].d)
           -dxi4 * OVERDAMPING_FACTOR * (ug[i][j+2].d + ug[i][j-2].d -
   4.0 * (ug[i][j+1].d + ug[i][j-1].d) + 6.0 * ug[i][j].d);
d[i][j].e = -deta4* OVERDAMPING_FACTOR * (ug[i+2][j].e + ug[i-2][j].e -
   4.0 * (ug[i+1][j].e + ug[i-1][j].e) + 6.0 * ug[i][j].e)
            - dxi4 * OVERDAMPING_FACTOR * (ug[i][j+2].e + ug[i][j-2].e -
   4.0 * (ug[i][j+1].e + ug[i][j-1].e) + 6.0 * ug[i][j].e);
      }
      else  {
d[i][j].a = -deta4 * (ug[i+2][j].a + ug[i-2][j].a -
   4.0 * (ug[i+1][j].a + ug[i-1][j].a) + 6.0 * ug[i][j].a)
            - dxi4 * (ug[i][j+2].a + ug[i][j-2].a -
   4.0 * (ug[i][j+1].a + ug[i][j-1].a) + 6.0 * ug[i][j].a);
d[i][j].b = -deta4 * (ug[i+2][j].b + ug[i-2][j].b -
   4.0 * (ug[i+1][j].b + ug[i-1][j].b) + 6.0 * ug[i][j].b)
            - dxi4 * (ug[i][j+2].b + ug[i][j-2].b -
   4.0 * (ug[i][j+1].b + ug[i][j-1].b) + 6.0 * ug[i][j].b);
d[i][j].c = -deta4 * (ug[i+2][j].c + ug[i-2][j].c -
   4.0 * (ug[i+1][j].c + ug[i-1][j].c) + 6.0 * ug[i][j].c)
            - dxi4 * (ug[i][j+2].c + ug[i][j-2].c -
   4.0 * (ug[i][j+1].c + ug[i][j-1].c) + 6.0 * ug[i][j].c);
d[i][j].d = -deta4 * (ug[i+2][j].d + ug[i-2][j].d -
   4.0 * (ug[i+1][j].d + ug[i-1][j].d) + 6.0 * ug[i][j].d)
            - dxi4 * (ug[i][j+2].d + ug[i][j-2].d -
   4.0 * (ug[i][j+1].d + ug[i][j-1].d) + 6.0 * ug[i][j].d);
d[i][j].e = -deta4 * (ug[i+2][j].e + ug[i-2][j].e -
   4.0 * (ug[i+1][j].e + ug[i-1][j].e) + 6.0 * ug[i][j].e)
            - dxi4 * (ug[i][j+2].e + ug[i][j-2].e -
   4.0 * (ug[i][j+1].e + ug[i][j-1].e) + 6.0 * ug[i][j].e);
      }

    }
  }
}
```

```
/* init.c */
/* Contains routine for setting initial run conditions */
/* Works 5/13/92 */

#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

/* If this flag is true, the initial conditions will turn the flow */
#define TURNFLOW 1

/* This flag determines whether the "turn flow" algorithm is turned on */
/* If it is, then the flow next to the wall is turned to be parallel to */
/* The wall. */

initializeflow()
/* Presently, this is written to give the 1-d solution to the flow equations */
/* The flow is then modified to fit the tangent to wall boundary condition   */
/* And the rest of the flow is smoothed from the wall to the centerline so   */
/* As to prevent shocks */
/* This solution does NOT initially fulfil the continuity equations */
{
  char subsonic;
  double afraction,
         area,
         centrifical_pressure,
         delta_r, delta_z,
         diff,
         dot,
         ftheta0,
         mach,
         magnitude_vtan,
         magnitude_v2,
         pressure,
         rho,
         scrap,
         swirl_velocity,
         theta1, theta2, theta,
         utotal2;
  int i, j;
  vector3 normal, tangential_velocity, velocity;

  /* Omit initializing mass fractions */
  predictor = TRUE;

  /* Set run time  counter */
  time = 0.0;

  /* Initialize inlet densities and velocities */
  for (j = 0; j < neta; ++j) {
    for (i = 0; i < nxi; ++i) {
      ug[0][j].a = INLET_DENSITY;
      ug[0][j].b = ug[0][j].a * INLET_Z_VELOCITY;
      ug[0][j].c = 0.0;
    }
  }

  /* Set 1-D Nozzle conditions down the nozzle */
  for (i = 0; i < nxi; ++i) {
    area = PI * SQ(r[i][neta-1]);

    /* Iterative Solver */
    if (z[i][neta-1] < THROAT_LOCATION)
      for (mach =0.05, diff=1.0; diff > 0.0; mach += 0.02) {
afraction=(1.0/mach)*pow(((1.0+0.2*SQ(mach))/(1.2)),3.0);
diff = afraction - area/THROAT_AREA;
      }
    else
      for (mach = 1.0, diff= -1.0; diff < 0.0; mach+=0.02) {
afraction = (1.0/mach)*pow(((1.0+0.2*SQ(mach))/(1.2)),3.0);
diff = afraction - area/THROAT_AREA;
      }
    /* End of Iterative Solver */

    for (j = 0; j < neta; ++j) {
      tg[i][j] = CHAMBER_TEMP/(1.0 + 0.2*SQ(mach));
      pressure = CHAMBER_P / pow((1.0 + 0.2 * SQ(mach)),3.5);
      ug[i][j].a = pressure / (RGAS * tg[i][j]);
      ug[i][j].b = ug[i][j].a * mach * sqrt(GAMMA*RGAS*tg[i][j]);
      ug[i][j].c = 0.0;
      ug[i][j].e = 0.0;
      utotal2=(SQ(ug[i][j].b)+SQ(ug[i][j].c)+SQ(ug[i][j].e))/SQ(ug[i][j].a);
      ug[i][j].d = (1.0/(GAMMA-1.0))*pg[i][j]+0.5*ug[i][j].a*utotal2;
      pg[i][j] = pressure;
    }
  }


  for (i = 0; i < nxi; ++i)
    for (j = 0; j < neta; ++j) {
      if (j == 0)
centrifical_pressure = 0.0;
      else
centrifical_pressure = centrifical_pressure
  + SQ(ug[i][j].e)/(ug[i][j].a * r[i][j])
    * (r[i][j] - r[i][j-1]);

      /* This routine in boundary.c */
      swirlProfile(r[i][j], r[i][neta-1], &scrap, &swirl_velocity);

      /* OK, calculate the pressure */
      /* This pressure based on a JxB swirl force model */
      /* Where the swirl force ADDS to the total pressure */
      centrifical_pressure =
exp(SQ(INLET_ROTATION_RATE) * SQ(r[i][j]) / (2.0 * RGAS * tg[i][j]));
      /* But, to keep things from getting to out of control... */
      if (centrifical_pressure > 10.0)
centrifical_pressure = 10.0;
      pg[i][j] = pg[i][j] * centrifical_pressure;
      rho = pg[i][j] / (RGAS * tg[i][j]);
      ug[i][j].b = ug[i][j].b / ug[i][j].a * rho;
      ug[i][j].c = ug[i][j].c / ug[i][j].a * rho;
      ug[i][j].e = swirl_velocity * rho;
      ug[i][j].a = rho;
      utotal2=(SQ(ug[i][j].b)+SQ(ug[i][j].c)+SQ(ug[i][j].e))/SQ(ug[i][j].a);
      ug[i][j].d = (1.0/(GAMMA-1.0))*pg[i][j]+0.5*ug[i][j].a*utotal2;
    }
```

```
#if TURNFLOW

  /* Ok; Turn the flow vector next to the wall so it is parallel to the */
  /* wall and bend all the other vectors in the flow to match, with a linear */
  /* tapering of the flow back to parallel at the centerline */
  /* For the moment, it will turn the flow vector and maintain its magnitude */
  for (i = 1; i < nxi; ++i) {

    /* Calculate wall angle */
    delta_z = z[i][neta-1] - z[i-1][neta-1];
    delta_r = r[i][neta-1] - r[i-1][neta-1];
    theta1 = atan(delta_r / delta_z) - PI/2.0;
    if (i != (nxi - 1)) {
      delta_z = z[i+1][neta-1] - z[i][neta-1];
      delta_r = r[i+1][neta-1] - r[i][neta-1];
      theta2 = atan(delta_r / delta_z) - PI/2.0;
      theta = (theta1 + theta2) / 2.0;
    }
    else
      theta = theta1;

    for (j = 0; j < neta; ++j) {
      velocity.x = ug[i][j].b / ug[i][j].a;
      velocity.y = ug[i][j].c / ug[i][j].a;
      velocity.z = ug[i][j].e / ug[i][j].a;

      theta1 = -(PI/2.0 - ((PI/2.0 + theta) * (double)(j) / (double)(neta-1)));

      normal.x = cos(theta1);
      normal.y = sin(theta1);
      normal.z = 0.0;

      dot = velocity.x * normal.x + velocity.y * normal.y
+ velocity.z * normal.z;

      normal.x = normal.x * dot;
      normal.y = normal.y * dot;
      normal.z = normal.z * dot;

      tangential_velocity.x = velocity.x - normal.x;
      tangential_velocity.y = velocity.y - normal.y;
      tangential_velocity.z = velocity.z - normal.z;

      magnitude_v2 = sqrt(SQ(velocity.x) + SQ(velocity.y) + SQ(velocity.z));

      magnitude_vtan = sqrt(SQ(tangential_velocity.x) +
    SQ(tangential_velocity.y) +
    SQ(tangential_velocity.z));

      velocity.x = magnitude_v2 * tangential_velocity.x / magnitude_vtan;
      velocity.y = magnitude_v2 * tangential_velocity.y / magnitude_vtan;
      velocity.z = magnitude_v2 * tangential_velocity.z / magnitude_vtan;

      /* Construct new State Vector */
      /* Leave the energy and density terms alone */
      ug[i][j].b = ug[i][j].a * velocity.x;
      ug[i][j].c = ug[i][j].a * velocity.y;
      ug[i][j].e = ug[i][j].a * velocity.z;
    }
  }
#endif
}
```

```
/* integrate.c */
/* Being Written 5/15/92 */

#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

integrate()
{
  double deltatime;
  int i,j;

  /* Copy current pressures into file for error calculation later and
  /* Copy the odl state vectors into the "predictor" matrix */
  for (i = 0; i < nxi; ++i) {
    for (j = 0; j < neta; ++j) {
      /* Copy current pressure to error file here */
      pg0[i][j] = pg[i][j];

      /* Copy state vectors here */
      ugp[i][j].a = ug[i][j].a;
      ugp[i][j].b = ug[i][j].b;
      ugp[i][j].c = ug[i][j].c;
      ugp[i][j].d = ug[i][j].d;
      ugp[i][j].e = ug[i][j].e;
      pgp[i][j] = pg[i][j];
    }
  }

  /* Set Predictor Flag */
  predictor = TRUE;

  /* This routine in time.c */
  calculatetimestep(&deltatime);
  time += deltatime;
  printf("Timestep: %g Run time: %g", deltatime, time);

  /* This routine is in this file */
  calculateSourceTerm();

  /* These routines are also in this file */
  calculateFluxtermF();
  calculateFluxtermG();

  /* No damping is applied in the predictor step */
  /* Calculate State Vector */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      ugp[i][j].a = (ug[i][j].a * jacobian[i][j] - deltatime *
     (fg[i][j].a+gg[i][j].a - sg[i][j].a))
             / jacobian[i][j];
      ugp[i][j].b = (ug[i][j].b * jacobian[i][j] - deltatime *
     (fg[i][j].b + gg[i][j].b - sg[i][j].b))
             / jacobian[i][j];
      ugp[i][j].c = (ug[i][j].c * jacobian[i][j] - deltatime *
     (fg[i][j].c + gg[i][j].c - sg[i][j].c))
             / jacobian[i][j];
      ugp[i][j].d = (ug[i][j].d * jacobian[i][j] - deltatime *
     (fg[i][j].d + gg[i][j].d - sg[i][j].d))
             / jacobian[i][j];
      ugp[i][j].e = (ug[i][j].e * jacobian[i][j] - deltatime *
     (fg[i][j].e + gg[i][j].e - sg[i][j].e))
             / jacobian[i][j];
    }
  }

  /* This routine in this file */
  calculateTemperature();

  /* Apply boundary conditions */
  /* This routine located in boundary.c */
  boundary();

  /* do the corrector step */
  predictor = FALSE;

  /* Calculate Corrector Flux terms and source terms */
  calculateFluxtermF();
  calculateFluxtermG();

  /* This Routine in damping.c */
  calculateSecondOrderDamping();

  /* This routine also in damping.c */
  calculateFourthOrderDamping();

  /* The will 2nd order damp at the inlet only */
/*  inletDamping();*/

  /* Calculate State Vector, Corrector Step */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      ug[i][j].a = (0.5 * ((ug[i][j].a + ugp[i][j].a) * jacobian[i][j] -
       (deltatime * (fg[i][j].a + gg[i][j].a - sg[i][j].a)))
    + 0.5 * (d2[i][j].a + d[i][j].a)) / jacobian[i][j];
      ug[i][j].b = (0.5 * ((ug[i][j].b + ugp[i][j].b) * jacobian[i][j] -
       (deltatime * (fg[i][j].b + gg[i][j].b - sg[i][j].b)))
        + 0.5 * (d2[i][j].b + d[i][j].b)) / jacobian[i][j];
      ug[i][j].c = (0.5 * ((ug[i][j].c + ugp[i][j].c) * jacobian[i][j] -
       (deltatime * (fg[i][j].c + gg[i][j].c - sg[i][j].c)))
        + 0.5 * (d2[i][j].c +  d[i][j].c)) / jacobian[i][j];
      ug[i][j].d = (0.5 * ((ug[i][j].d + ugp[i][j].d) * jacobian[i][j] -
       (deltatime * (fg[i][j].d + gg[i][j].d - sg[i][j].d)))
        + 0.5 * (d2[i][j].d +  d[i][j].d)) / jacobian[i][j];
      ug[i][j].e = (0.5 * ((ug[i][j].e + ugp[i][j].e) * jacobian[i][j] -
       (deltatime * (fg[i][j].e + gg[i][j].e - sg[i][j].e)))
        + 0.5 * (d2[i][j].e + d[i][j].e)) / jacobian[i][j];
    }
  }

  calculateTemperature();

  boundary();

  predictor = TRUE;
  time = time + deltatime;

}
```

```
calculateFluxtermF()
/* depends heavily on the global flag PREDICTOR */
{
  int i, j;

  /* Calculate the F[i,j] term (predictor) or f[i-1,j] term (corrector) */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      if (predictor) {
f1[i][j].a = -ug[i][j].c * dzdeta[i][j] + ug[i][j].b*drdeta[i][j];
f1[i][j].b = (pg[i][j]+(SQ(ug[i][j].b)/ug[i][j].a))*drdeta[i][j]
  -((ug[i][j].b*ug[i][j].c)/ug[i][j].a) * dzdeta[i][j];
f1[i][j].c = -(pg[i][j] + SQ(ug[i][j].c)/ug[i][j].a) * dzdeta[i][j] +
   ((ug[i][j].b*ug[i][j].c)/ ug[i][j].a) * drdeta[i][j];
f1[i][j].d = (ug[i][j].d / ug[i][j].a + RGAS *tg[i][j]) *
  (ug[i][j].b * drdeta[i][j] - ug[i][j].c * dzdeta[i][j]);
f1[i][j].e = -(ug[i][j].c * ug[i][j].e/ug[i][j].a)*dzdeta[i][j] +
  (ug[i][j].b * ug[i][j].e / ug[i][j].a) * drdeta[i][j];
      }
      else {
f1[i][j].a=
    -ugp[i-1][j].c * dzdeta[i-1][j] + ugp[i-1][j].b*drdeta[i-1][j];
f1[i][j].b=
   (pgp[i-1][j]+(SQ(ugp[i-1][j].b)/ugp[i-1][j].a))*drdeta[i-1][j]
  -((ugp[i-1][j].b*ugp[i-1][j].c)/ugp[i-1][j].a) * dzdeta[i-1][j];
f1[i][j].c=
  -(pgp[i-1][j] + SQ(ugp[i-1][j].c)/ugp[i-1][j].a) * dzdeta[i-1][j] +
   ((ugp[i-1][j].b*ugp[i-1][j].c)/ ugp[i-1][j].a) * drdeta[i-1][j];
f1[i][j].d=(ugp[i-1][j].d / ugp[i-1][j].a + RGAS *tg[i-1][j]) *
  (ugp[i-1][j].b * drdeta[i-1][j] - ugp[i-1][j].c * dzdeta[i-1][j]);
f1[i][j].e=
  -(ugp[i-1][j].c * ugp[i-1][j].e/ugp[i-1][j].a)*dzdeta[i-1][j] +
  (ugp[i-1][j].b * ugp[i-1][j].e / ugp[i-1][j].a) * drdeta[i-1][j];
      } /* Ends Else */
    }
  }

  /* Calculate the F[i+1][j] term (predictor) or F[i][j] term (corrector) */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      if (predictor) {

f2[i][j].a =
  -ug[i+1][j].c * dzdeta[i+1][j] + ug[i+1][j].b*drdeta[i+1][j];
f2[i][j].b =
  (pg[i+1][j]+(SQ(ug[i+1][j].b)/ug[i+1][j].a))*drdeta[i+1][j]
  -((ug[i+1][j].b*ug[i+1][j].c)/ug[i+1][j].a) * dzdeta[i+1][j];
f2[i][j].c =
  -(pg[i+1][j] + SQ(ug[i+1][j].c)/ug[i+1][j].a) * dzdeta[i+1][j] +
  ((ug[i+1][j].b*ug[i+1][j].c)/ ug[i+1][j].a) * drdeta[i+1][j];
f2[i][j].d =
  (ug[i+1][j].d / ug[i+1][j].a + RGAS *tg[i+1][j]) *
  (ug[i+1][j].b * drdeta[i+1][j] - ug[i+1][j].c * dzdeta[i+1][j]);
f2[i][j].e =
  -(ug[i+1][j].c * ug[i+1][j].e/ug[i+1][j].a)*dzdeta[i+1][j] +
  (ug[i+1][j].b * ug[i+1][j].e / ug[i+1][j].a) * drdeta[i+1][j];
      } /* Ends if */
      else {
f2[i][j].a = -ugp[i][j].c * dzdeta[i][j] + ugp[i][j].b*drdeta[i][j];
f2[i][j].b = (pgp[i][j]+(SQ(ugp[i][j].b)/ugp[i][j].a))*drdeta[i][j]
```



```
  -((ugp[i][j].b*ugp[i][j].c)/ugp[i][j].a) * dzdeta[i][j];
f2[i][j].c = -(pgp[i][j] + SQ(ugp[i][j].c)/ugp[i][j].a) *dzdeta[i][j] +
   ((ugp[i][j].b*ugp[i][j].c)/ ugp[i][j].a) * drdeta[i][j];
f2[i][j].d = (ugp[i][j].d / ugp[i][j].a + RGAS *tg[i][j]) *
  (ugp[i][j].b * drdeta[i][j] - ugp[i][j].c * dzdeta[i][j]);
f2[i][j].e = -(ugp[i][j].c * ugp[i][j].e/ugp[i][j].a)*dzdeta[i][j] +
  (ugp[i][j].b * ugp[i][j].e / ugp[i][j].a) * drdeta[i][j];
      }   /* Ends else loop */
    }
  } /* Ends for loops */

  /* Calculate Net Flux Vector */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      fg[i][j].a = f2[i][j].a - f1[i][j].a;
      fg[i][j].b = f2[i][j].b - f1[i][j].b;
      fg[i][j].c = f2[i][j].c - f1[i][j].c;
      fg[i][j].d = f2[i][j].d - f1[i][j].d;
      fg[i][j].e = f2[i][j].e - f1[i][j].e;
    }
  }
}

calculateFluxtermG()
{
  int i, j;

  /* g[i][j] term (predictor) or g[i][j-1] term (corrector) */
  if (predictor) {
    for (j = 1; j < neta-1; ++j) {
      for (i = 1; i < nxi-1; ++i) {
g1[i][j].a = ug[i][j].c * dzdxi[i][j]-ug[i][j].b * drdxi[i][j];
g1[i][j].b = -(pg[i][j] + (SQ(ug[i][j].b)/ug[i][j].a))*drdxi[i][j] +
  (ug[i][j].b*ug[i][j].c/ug[i][j].a) * dzdxi[i][j];
g1[i][j].c = (pg[i][j] + (SQ(ug[i][j].c) / ug[i][j].a)) * dzdxi[i][j] -
  (ug[i][j].b * ug[i][j].c / ug[i][j].a) * drdxi[i][j];
g1[i][j].d = (ug[i][j].d/ug[i][j].a + RGAS* tg[i][j]) *
  (-ug[i][j].b * drdxi[i][j] + ug[i][j].c * dzdxi[i][j]);
g1[i][j].e = (ug[i][j].c * ug[i][j].e / ug[i][j].a) * dzdxi[i][j] -
  (ug[i][j].b * ug[i][j].e / ug[i][j].a) * drdxi[i][j];
      }
    }
  }
  else {
    for (j = 1; j < neta-1; ++j) {
      for ( i = 1; i < nxi-1; ++i) {
g1[i][j].a =
  ugp[i][j-1].c * dzdxi[i][j-1]-ugp[i][j-1].b * drdxi[i][j-1];
g1[i][j].b =
  -(pgp[i][j-1] + (SQ(ugp[i][j-1].b)/ugp[i][j-1].a))*drdxi[i][j-1] +
  (ugp[i][j-1].b*ugp[i][j-1].c/ugp[i][j-1].a) * dzdxi[i][j-1];
g1[i][j].c =
  (pgp[i][j-1] + (SQ(ugp[i][j-1].c) / ugp[i][j-1].a)) * dzdxi[i][j-1] -
  (ugp[i][j-1].b * ugp[i][j-1].c / ugp[i][j-1].a) * drdxi[i][j-1];
g1[i][j].d =
  (ugp[i][j-1].d/ugp[i][j-1].a + RGAS* tg[i][j-1]) *
  (-ugp[i][j-1].b * drdxi[i][j-1] + ugp[i][j-1].c * dzdxi[i][j-1]);
g1[i][j].e =
  (ugp[i][j-1].c * ugp[i][j-1].e / ugp[i][j-1].a) * dzdxi[i][j-1] -
  (ugp[i][j-1].b * ugp[i][j-1].e / ugp[i][j-1].a) * drdxi[i][j-1];
```

```
      }
    }
  }


  /* Calculate G[i][j+1] term (predictor) or G[i][j] term (corrector) */
  if (predictor) {
    for (j = 1; j < neta-1; ++j) {
      for (i = 1; i < nxi-1; ++i) {
g2[i][j].a =
  ug[i][j+1].c * dzdxi[i][j+1]-ug[i][j+1].b * drdxi[i][j+1];
g2[i][j].b =
  -(pg[i][j+1] + (SQ(ug[i][j+1].b)/ug[i][j+1].a))*drdxi[i][j+1] +
  (ug[i][j+1].b*ug[i][j+1].c/ug[i][j+1].a) * dzdxi[i][j+1];
g2[i][j].c =
  (pg[i][j+1] + (SQ(ug[i][j+1].c) / ug[i][j+1].a)) * dzdxi[i][j+1] -
  (ug[i][j+1].b * ug[i][j+1].c / ug[i][j+1].a) * drdxi[i][j+1];
g2[i][j].d =
  (ug[i][j+1].d/ug[i][j+1].a + RGAS * tg[i][j+1]) *
  (-ug[i][j+1].b * drdxi[i][j+1] + ug[i][j+1].c * dzdxi[i][j+1]);
g2[i][j].e =
  (ug[i][j+1].c * ug[i][j+1].e / ug[i][j+1].a) * dzdxi[i][j+1] -
  (ug[i][j+1].b * ug[i][j+1].e / ug[i][j+1].a) * drdxi[i][j+1];
      }
    }
    /* Debugging stuff */
    j = j;
    /* End debugging stuff */
  }
  else {
    for (j = 1; j < neta-1; ++j) {
      for ( i = 1; i < nxi-1; ++i) {
g2[i][j].a = ugp[i][j].c * dzdxi[i][j]-ugp[i][j].b * drdxi[i][j];
g2[i][j].b = -(pgp[i][j] + (SQ(ugp[i][j].b)/ugp[i][j].a))*drdxi[i][j] +
  (ugp[i][j].b*ugp[i][j].c/ugp[i][j].a) * dzdxi[i][j];
g2[i][j].c =
  (pgp[i][j] + (SQ(ugp[i][j].c) / ugp[i][j].a)) * dzdxi[i][j] -
  (ugp[i][j].b * ugp[i][j].c / ugp[i][j].a) * drdxi[i][j];
g2[i][j].d = (ugp[i][j].d/ugp[i][j].a + RGAS* tg[i][j]) *
  (-ugp[i][j].b * drdxi[i][j] + ugp[i][j].c * dzdxi[i][j]);
g2[i][j].e = (ugp[i][j].c * ugp[i][j].e / ugp[i][j].a) * dzdxi[i][j] -
  (ugp[i][j].b * ugp[i][j].e / ugp[i][j].a) * drdxi[i][j];
      }
    }
  }

  /* Calculate Net Flux Vector */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      gg[i][j].a = g2[i][j].a - g1[i][j].a;
      gg[i][j].b = g2[i][j].b - g1[i][j].b;
      gg[i][j].c = g2[i][j].c - g1[i][j].c;
      gg[i][j].d = g2[i][j].d - g1[i][j].d;
      gg[i][j].e = g2[i][j].e - g1[i][j].e;
    }
  }
}

calculateSourceTerm()
{
```



```
  int i, j;

  /* The source terms in row 0 are NOT calculated so the programs doesn't */
  /* Crash when r = 0 */
  /* Turn off for debugging */

  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      sg[i][j].a = -ug[i][j].c * jacobian[i][j] / r[i][j];
      sg[i][j].b=
-(ug[i][j].c * (ug[i][j].b/ug[i][j].a))*jacobian[i][j] / r[i][j];
      sg[i][j].c =
     -((SQ(ug[i][j].c) - SQ(ug[i][j].e))/ ug[i][j].a) * jacobian[i][j]/r[i][j];
      sg[i][j].d = -((ug[i][j].d+ug[i][j].a*RGAS*tg[i][j])*
     (ug[i][j].c/ug[i][j].a))*jacobian[i][j] / r[i][j];
      sg[i][j].e = - 2.0* ((ug[i][j].c*ug[i][j].e)/ug[i][j].a) *
jacobian[i][j]/r[i][j];

#if NOSOURCE
      /* DEBUG STUFF */
      sg[i][j].a = 0.0;
      sg[i][j].b = 0.0;
      sg[i][j].c = 0.0;
      sg[i][j].d = 0.0;
      sg[i][j].e = 0.0;
      /* end debug stuff */
#endif
    }
  }
}

calculateTemperature()
/* Well, in Scott's code, this is something fancy.  I'm just going to read */
/* Off the temperature and not bother to change anything else... */
{
  double pressure;
  int i,j;

  if (predictor) {
    for (i = 1; i < nxi-1; ++i) {
      for (j = 1; j < neta-1; ++j) {
pgp[i][j] = (GAMMA-1.0) * (ugp[i][j].d -
        0.5*(SQ(ugp[i][j].b)+SQ(ugp[i][j].c)+SQ(ugp[i][j].e))/ugp[i][j].a);
tg[i][j] = pgp[i][j] / (ugp[i][j].a * RGAS);
      }
    }
  }
  else {
    for (i = 1; i < nxi-1; ++i) {
      for (j = 1; j < neta-1; ++j) {
pg[i][j] = (GAMMA-1.0) * (ug[i][j].d -
  0.5*(SQ(ug[i][j].b)+SQ(ug[i][j].c)+SQ(ug[i][j].e))/ug[i][j].a);
tg[i][j] = pg[i][j] / (ug[i][j].a * RGAS);
      }
    }
  }
}
```

```
/* The io routines for nozzle */
/* Contains the routines getflow(fname) and writefile(fname) and */
/* getgrid(gridname) */
/* All routines tested and work to SINGLE PRECISION 5/27/92 */
/* The text files simply do NOT save to double precision */

#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

#define READASCII 0

getflow(fname)
     char fname[];
{
  char binarydatastring[40], statstring[40], ugstring[40];
  double scrap[20];
  FILE *binarydatafile, *statfile, *ugfile;
  int doublesize, i, j, vector5size;

  sprintf(binarydatastring, "%s.dat", fname);
  binarydatafile = fopen(binarydatastring, "r");
  doublesize = sizeof(double);
  vector5size = sizeof(vector5);

  fread(&nxi, sizeof(int), 1, binarydatafile);
  fread(&neta, sizeof(int),1,binarydatafile);
  fread(&time, sizeof(double),1,binarydatafile);
  fread(&ftheta, sizeof(double),1,binarydatafile);
  fread(&iter, sizeof(int), 1, binarydatafile);
  fread(scrap, sizeof(double), 8, binarydatafile);
  for (i=0; i < nxi; ++i) {
    fread (ug[i], vector5size, neta, binarydatafile);
    fread (pg[i], doublesize, neta, binarydatafile);
  }
  fclose(binarydatafile);

#if READASCII

  /* Construct full file names */
  sprintf(statstring, "%s.stat", fname);
  sprintf(ugstring, "%s.ug", fname);

  statfile =  fopen(statstring, "r");
  ugfile = fopen (ugstring, "r");

  if (statfile == NULL || ugfile == NULL) {
    printf("\n\nUnable to open data files.");
    exit();
  }

  fscanf(statfile, "%d %d", &nxi, &neta);   /* reads number of grid cells */
  fscanf(statfile, "%d", &iter);
  fscanf(statfile, "%lf", &time);
  fscanf(statfile, "%lf", &twall);  /* This line is not needed */
  fscanf(statfile, "%lf %lf", &mdot, &ftheta);

  /* Read Inlet conditions */
  /* Reads file stat */
  for (j = 0; j < neta; ++j)
    fscanf(statfile, "%lf %lf", &hgin[j], &ugin[j]);

  /* Read State Vectors */
  /* Uses file ug */
  for (j = 0; j < neta; ++j){
    for (i = 0; i < nxi; ++i) {
      fscanf(ugfile, "%lf %lf %lf %lf %lf", &(ug[i][j].a), &(ug[i][j].b),
     &(ug[i][j].c), &(ug[i][j].d), &(ug[i][j].e));
      fscanf(ugfile, "%lf", &(pg[i][j]));
    }
  }

  fclose (statfile);
  fclose (ugfile);

#endif

  /* Calculate gas temperature */
  for (j = 0; j < neta; ++j) {
    for (i = 0; i < nxi; ++i) {
      tg[i][j]=pg[i][j]/(RGAS * ug[i][j].a);
    }
  }
}

getgrid(gridname)
/* This routine loads the grid and calculates the grid metrics */
/* For the moment, it only loads; 5/12/92 */
     char gridname[];
{
  char gridstring[40];
  FILE *gridfile;
  int i, j;

  sprintf(gridstring, "grid.%s", gridname);
  gridfile = fopen (gridstring, "r");
  if (gridfile == NULL) {
    printf ("\nUnable to open grid file...");
    exit();
  }
  fscanf(gridfile, "%d %d", &nxi, &neta);

  /* This routine in memory.c */
  allocatememory();

  /* Read grid file */
  for (j = 0; j < neta; ++j)
    for (i = 0; i < nxi; ++i)
      fscanf(gridfile, "%lf %lf", &z[i][j], &r[i][j]);
}

writeflow(fname)
char fname[];
{
  char binarydatastring[40], statstring[40], ugstring[40];
  double normalizedgrms, scrap[10];
  FILE *binarydatafile, *errorfile, *statfile, *ugfile;
  int doublesize, i, j, vector5size;
```

```
printf("\nWriting Output Files.");

/* First, write the ASCII files */
/* Construct full file names */
sprintf(statstring, "%s.stat", fname);
sprintf(ugstring, "%s.ug", fname);

statfile =  fopen(statstring, "w");
ugfile = fopen (ugstring, "w");

if (statfile == NULL || ugfile == NULL) {
  printf("\n\nUnable to write data files.");
  exit();
}

fprintf(statfile, "%d\n %d\n", nxi, neta);   /* reads number of grid cells */
fprintf(statfile, "%d\n", iter);
fprintf(statfile, "%f\n", time);
fprintf(statfile, "%f\n", twall);  /* This line is not needed */
fprintf(statfile, "%f\n %f\n", mdot, ftheta);

/* Write Inlet conditions */
for (j = 0; j < neta; ++j)
  fprintf (statfile, "%f\n %f\n", hgin[j], ugin[j]);

/* Write State Vector */
for (j = 0; j < neta; ++j) {
  for (i = 0; i < nxi; ++i) {
    fprintf (ugfile, "%f\n %f\n %f\n %f\n %f\n", ug[i][j].a, ug[i][j].b,
     ug[i][j].c,  ug[i][j].d,  ug[i][j].e);
    fprintf (ugfile, "%f\n", pg[i][j]);
  }
}

/* Close files */
fclose (statfile);
fclose (ugfile);

/* Now write the binary files */
sprintf(binarydatastring, "%s.dat", fname);
binarydatafile = fopen(binarydatastring, "w");
doublesize = sizeof(double);
vector5size = sizeof(vector5);

fwrite(&nxi, sizeof(int), 1, binarydatafile);
fwrite(&neta, sizeof(int),1,binarydatafile);
fwrite(&time, sizeof(double),1,binarydatafile);
fwrite(&ftheta, sizeof(double),1,binarydatafile);
fwrite(&iter, sizeof(int),1,binarydatafile);
fwrite(scrap, sizeof(double), 8, binarydatafile);
for (i=0; i < nxi; ++i) {
  fwrite(ug[i], vector5size, neta, binarydatafile);
  fwrite(pg[i], doublesize, neta, binarydatafile);
}
fclose(binarydatafile);

/* Write an errorfile */
errorfile = fopen("errorplot", "w");
fprintf(errorfile, "%d %d\n", nxi, neta);
for (j = 0; j < neta; ++j)
  for (i = 0; i < nxi; ++i)
    fprintf(errorfile, "%f %f %f\n", z[i][j], r[i][j],
    SQ(pg[i][j]-pg0[i][j]));
  fclose(errorfile);
}
```

```
/* memory.c */
#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

allocatememory()
/* This routine actually allocates all the memory */
/* NOTE THAT THERE IS NO ERROR CHECKING HERE; IF THE MACHINE RUNS OUT */
/* OF MEMORY THE PROGRAM CRASHES, NO ANDS IFS ORS or BUTS */
{
  int i, j;

  /* Allocate memory */
  hgin = (double *) (calloc (neta, sizeof(double)));
  ugin = (double *) (calloc (neta, sizeof(double)));
  theta = (double *) (calloc (nxi, sizeof(double)));

  pg = (double **) (calloc (nxi, sizeof(double *)));
  pg0 = (double **) (calloc (nxi, sizeof(double *)));
  pgp = (double **) (calloc (nxi, sizeof(double *)));


  tg = (double **) (calloc (nxi, sizeof(double *)));
  r =  (double **) (calloc (nxi, sizeof(double *)));
  z =  (double **) (calloc (nxi, sizeof(double *)));
  d =  (vector5 **) (calloc (nxi, sizeof(double *)));
  d2 = (vector5 **) (calloc (nxi, sizeof(double *)));
  f1 = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  f2 = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  fg = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  g1 = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  g2 = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  gg = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  sg = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  sxg = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  syg = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  ug = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  ugp = (vector5 **) (calloc (nxi, sizeof(vector5 *)));
  /* Allocate Metrics */
  dxidz =  (double **) (calloc (nxi, sizeof(double *)));
  dxidr =  (double **) (calloc (nxi, sizeof(double *)));
  detadz =  (double **) (calloc (nxi, sizeof(double *)));
  detadr =  (double **) (calloc (nxi, sizeof(double *)));
  xirr =  (double **) (calloc (nxi, sizeof(double *)));
  xizz =  (double **) (calloc (nxi, sizeof(double *)));
  etarr =  (double **) (calloc (nxi, sizeof(double *)));
  etazz =  (double **) (calloc (nxi, sizeof(double *)));
  jacobian =  (double **) (calloc (nxi, sizeof(double *)));
  drdeta =  (double **) (calloc (nxi, sizeof(double *)));
  drdxi =  (double **) (calloc (nxi, sizeof(double *)));
  dzdeta =  (double **) (calloc (nxi, sizeof(double *)));
  dzdxi =  (double **) (calloc (nxi, sizeof(double *)));

  for (j = 0; j < nxi; ++j) {
    pg[j] = (double *) (calloc(neta, sizeof(double)));
    pg0[j] = (double *) (calloc(neta, sizeof(double)));
    pgp[j] = (double *) (calloc(neta, sizeof(double)));
    tg[j] = (double *) (calloc (neta, sizeof (double)));
    r[j] =  (double *) (calloc(neta, sizeof(double)));
    z[j] =  (double *) (calloc(neta, sizeof(double)));
    d[j] =  (vector5 *) (calloc (neta, sizeof(vector5)));
    d2[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    f1[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    f2[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    fg[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    g1[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    g2[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    gg[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    sg[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    sxg[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    syg[j] = (vector5 *) (calloc (neta, sizeof(vector5)));
    ug[j] = (vector5 *) (calloc(neta, sizeof(vector5)));
    ugp[j]= (vector5 *) (calloc(neta, sizeof(vector5)));

    dxidz[j] =  (double *) (calloc (neta, sizeof(double)));
    dxidr[j] =  (double *) (calloc (neta, sizeof(double)));
    detadz[j] =  (double *) (calloc (neta, sizeof(double)));
    detadr[j] =  (double *) (calloc (neta, sizeof(double)));
    xirr[j] =  (double *) (calloc (neta, sizeof(double)));
    xizz[j] =  (double *) (calloc (neta, sizeof(double)));
    etarr[j] =  (double *) (calloc (neta, sizeof(double)));
    etazz[j] =  (double *) (calloc (neta, sizeof(double)));
    jacobian[j] =  (double *) (calloc (neta, sizeof(double)));
    drdeta[j] =  (double *) (calloc (neta, sizeof(double)));
    drdxi[j] =  (double *) (calloc (neta, sizeof(double)));
    dzdeta[j] =  (double *) (calloc (neta, sizeof(double)));
    dzdxi[j] =  (double *) (calloc (neta, sizeof(double)));
  }
}
```

```
/* File metrics.c */
/* Contains a routine to calculate grid metrics */
/* UNTESTED; I'm just going to forge ahead.... */

#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

calculatemetrics()
/* Calculate the Grid Metrics */
{
  /* Allocate and set up memory */
  double **drdnn, **drdxx, **drdxn, **dzdnn, **dzdxx, **dzdxn, **djdeta,
         **djdxi;
  int i,j;
  int neta1,nxi1;

  neta1 = neta-1;
  nxi1 = nxi-1;

  drdnn = (double **) (calloc (nxi, sizeof(double *)));
  drdxx =  (double **) (calloc (nxi, sizeof(double *)));
  drdxn =  (double **) (calloc (nxi, sizeof(double *)));
  dzdnn =  (double **) (calloc (nxi, sizeof(double *)));
  dzdxx =  (double **) (calloc (nxi, sizeof(double *)));
  dzdxn =  (double **) (calloc (nxi, sizeof(double *)));
  djdeta =  (double **) (calloc (nxi, sizeof(double *)));
  djdxi =  (double **) (calloc (nxi, sizeof(double *)));

  for (j = 0; j < nxi; ++j) {
    drdnn[j] = (double *) (calloc (neta, sizeof(double)));
    drdxx[j] =  (double *) (calloc (neta, sizeof(double)));
    drdxn[j] =  (double *) (calloc (neta, sizeof(double)));
    dzdnn[j] =  (double *) (calloc (neta, sizeof(double)));
    dzdxx[j] =  (double *) (calloc (neta, sizeof(double)));
    dzdxn[j] =  (double *) (calloc (neta, sizeof(double)));
    djdeta[j] =  (double *) (calloc (neta, sizeof(double)));
    djdxi[j] =  (double *) (calloc (neta, sizeof(double)));
  }

  for (j = 0; j < neta; ++j) {
    for (i = 0; i < nxi; ++i) {
      if (i == 0 && j == 0) {
dzdxi[i][j]=(z[i+1][j]-z[i][j]);
dzdeta[i][j]=(z[i][j+1]-z[i][j]);
drdxi[i][j]=(r[i+1][j]-r[i][j]);
drdeta[i][j]=(r[i][j+1]-r[i][j]);
drdnn[i][j]=r[i][j]-2.0*r[i][j+1]+r[i][j+2];
drdxx[i][j]=r[i][j]-2.0*r[i+1][j]+r[i+2][j];
dzdnn[i][j]=z[i][j]-2.0*z[i][j+1]+z[i][j+2];
dzdxx[i][j]=z[i][j]-2.0*z[i+1][j]+z[i+2][j];
drdxn[i][j]=(r[i+1][j+1]-r[i+1][j]-r[i][j+1]
     +r[i][j]);
dzdxn[i][j]=(z[i+1][j+1]-z[i+1][j]-z[i][j+1]
     +z[i][j]);
      }
      else if (i == 0 && j == neta1) {
dzdxi[i][j]=(z[i+1][j]-z[i][j]);
dzdeta[i][j]=(z[i][j]-z[i][j-1]);
drdxi[i][j]=(r[i+1][j]-r[i][j]);
drdeta[i][j]=(r[i][j]-r[i][j-1]);
drdnn[i][j]=r[i][j]-2.0*r[i][j-1]+r[i][j-2];
drdxx[i][j]=r[i][j]-2.0*r[i+1][j]+r[i+2][j];
dzdnn[i][j]=z[i][j]-2.0*z[i][j-1]+z[i][j-2];
dzdxx[i][j]=z[i][j]-2.0*z[i+1][j]+z[i+2][j];
drdxn[i][j]=(r[i+1][j]-r[i+1][j-1]-r[i][j]
     +r[i][j-1]);
dzdxn[i][j]=(z[i+1][j]-z[i+1][j-1]-z[i][j]
     +z[i][j-1]);
      }
      else if (i == 0 && j < neta1 && j > 0) {
dzdxi[i][j]=(z[i+1][j]-z[i][j]);
dzdeta[i][j]=(z[i][j+1]-z[i][j-1])/2.0;
drdxi[i][j]=(r[i+1][j]-r[i][j]);
drdeta[i][j]=(r[i][j+1]-r[i][j-1])/2.0;
drdnn[i][j]=r[i][j+1]-2.0*r[i][j]+r[i][j-1];
drdxx[i][j]=r[i][j]-2.0*r[i+1][j]+r[i+2][j];
dzdnn[i][j]=z[i][j+1]-2.0*z[i][j]+z[i][j-1];
dzdxx[i][j]=z[i][j]-2.0*z[i+1][j]+z[i+2][j];
drdxn[i][j]=0.0*(r[i+1][j+1]-r[i+1][j-1]-r[i][j+1]
 +r[i][j-1]);
dzdxn[i][j]=0.5*(z[i+1][j+1]-z[i+1][j-1]-z[i][j+1]
 +z[i][j-1]);
      }
      else if (i == nxi1 && j == 0) {
dzdxi[i][j]=(z[i][j]-z[i-1][j]);
dzdeta[i][j]=(z[i][j+1]-z[i][j]);
drdxi[i][j]=(r[i][j]-r[i-1][j]);
drdeta[i][j]=(r[i][j+1]-r[i][j]);
drdnn[i][j]=r[i][j]-2.0*r[i][j+1]+r[i][j+2];
drdxx[i][j]=r[i][j]-2.0*r[i-1][j]+r[i-2][j];
dzdnn[i][j]=z[i][j]-2.0*z[i][j+1]+z[i][j+2];
dzdxx[i][j]=z[i][j]-2.0*z[i-1][j]+z[i-2][j];
drdxn[i][j]=(r[i][j+1]-r[i][j]-r[i-1][j+1]
     +r[i-1][j]);
dzdxn[i][j]=(z[i][j+1]-z[i][j]-z[i-1][j+1]
     +z[i-1][j]);
      }
      else if (i == nxi1 && j == neta1) {
dzdxi[i][j]=(z[i][j]-z[i-1][j]);
dzdeta[i][j]=(z[i][j]-z[i][j-1]);
drdxi[i][j]=(r[i][j]-r[i-1][j]);
drdeta[i][j]=(r[i][j]-r[i][j-1]);
drdnn[i][j]=r[i][j]-2.0*r[i][j-1]+r[i][j-2];
drdxx[i][j]=r[i][j]-2.0*r[i-1][j]+r[i-2][j];
dzdnn[i][j]=z[i][j]-2.0*z[i][j-1]+z[i][j-2];
dzdxx[i][j]=z[i][j]-2.0*z[i-1][j]+z[i-2][j];
drdxn[i][j]=(r[i][j]-r[i][j-1]-r[i-1][j]
     +r[i-1][j-1]);
dzdxn[i][j]=(z[i][j]-z[i][j-1]-z[i-1][j]
     +z[i-1][j-1]);
      }
      else if (i == nxi1 && j != neta1 && j !=0) {
dzdxi[i][j]=(z[i][j]-z[i-1][j]);
dzdeta[i][j]=(z[i][j+1]-z[i][j-1])/2.0;
drdxi[i][j]=(r[i][j]-r[i-1][j]);
drdeta[i][j]=(r[i][j+1]-r[i][j-1])/2.0;
drdnn[i][j]=r[i][j+1]-2.0*r[i][j]+r[i][j-1];
drdxx[i][j]=r[i][j]-2.0*r[i-1][j]+r[i-2][j];
```

```
dzdnn[i][j]=z[i][j+1]-2.0*z[i][j]+z[i][j-1];
dzdxx[i][j]=z[i][j]-2.0*z[i-1][j]+z[i-2][j];
drdxn[i][j]=0.5*(r[i][j+1]-r[i][j-1]-r[i-1][j+1]
 +r[i-1][j-1]);
dzdxn[i][j]=0.5*(z[i][j+1]-z[i][j-1]-z[i-1][j+1]
 +z[i-1][j-1]);
      }
      else if (j == 0 && i > 0 && i < nxi1)  {
dzdxi[i][j]=(z[i+1][j]-z[i-1][j])/2.0;
dzdeta[i][j]=(z[i][j+1]-z[i][j]);
drdxi[i][j]=(r[i+1][j]-r[i-1][j])/2.0;
drdeta[i][j]=(r[i][j+1]-r[i][j]);
drdnn[i][j]=r[i][j]-2.0*r[i][j+1]+r[i][j+2];
drdxx[i][j]=r[i+1][j]-2.0*r[i][j]+r[i-1][j];
dzdnn[i][j]=z[i][j]-2.0*z[i][j+1]+z[i][j+2];
dzdxx[i][j]=z[i+1][j]-2.0*z[i][j]+z[i-1][j];
drdxn[i][j]=0.5*(r[i+1][j+1]-r[i+1][j]-r[i-1][j+1]
 +r[i-1][j]);
dzdxn[i][j]=0.5*(z[i+1][j+1]-z[i+1][j]-z[i-1][j+1]
 +z[i-1][j]);
      }
      else if (j == neta1 && i != nxi1 && i != 0) {
dzdxi[i][j]=(z[i+1][j]-z[i-1][j])/2.0;
dzdeta[i][j]=(z[i][j]-z[i][j-1]);
drdxi[i][j]=(r[i+1][j]-r[i-1][j])/2.0;
drdeta[i][j]=(r[i][j]-r[i][j-1]);
drdnn[i][j]=r[i][j]-2.0*r[i][j-1]+r[i][j-2];
drdxx[i][j]=r[i+1][j]-2.0*r[i][j]+r[i-1][j];
dzdnn[i][j]=z[i][j]-2.0*z[i][j-1]+z[i][j-2];
dzdxx[i][j]=z[i+1][j]-2.0*z[i][j]+z[i-1][j];
drdxn[i][j]=0.5*(r[i+1][j]-r[i+1][j-1]-r[i-1][j]
 +r[i-1][j-1]);
dzdxn[i][j]=0.5*(z[i+1][j]-z[i+1][j-1]-z[i-1][j]
 +z[i-1][j-1]);
      }
      else {
dzdxi[i][j]=(z[i+1][j]-z[i-1][j])/2.0;
dzdeta[i][j]=(z[i][j+1]-z[i][j-1])/2.0;
drdxi[i][j]=(r[i+1][j]-r[i-1][j])/2.0;
drdeta[i][j]=(r[i][j+1]-r[i][j-1])/2.0;
drdnn[i][j]=r[i][j+1]-2.0*r[i][j]+r[i][j-1];
drdxx[i][j]=r[i+1][j]-2.0*r[i][j]+r[i-1][j];
dzdnn[i][j]=z[i][j+1]-2.0*z[i][j]+z[i][j-1];
dzdxx[i][j]=z[i+1][j]-2.0*z[i][j]+z[i-1][j];
drdxn[i][j]=0.25*(r[i+1][j+1]-r[i+1][j-1]-r[i-1][j+1]
  +r[i-1][j-1]);
dzdxn[i][j]=0.25*(z[i+1][j+1]-z[i+1][j-1]-z[i-1][j+1]
  +z[i-1][j-1]);
      }
    }
  }
  for (j = 0; j < neta; ++j) {
    for (i = 0; i < nxi; ++i) {
      jacobian[i][j]=dzdxi[i][j]*drdeta[i][j];
      detadr[i][j]=dzdxi[i][j]/jacobian[i][j];
      detadz[i][j]= -drdxi[i][j]/jacobian[i][j];
      dxidr[i][j]=0.0;
      dxidz[i][j]=drdeta[i][j]/jacobian[i][j];
      djdeta[i][j]=drdnn[i][j]*dzdxi[i][j]+drdeta[i][j]*dzdxn[i][j];
      djdxi[i][j]=drdxn[i][j]*dzdxi[i][j]+drdeta[i][j]*dzdxx[i][j];
    }
  }

/*  Calculate the 2nd derivative metrics */
  for (j = 1; j < neta-1; ++j) {
    for (i = 1; i < nxi-1; ++i) {
      etazz[i][j]= -(dxidz[i][j]*(drdxx[i][j]+detadz[i][j]*djdxi[i][j])
    +detadz[i][j]*(drdxn[i][j]+detadz[i][j]*djdeta[i][j]))
             /jacobian[i][j];
      xizz[i][j]=(dxidz[i][j]*(drdxn[i][j]-dxidz[i][j]*djdxi[i][j])
  +detadz[i][j]*(drdnn[i][j]-dxidz[i][j]*djdeta[i][j]))
            /jacobian[i][j];
      etarr[i][j]=(dxidr[i][j]*(dzdxx[i][j]-detadr[i][j]*djdxi[i][j])
                  +detadr[i][j]*(dzdxn[i][j]-detadr[i][j]*djdeta[i][j]))
            /jacobian[i][j];
      xirr[i][j]=0.0;
    }
  }

  /* Free Memory */
  for (j = 1; j < nxi; ++j) {
    free (drdnn[j]);
    free(drdxx[j]);
    free(drdxn[j]);
    free(dzdnn[j]);
    free(dzdxx[j]);
    free(dzdxn[j]);
    free(djdeta[j]);
    free(djdxi[j]);
  }
  free (drdnn);
  free (drdxx);
  free (drdxn);
  free(dzdnn);
  free(dzdxx);
  free(dzdxn);
  free(djdeta);
  free(djdxi);
}
```

```
/* File time.c */

#include <math.h>
#include <stdio.h>
#include "nozzle.h"
#include "globals.h"

calculatetimestep(deltat)
/* Uses CFL Condition */
/* Appears to work, 5/15/92 */
double *deltat;
{
  double **a, **dz, **dr;
  double safety_factor, time;
  int i,j;

  /* Allocate Memory */
  a =  (double **) (calloc (nxi, sizeof(double *)));
  dz = (double **) (calloc (nxi, sizeof(double *)));
  dr = (double **) (calloc (nxi, sizeof(double *)));
  for (j = 0; j < nxi; ++j) {
    a[j] = (double *) (calloc(neta, sizeof(double)));
    dz[j] = (double *) (calloc(neta, sizeof(double)));
    dr[j] = (double *) (calloc(neta, sizeof(double)));
  }

  safety_factor = SAFETY_FACTOR;
  *deltat = 10000.0;

  /* Calculate Derivatives using differencing */
  for (j = 1; j < (neta - 1); ++j) {
    for (i = 1; i < (nxi - 1); ++i) {
      a[i][j] = sqrt(GAMMA * (pg[i][j] / ug[i][j].a));
      dz[i][j] = (z[i+1][j]-z[i-1][j])/2.0;
      dr[i][j] = (r[i][j+1]-r[i][j-1])/2.0;
    }
  }

  /* Apply CFL Condition */
  for (j = 1; j < (neta -1); ++j) {
    for (i = 1; i < (nxi -1); ++i) {
      time=(safety_factor*dz[i][j])/(a[i][j] + fabs(ug[i][j].b/ug[i][j].a));
      if (time < *deltat) {
*deltat = time;
      }
      time=(safety_factor*dr[i][j])/(a[i][j] + fabs(ug[i][j].c/ug[i][j].a));
      if (time < *deltat) {
*deltat = time;
      }
    }
  }

  /* Free memory */
  for (j = 0; j < nxi; ++j) {
    free (a[j]);
    free (dz[j]);
    free (dr[j]);
  }
  free (a);
  free (dz);
  free (dr);
}
```

```
/* Basic include file for nozzle.c */
/* The axisymmetric, invicid flow navier stokes equation solver */

/* Debugging Flag */
/* Set to true to turn off Source Terms */
#define NOSOURCE 0

/* Inlet Conditions */
/* Presently set for total temperature = 2500 */
#define INLET_TEMPERATURE 2358.0
#define INLET_DENSITY 0.351
#define INLET_Z_VELOCITY 2037.5
#define INLET_ROTATION_RATE 10000.0

/* Timestep Safety Factor */
#define SAFETY_FACTOR 0.8

/* Damping Coefficients */
#define DXI2 1e-6
#define DETA2 1e-6

#define DXI4 0.0
#define DETA4 0.0

/* Moving them lower for the ten thousand case from 8e-7 */
/* For 10000 rad/sec, damping set at 5e-7 */
/* For 5000 rad/sec, I'm moving the damping up to 8e-7 */
/* That didn't work;  try moving down to 3e-7

/* Data type for state vectors */
typedef struct {
  double x;
  double y;
} vector2;

typedef struct {
  double x;
  double y;
  double z;
} vector3;

typedef struct {
  double a;
  double b;
  double c;
  double d;
  double e;
} vector5;

/* Thermodynamic Constants for H2 */
#define RGAS 4157.3              /* J/kg-K */
#define GAMMA 1.4

/* For NERVA model, CHAMBER_TEMP = 2500.00 */
#define CHAMBER_TEMP 2500.0
/* For NERVA model, CHAMBER_P = 3.44e6 */
#define CHAMBER_P 3.44e6

/* Throat Geometry */
/* For grid.paper, THROAT_AREA = 0.04445 */
/* For grid.paper, THROAT_LOCATION = 0.0771 */
/* For grid.paper3 THROAT_AREA = 0.0444 */
/* For grid.paper3 THROAT_LOCATION = 0.600 */
/* for grid.nozzle, THROAT_AREA = 0.0438 */
/* For grid.pipe, THROAT_LOCATION = 1.5764 */
/* for grid.pipe, THROAT_AREA = 0.04445 */
/* For grid.pipe2 and pipe6, THROAT_AREA = 0.0444*/
/* For grid.pipe2 and pipe6, THROAT_LOCATION = 1.1045 */
#define THROAT_AREA 0.0444
#define THROAT_LOCATION 1.1045

/* Generic Constants */
#define PI 3.14159265358979323846
#define TRUE 1
#define FALSE 0
#define EPSILON 1e-10

#define SQ(X) ((X)*(X))
```

```
extern char   predictor;       /* boolean flag for predictor/corrector */
extern double *hgin,           /* inlet enthalpy */
              *theta,          /* Upper Wall Angle */
              *ugin;           /* inlet velocity */
extern double **pg,            /* Cell pressure */
              **pgp,           /* Predictor pg */
              **pg0,           /* Old Cell Pressures (for Error Calc ) */
              **tg;            /* Cell temperatures */
extern double **r,             /* Grid points */
              **z;
extern double dxi2, deta2,     /* Second order damping terms */
              dxi4, deta4,     /* Fourth order damping terms */
              dcenterxi2, dcentereta2;
extern double ftheta,
              mdot,
              time,
              twall;
/* For metrics */
extern double **dxidz, **dxidr, **detadz, **detadr, **xirr, **xizz, **etarr;
extern double **etazz, **jacobian, **drdeta, **drdxi, **dzdeta, **dzdxi;
extern int nxi, neta;          /* number of xi and eta grid cells */
extern int iter;
extern vector5 **d,            /* Fourth Order Damping Term */
               **d2,           /* Second Order Damping Term */
               **f1, **f2,     /* Used in Flux; here for speed purposes */
               **fg,           /* Flux term F */
               **g1, **g2,     /* Used in FluxG; here for speed purposes */
               **gg,           /* Flux term G */
               **sg,           /* Source Term */
               **sxg,          /* Damping Term */
               **syg,          /* Damping Term */
               **ug,           /* State Vector */
               **ugp;          /* Copy of State Vector for Predictor */

/* For debugging */
extern FILE *debuggingoutput;
```

This section contains the code used to model the flow of particles. This code is designed to be used as a post-processor, and takes results from the fluid modeling code as input. It is based on a differential equations solver from Lawrence Livermore Laboratories called Lsode. Lsode is written in FORTRAN, and it is not included in this listing. One routine written in C, called integrate, is called by Lsode on each iteration and is used to calculate the new values at each time step.

This code consists of five modules and one header file.

particle.c: contains the main routine and the integrate routine
dlsode.f: (not listed) contains the differential equation solver lsode.
io.c: contains input/output routines
memory.c: allocates dynamic memory
vector.c: contains vector manipulation routines

And the header file

particle.h: contains general information

```
/*************************************************************************/
/* Test Particle Tracking Programs                                       */
/* This takes a fluid data file taken from "nozzle", places a particle */
/* In it, and traces the path of the particle                            */
/* At present, it only works with certain grid geometries                */
/* Uses dlsode, a module written in FORTRAN                              */
/* By bamf; 7/7/92                                                       */
/*************************************************************************/
/* Modified to include pressure forces on the particle, 10/7/92          */
/* By bamf */
/*************************************************************************/
#include <math.h>
#include <stdio.h>
#include "particle.h"

/* Programming Flag; if set to TRUE, particles begin with the flow's swirl
   Velocity.  If not, they begin with a swirl velocity of 0, but a non-zero
   axial velocity */
#define INITIAL_SWIRL 0

/* Programming Flag; if set to TRUE particle tracks will be saved also.
   in GNUplot format.  Be careful using this;  it chews up disk space
   rapidly. */
#define SAVE_PLOT 0

#define DELTA_T 0.000005
#define SMALLEST_RADIUS 1e-6
#define LARGEST_RADIUS 1e-4
#define INCREMENT_RADIUS ((1e-4-1e-6)/10)

/* Declaration of Globals */
cylindrical_coord fluid_velocity, **particle_velocity, **particle_position;
double **r, **z;                    /* The grid */
double **pg, **tg;                  /* The temperature */
double ftheta, time, **timerecord, mdot;
double particle_radius,
       particle_mass,
       temperature;
int iter, nxi, neta;
vector5 **ug;

main(argc, argv)
     int argc;
     char *argv[];
{
  extern void integrate(), jacobian(), lsode_();

  /* Flags and Stuff for dlsode() */
  int neq = NUMBER_OF_EQUATIONS;            /* Number of Equations */
  double variables[NUMBER_OF_EQUATIONS];    /* The Variables */
  double time, time2;                       /* Starting and Ending Time */
  int itol = 1;                             /* Random Flag */
  double  atol = 1e-10, rtol = 1e-5;        /* Error Variables */
  /* rtol was originally 1e-5 */
  /* atol was 1e-15 */

  int iopt = 0, istate, itask = 1;       /* Flags */
  int lrw = 20 + 16 * NUMBER_OF_EQUATIONS;      /* Length of real work array */
  double rwork[20 + 16 * NUMBER_OF_EQUATIONS];
```



```
  int liw = 20;                             /* Length of integer work array */
  int iwork[20];                            /* Integer Work Array */
  int mf = NONSTIFF;                        /* Type of System Flag */

  /* Ok, useful variable declarations */
  char answer[30];                           /* Dummy answer variable */
  char dataname[100], fname[100], gridname[100]; /* File Names */
  char multiple_particle,                   /* Boolean Flag */
       particle_found;                      /* Boolean Flag */
  double initial_radius,
         initial_swirl_velocity,
         initial_velocity,
         particle_density;
  double inlet_radius,
         particle_alignment_angle;          /* Initial theta  of the line
     of released particles */
  int endstep[MAX_PARTICLE],
      i, j, l, s,                           /* Dummies */
      gridr,                                /* The coordinates of the grid  */
      gridz,                                /* cell containing the particle */
      number_of_particles,                  /* # of particles to release */
      particle_count,
      p_count,
      step;

  /* For analysis purposes */
#include "data.h"

  /* Take the inputs either from the command line or the old way */
  if (argc > 1) {
    readCmdLine(argc, argv, dataname, fname, gridname, &particle_density,
&particle_radius);
    printf("\nFluid file: %s", fname);
    printf("\nGrid file: %s", gridname);
    printf("\nData file: %s", dataname);
    printf("\nParticle Density: %f", particle_density);
    printf("\nParticle Radius: %f", particle_radius);
    multiple_particle = TRUE;
    number_of_particles = 200;
  }
  else {
    /* First, get the flow data file information */
    printf("\nParticle Tracking Programs, Version 1");
    printf("\nWhat is the run file prefix? ");
    scanf("%s", fname);
    printf("\nWhat is the Grid file suffix? ");
    scanf("%s", gridname);
    printf("\nWhat is the data file prefix? ");
    scanf("%s", dataname);
    printf("\nwhat's the particle density? ");
    scanf("%lf", &particle_density);
    printf("\nWhat's the particle radius? ");
  scanf("%lf", &particle_radius);
    multiple_particle = TRUE;
    printf("\nNumber of particles to release: ");
    scanf("%d", &number_of_particles);
  }

  /* Load the Fluid Data Files */
  /* This routine is in io.c */
```

```
  getgrid(gridname);
  getflow(fname);

  /* Allocate memory */
  allocateTrajectoryRecords(number_of_particles);

/*********************************************************************

  for (particle_radius = SMALLEST_RADIUS; particle_radius <= LARGEST_RADIUS;
       particle_radius = particle_radius + INCREMENT_RADIUS) {

*********************************************************************/

    printf("\nFile: %s Radius: %g", fname, particle_radius);

    /* Clear the particle count array */
    for (i = 0; i < lend; ++i)
      for (j = 0; j < send; ++j)
particle_caught[i][j] = 0;

    particle_mass = particle_density * (4.0/3.0) * PI
      * particle_radius*particle_radius*particle_radius;

    inlet_radius = r[0][neta-1];
    for(particle_count = 0; particle_count < number_of_particles;
++particle_count) {
      /* Set initial Velocities */
      /* These set ur = 0, uz = inlet axial; utheta = 0.0; */
      /* First, "find" the particle on the inlet grid */
      if (multiple_particle)
initial_radius =
  ((double) particle_count/(double)number_of_particles) * inlet_radius;
      else
initial_radius = DEFAULT_R;

      particle_found = FALSE;
      for (j = 0; j < neta && particle_found == FALSE; ++j) {
if (r[0][j] > initial_radius) {
  if (j > 0)
    j = j-1; /*(compensates for the j about to be added on) */
  particle_found = TRUE;
}
      }

#if PARTICLE_SWIRL
      if (j == neta) {
initial_velocity = ug[0][j-1].b / ug[0][j-1].a;
initial_swirl_velocity = ug[0][j-1].e / ug[0][j-1].a;
      }
      else {
initial_velocity = ug[0][j].b / ug[0][j].a;
initial_swirl_velocity = ug[0][j].e / ug[0][j].a;
      }
      particle_alignment_angle = 0.0;
#else
      if (j==neta) {
initial_velocity = ug[0][j-1].b / ug[0][j-1].a;
initial_swirl_velocity = 0.0;
      }
      else {
```



```
initial_velocity = ug[0][j].b / ug[0][j-1].a;
initial_swirl_velocity = 0.0;
      }
#endif

      variables[0] = 0.0;                          /* Ur */
      variables[1] = initial_swirl_velocity;       /* Utheta */
      variables[2] = initial_velocity;             /* Uz */
      variables[3] = initial_radius;               /* r */
      variables[4] = particle_alignment_angle;     /* theta */
      variables[5] = 0.0;                          /* z */

      particle_velocity[particle_count][0].r = variables[0];
      particle_velocity[particle_count][0].theta = variables[1];
      particle_velocity[particle_count][0].z = variables[2];
      particle_position[particle_count][0].r = variables[3];
      particle_position[particle_count][0].theta = variables[4];
      particle_position[particle_count][0].z = variables[5];

      timerecord[particle_count][0] = 0.0;
      time = 0.0;
      time2 = 0.0;
      istate = 1;

      /* Start Tracking the Particle */
      for (step = 0; step < MAX_STEP; ++step) {
time2 = time2 + DELTA_T;

/* This is a FORTRAN call, so you include the underscore and call */
/* By reference */
lsode_(integrate, &neq, variables, &time, &time2, &itol, &rtol, &atol,
       &itask, &istate, &iopt, rwork, &lrw, iwork, &liw, jacobian,
       &mf);
if (istate < 0) {
  printf("Error: %d", istate);
  exit();
}

/* Record the new position and stuff */
particle_velocity[particle_count][step+1].r = variables[0];
particle_velocity[particle_count][step+1].theta = variables[1];
particle_velocity[particle_count][step+1].z = variables[2];
particle_position[particle_count][step+1].r = variables[3];
particle_position[particle_count][step+1].theta = variables[4];
particle_position[particle_count][step+1].z = variables[5];
timerecord[particle_count][step+1] = time;

/* Check to see if the particle has run off the grid */
particle_found = FALSE;
for (i = 1; (i < nxi) && (particle_found == FALSE); ++i)
  if (z[i][0] > particle_position[particle_count][step].z) {
    gridz = i - 1;
    particle_found = TRUE;
  }

      /* Deal with the case where the particle has run off the grid... */
if (particle_found == FALSE)
  goto end_tracking;

particle_found = FALSE;
```

```
      for (j = 1; (j < neta) && (particle_found == FALSE); ++j)
if (r[gridz][j] > particle_position[particle_count][step].r) {
  gridr = j - 1;
  particle_found = TRUE;
}
if (particle_found == FALSE)
  goto end_tracking;
      }

    end_tracking:
      endstep[particle_count] = step;
      if (!(particle_count%20))
printf("\nParticle track %d ended at step: %d", particle_count, step);
    } /* ends "for (count)" */

    /* Save to File */
    /* This routine in io.c */
    /*    saveTrack(endstep, fname, number_of_particles, step); */

    /* Saves particle tracks in gnuplot format */
    /* It's awfully memory intensive if you use it.... */
#if SAVE_PLOT
      savePlot(dataname, endstep, number_of_particles, particle_radius);
#endif

    /* Does a particle removal analysis */
    for (p_count = 0; p_count < number_of_particles;
 ++p_count) {
      /* Reset the length counter for each track */
      l = 0;

      /* Start tracking particle */
      for (step = 0; step < endstep[p_count]; ++step) {
/* First, find the particle */
for (i = 0, particle_found=FALSE;
     i < nxi && particle_found == FALSE; ++i) {
  if (z[i][0] > particle_position[p_count][step].z) {
    if (i > 0)
      i--;
    particle_found = TRUE;
  }
}

/* Then, check and see if it has crossed cut off length */
if (length[l] < particle_position[p_count][step].z) {
  /* Now check to see if the particle has been caught */
  for (s = 0; s < send; ++s) {
    if (sratio[s] > (0.0 - EPSILON) &&
(particle_position[p_count][step].r >=
        (r[i][neta-1] - sratio[s] * r[i][neta-1])))
      ++particle_caught[l][s];
  }
  /* Advance to it goes to the next length */
  ++l;
  /* If we've reached the end of the lengths, we can stop counting */
  if (length[l] < (0.0+EPSILON))
    step = endstep[p_count];
}
      }
      /* Now, at the end of the track, find the particle and see if it left
```



```
 Or collided with a wall */
      for (l = 1; l < lend; ++l) {
if (length[l] > particle_position[p_count][endstep[p_count]-1].z) {
  /* Since it collided with the wall, assume it got caught in all  */
  /* Slimming ratios */
  for (s = 0; s < send; ++s)
    ++particle_caught[l][s];
}
      }
    }

  /* Now convert the "particles caught" to eta and true eta */
  /* (eta = linear efficiency ratio; trueeta = true efficiency) */
  for (l = 0; l < lend; ++l)
    for (s = 0; s < send; ++s) {
      if (sratio[s] >= (0.0-EPSILON) && length[l] > 0.0) {
eta[l][s] =
  (double) particle_caught[l][s] / (double) number_of_particles;
trueeta[l][s] = 1.0 - SQ(1.0 - eta[l][s]);
      }
      else {
eta[l][s] = -1.0;
trueeta[l][s] = -1.0;
      }
    }

  /* Now we need to write the output file; routine in io.c */
  saveData(dataname, eta, lend, length, particle_radius, send,
   sratio, trueeta);
  /*****************************************************************
    }    /* Ends the "particle_radius" loop
  *****************************************************************/
}

double calculateIntegralCrossSection(temperature)
/* This gets the integral cross section for viscosity calculations */
/* Using a lookup table and linear interpolation */
/* Tested numerically, but not graphically, 6/17/92 */
double temperature;
{
  double omega, omega1, omega2;
  double temp1, temp2;                    /* Temperatures */

 /* This is a lookup table */
  if (temperature  < 300.0) {
    omega1 = 7.340; omega2 = 6.747;
    omega = omega1 - (omega2 - omega1) * (300.0 - temperature) / 200.0;
  }
  else if (temperature < 500.0) {
    omega1 = 7.340;  omega2 = 6.747;
    temp1 = 300.0; temp2 = 500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 1000.0) {
    omega1 = 6.747, omega2 = 6.002;
    temp1 = 500.0; temp2 = 1000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 1500.0) {
    omega1 = 6.002; omega2 = 5.593;
```

```
    temp1 = 1000.0, temp2 = 1500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 2000.0) {
    omega1 = 5.593; omega2 = 5.328;
    temp1 = 1500.0; temp2 = 2000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 2500.0) {
    omega1 = 5.328; omega2 = 5.020;
    temp1 = 2000.0; temp2 = 2500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 3000.0) {
    omega1 = 5.020; omega2 = 4.732;
    temp1 = 2500.0; temp2 = 3000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 3500.0) {
    omega1 = 4.732;  omega2 = 4.495;
    temp1 = 3000.0; temp2 = 3500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 4000.0) {
    omega1 = 4.495; omega2 = 4.293;
    temp1 = 3500.0;  temp2 = 4000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 4500.0) {
    omega1 = 4.293; omega2 = 4.119;
    temp1 = 4000.0; temp2 = 4500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 5000.0) {
    omega1 = 4.119; omega2 = 3.966;
    temp1 = 4500.0;  temp2 = 5000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 5500.0) {
    omega1 = 3.966; omega2 = 3.831;
    temp1 = 5000.0; temp2 = 5500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 6000.0) {
    omega1 = 3.831;  omega2 = 3.710;
    temp1 = 5500.0; temp2 = 6000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 6500.0) {
    omega1 = 3.710;  omega2 = 3.598;
    temp1 = 6000.0; temp2 = 6500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 7000.0) {
    omega1 = 3.598; omega2 = 3.497;
    temp1 = 6500.0; temp2 = 7000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 7500.0) {
    omega1 = 3.497;  omega2 = 3.405;
```



```
    temp1 = 7000.0; temp2 = 7500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 8000.0) {
    omega1 = 3.405;  omega2 = 3.320;
    temp1 = 7500.0; temp2 = 8000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 8500.0) {
    omega1 = 3.320; omega2 = 3.230;
    temp1 = 8000.0; temp2 = 8500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 9000.0) {
   omega1 = 3.230; omega2 = 3.166;
   temp1 = 8500.0; temp2 = 9000.0;
   omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 9500.0) {
    omega1 = 3.166; omega2 = 3.097;
    temp1 = 9000.0;  temp2 = 9500.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 10000.0) {
    omega1 = 3.097; omega2 = 3.033;
    temp1 = 9500.0; temp2 = 10000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 11000.0) {
    omega1 = 3.033; omega2 = 2.913;
    temp1 = 10000.0; temp2 = 11000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 12000.0) {
    omega1 = 2.913; omega2 = 2.806;
    temp1 = 11000.0; temp2 = 12000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 13000.0) {
    omega1 = 2.806; omega2 = 2.710;
    temp1 = 12000.0; temp2 = 13000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 14000.0) {
    omega1 = 2.710;  omega2 = 2.622;
    temp1 = 13000.0; temp2 = 14000.0;
    omega = omega1 + (omega2-omega1) * (temperature-temp1) / (temp2-temp1);
  }
  else if (temperature < 15000.0) {
    omega1 = 2.622; omega2 = 2.542;
    temp1 = 14000.0; temp2 = 15000.0;
    omega = omega1 + (omega2 = omega1) * (temperature - temp1) / (temp2-temp1);
  }
  else {
    omega1 = 2.622; omega2 = 2.542;
    omega = omega2 + (omega2-omega1) * (temperature - 15000.0) / 1000.0;
  }

  omega = omega * 1e-20;
  return (omega);
```

```
}

double calculateViscosity(temperature)
/* This calculates the viscosity of H2 gas                          */
/* Takes only temperature as input                                  */
/* Based on the viscosity formula used by Scott in his thesis       */
/* This is the first order approximation for the viscosity of an    */
/* ideal gas; numerically all check to order of magnitude 6/17/92   */
double temperature;
{
  double calculateIntegralCrossSection(), omega, viscosity;

  omega = calculateIntegralCrossSection(temperature);
  viscosity = 2.6693e-26 * sqrt(GAS_MOLECULAR_WT * temperature) / omega;

  return(viscosity);
}

void integrate(neq, t, y, ydot)
/* This routine is called from the FORTRAN routine dlsode */
/* It calculates and breaks down the drag forces on a sphere and is valid */
/* within 10% for 0 < Re < 2x10^5                                         */
/* It's relatively computational inefficient at the moment, as there's a  */
/* lot of variable renaming that goes on to make it more readable.....    */
/* Under construction, 7/9/92 */
int *neq;
double *t, y[], ydot[];
{
  char particle_found;
  cylindrical_coord
    drag,                           /* Drag Force Vector */
    fluid_unit_vector,              /* Unit vector in the dir. of Fluid flow
       in Inertial frame */
    particle_v,                     /* Particle Velocity Vector */
    relative_velocity,              /* Particle velocity in fluid frame */
    relative_unit_vector;           /* Inertial unit vector in relative
       velocity direction */
  int gridr, gridz, i, j;
  double
    Cd,                             /* Drag Coeff. */
    drag_force,                     /* Drag force magnitude */
    fluid_speed,                    /* Fluid speed in inertial frame */
    relative_speed,                 /* Particle speed in fluid frame */
    Re,                             /* Flow Reynolds number */
    rho,                            /* Fluid Density */
    viscosity;
  double calculateViscosity();
  extern double dot(), magnitude();

  /* First, find the cell that the particle is in. */
  /* Caution: this find routine                    */
  /* Makes the assumption that the grid lines are straight in the y dir. */
  particle_found = FALSE;
  for (i = 1; (i < nxi) && (particle_found == FALSE); ++i)
    if (z[i][0] > y[5]) {
      gridz = i - 1;
      particle_found = TRUE;
    }

  /* Deal with the case where the particle has run off the grid. */
```



```
  /* If the particle is off the grid, give it acceleration = 0 so it */
  /* Will continue on its way off the grid... */
  if (particle_found == FALSE) {
    ydot[0] = 0.0;
    ydot[1] = 0.0;
    ydot[2] = 0.0;
    goto end_integration;
  }

  particle_found = FALSE;
  for (j = 1; (j < neta) && (particle_found == FALSE); ++j)
    if (r[gridz][j] > y[3]) {
      gridr = j - 1;
      particle_found = TRUE;
    }
  if (particle_found == FALSE) {
    ydot[0] = 0.0;
    ydot[1] = 0.0;
    ydot[2] = 0.0;
    goto end_integration;
  }

  /* Set fluid conditions */
  /* This should be changed to interpolate values instead of reading them */
  fluid_velocity.r = ug[gridz][gridr].c / ug[gridz][gridr].a;
  fluid_velocity.theta = ug[gridz][gridr].e / ug[gridz][gridr].a;
  fluid_velocity.z = ug[gridz][gridr].b / ug[gridz][gridr].a;
  rho = ug[gridz][gridr].a;
  temperature = tg[gridz][gridr];
  viscosity = calculateViscosity(temperature);

  /* Define the particle_v vector */
  particle_v.r = y[0];
  particle_v.theta = y[1];
  particle_v.z = y[2];

  /* Ok; calculate drag forces */
  /* First, we need the flow Reynold's number */
  fluid_speed = magnitude(&fluid_velocity);

  relative_velocity.r = fluid_velocity.r - particle_v.r;
  relative_velocity.theta = fluid_velocity.theta - particle_v.theta;
  relative_velocity.z = fluid_velocity.z - particle_v.z;

  relative_speed = magnitude(&relative_velocity);
  Re = 2.0 * particle_radius * rho * relative_speed / viscosity;

  /* Now for the magnitude of the drag forces */
  if (Re > EPSILON)
    Cd = (24.0 / Re) + 6.0 / (1.0 + sqrt(Re)) + 0.4;
  else
    Cd = 0.0;
  drag_force = 0.5 * rho * SQ(relative_speed) * Cd * PI
    * particle_radius * particle_radius;

  scaleVector(&relative_velocity, 1.0/relative_speed, &relative_unit_vector);
  /* Then multiply by drag */
  scaleVector(&relative_unit_vector, drag_force, &drag);

  if (y[3] > EPSILON) {
```

```
    /* Particle Forces are added here */
    ydot[0] = SQ(y[1])/y[3] + drag.r/particle_mass -
      (4.0/3.0)*PI*SQ(particle_radius)*particle_radius*SQ(y[1])*
ug[gridz][gridr].a / (particle_mass * y[3]);
    ydot[1] = drag.theta / particle_mass -  y[0]*y[1]/y[3];
  }
  else {
    ydot[0] = drag.r / particle_mass;
    ydot[1] = 0.0;
  }
  ydot[2] = drag.z / particle_mass;

  if (ydot[2] < 0.0)
    ydot[2] = ydot[2];

 end_integration:
  ydot[3] = y[0];
  if (y[3] > EPSILON)
    ydot[4] = y[1] / y[3];
  else
    ydot[4] = 0.0;
  ydot[5] = y[2];
}

void jacobian()
     /* This routine is called from dlsode */
     /* It's a dummy routine;  it doesn't do anything */
{
  int j;
  j = 0;
}
```

```
/* io.c */
/* Contains input/output routines for particle */
/* Under construction 6/15/92 */
/* Written by David Oh */

#include <math.h>
#include <stdio.h>
#include "particle.h"
#include "globals.h"

getflow(fname)
/* This routine loads the flow data; works as of 6/16/92 */
     char fname[];
{
  char statstring[40], ugstring[40];
  FILE *statfile, *ugfile;
  int i, j;                                   /* scrap integer */
  double scrap;                               /* scrap double */

  /* Construct full file names */
  sprintf(statstring, "%s/%s.stat", FLOWDIRECTORY, fname);
  sprintf(ugstring, "%s/%s.ug", FLOWDIRECTORY, fname);

  statfile =  fopen(statstring, "r");
  ugfile = fopen (ugstring, "r");

  if (statfile == NULL || ugfile == NULL) {
    printf("\n\nUnable to open data files.");
    exit();
  }

  fscanf(statfile, "%d %d", &nxi, &neta);   /* reads number of grid cells */
  fscanf(statfile, "%d", &iter);
  fscanf(statfile, "%lf", &time);
  fscanf(statfile, "%lf", &scrap);  /* This line is not needed */
  fscanf(statfile, "%lf %lf", &mdot, &ftheta);

  /* Read State Vectors */
  /* Uses file ug */
  for (j = 0; j < neta; ++j){
    for (i = 0; i < nxi; ++i) {
      fscanf(ugfile, "%lf %lf %lf %lf %lf", &(ug[i][j].a), &(ug[i][j].b),
     &(ug[i][j].c), &(ug[i][j].d), &(ug[i][j].e));
      fscanf(ugfile, "%lf", &(pg[i][j]));
    }
  }

  /* Calculate gas temperature */
  for (j = 0; j < neta; ++j) {
    for (i = 0; i < nxi; ++i) {
      tg[i][j]=pg[i][j]/(RGAS * ug[i][j].a);
    }
  }

  fclose (statfile);
  fclose (ugfile);
}

getgrid(gridname)
/* This routine loads the grid; works as of 6/16/92 */
char gridname[];
{
  char gridstring[40];
  FILE *gridfile;
  int i, j;

  sprintf(gridstring, "grid.%s", gridname);
  gridfile = fopen (gridstring, "r");
  if (gridfile == NULL) {
    printf ("\nUnable to open grid file...");
    exit();
  }
  fscanf(gridfile, "%d %d", &nxi, &neta);

  /* This routine in memory.c */
  allocatememory();

  /* Read grid file */
  for (j = 0; j < neta; ++j)
    for (i = 0; i < nxi; ++i)
      fscanf(gridfile, "%lf %lf", &z[i][j], &r[i][j]);
}

readCmdLine(argc, argv, dataname, fname, gridname, particle_density,
    particle_radius)
     /* This is a command line parser.  It reads the unix commandline */
     /* This format of the line is: */
     /* particle [fluidfile] -G [gridfilename] -D [datafilename]
-d particle_density -r particle_radius */
     char *argv[], dataname[], fname[], gridname[];
     double *particle_density, *particle_radius;
     int argc;
{
  char *s;

  if (sscanf (*++argv, "%s", fname) == 0) {
    printf("Error reading Fluid File name.");
    readerror();
  }
  argc--;
  while (--argc > 0 && (*++argv)[0]=='-') {
    s = argv[0] + 1;
    switch (*s) {
    case 'D':
      if (sscanf (*++argv, "%s", dataname) == 0) {
printf("Error reading Data file name.");
readerror();
      }
      argc--;
      break;
    case 'G':
      if (sscanf (*++argv, "%s", gridname) == 0) {
printf("Error reading Grid file name.");
readerror();
      }
      argc--;
      break;
    case 'd':
      if (sscanf (*++argv, "%lf", particle_density) == 0) {
printf("Error reading Particle Density.");
```

```
readerror();
      }
      argc--;
      break;
    case 'r':
      if (sscanf (*++argv, "%lf", particle_radius) == 0) {
printf("Error reading Particle Radius.");
readerror();
      }
      argc--;
      break;
    }
  }
}

readerror()
{
  printf("\nError reading the command line.\n");
  exit();
}

saveData(dataname, eta, lend, length, particle_radius, send, sratio, trueeta)
char dataname[];
double **eta, length[], particle_radius, sratio[], **trueeta;
int lend, send;
/* Saves the analyzed data file as a text file */
/* Hopefully formatted for conversion to Mac Excel */
{
  char outputstring[50], outputstring2[50];
  FILE *output, *output2;
  int l, s;

  sprintf(outputstring, "%s/%s.%.0f.data", DATADIRECTORY, dataname,
                                  particle_radius/1e-6);
  sprintf(outputstring2, "%s/%s.%.0f.truedata", DATADIRECTORY, dataname,
  particle_radius/1e-6);

  output = fopen(outputstring, "w");
  output2 = fopen(outputstring2, "w");

  /* Write header */
  for (l = 0; l < lend && length[l] > (0.0 + EPSILON); ++l) {
    fprintf(output,"\t%.3f", length[l]);
    fprintf(output2, "\t%.3f", length[l]);
  }

  for (s = 0; s < send && sratio[s] >=  (0.0 - EPSILON); ++s) {
    fprintf(output, "\n%.3f\t", sratio[s]);
    fprintf(output2, "\n%.3f\t", sratio[s]);
    for (l = 0; l < lend && length[l] > 0.0; ++l) {
      fprintf(output, "%.3f\t", eta[l][s]);
      fprintf(output2, "%.3f\t", trueeta[l][s]);
    }
  }
  fclose(output);
  fclose(output2);
}


savePlot(fname, endstep, number_of_particles, particle_radius)
```



```
char fname[];
double particle_radius;
int number_of_particles, endstep[];
{
  char outputstring[50];
  FILE *output;
  int i, particle_count;
  double x, y, z;

  /* Ok; we have to loop thru the particles */
  for (particle_count=0;
       particle_count<number_of_particles;++particle_count){
    /* Open data file */
    sprintf(outputstring, "%s/%s.%.0f.%d", PLOTDIRECTORY, fname,
    particle_radius/1e-6, particle_count);
    output = fopen(outputstring, "w");
    if (output == NULL) {
      printf("\nUnable to open plot file.");
      exit();
    }

    /* Insert a key at the top of the file */
    fprintf(output, "#####\n");
    fprintf(output, "# Particle Track Gnuplot Data File\n");
    fprintf(output, "# X       Y        Z\n");
    fprintf(output, "#####\n");
    for (i = 0; i < endstep[particle_count]; ++i) {
      x = particle_position[particle_count][i].r *
       cos (particle_position[particle_count][i].theta);
      y = particle_position[particle_count][i].r *
       sin(particle_position[particle_count][i].theta);
      z = particle_position[particle_count][i].z;
      fprintf(output, "%f %f %f\n", x, y, z);
    }


    /* Different File Format */
/*    fprintf(output, "#####\n");
    fprintf(output, "# 2-d Gnuplot data file \n");
    fprintf(output, "# Run: %s\n", fname);
    fprintf(output, "# z         r\n");
    for (i = 0; i < endstep[particle_count]; ++i)
      fprintf(output, "%f     %f\n", particle_position[particle_count][i].z,
      particle_position[particle_count][i].r);
*/
    fclose(output);
  }
}

saveTrack(endstep, fname, number_of_particles, step)
char fname[];
int endstep[], number_of_particles, step;
{
  char outputstring[50];
  FILE *output;
  int i, particle_count = 0;

  for (particle_count = 5; particle_count < 8; ++particle_count) {
    sprintf(outputstring, "Countdata/particle.%d.%s", particle_count, fname);
    output = fopen (outputstring, "w");
```

```
    if (output == NULL) {
      printf("\nUnable to open output file...");
      exit();
    }
    fprintf(output, "%d", endstep[particle_count]);
    for (i = 0; i < endstep[particle_count]; ++i) {
      fprintf(output, "\n%f", timerecord[i]);
      fprintf(output, "\n%f %f %f", particle_position[particle_count][i].r,
      particle_position[particle_count][i].theta,
      particle_position[particle_count][i].z);
      fprintf(output, "\n%f %f %f", particle_velocity[particle_count][i].r,
      particle_velocity[particle_count][i].theta,
      particle_velocity[particle_count][i].z);
    }
    fclose(output);
  }
}
```

```
/* memory.c */
/* Memory allocation routines for "particle"; written by bamf */
/* Under development 6/16/92 */
#include <math.h>
#include <stdio.h>
#include "particle.h"
#include "globals.h"

allocatememory()
/* This routine actually allocates all the memory */
/* NOTE THAT THERE IS NO ERROR CHECKING HERE; IF THE MACHINE RUNS OUT */
/* OF MEMORY THE PROGRAM CRASHES, NO ANDS IFS ORS or BUTS */
{
  int j;

  /* Allocate memory */
  pg = (double **) (calloc (nxi, sizeof(double *)));

  tg = (double **) (calloc (nxi, sizeof(double *)));
  r =  (double **) (calloc (nxi, sizeof(double *)));
  z =  (double **) (calloc (nxi, sizeof(double *)));
  ug = (vector5 **) (calloc (nxi, sizeof(vector5 *)));

  for (j = 0; j < nxi; ++j) {
    pg[j] = (double *) (calloc(neta, sizeof(double)));
    tg[j] = (double *) (calloc (neta, sizeof (double)));
    r[j] =  (double *) (calloc(neta, sizeof(double)));
    z[j] =  (double *) (calloc(neta, sizeof(double)));
    ug[j] = (vector5 *) (calloc(neta, sizeof(vector5)));
  }
}

allocateTrajectoryRecords(number_of_particles)
int number_of_particles;
/* This routine allocates the particle velocity, time, and position records */
{
  int number_of_steps;
  int j;

  number_of_steps = MAX_STEP;

  particle_position = (cylindrical_coord **) (calloc(number_of_particles,
 sizeof(cylindrical_coord *)));
  particle_velocity = (cylindrical_coord **) (calloc(number_of_particles,
 sizeof(cylindrical_coord *)));
  timerecord = (double **) (calloc (number_of_particles, sizeof(double *)));

  for (j = 0; j < number_of_particles; ++j) {
    particle_position[j] = (cylindrical_coord *) (calloc(number_of_steps,
 sizeof(cylindrical_coord)));
    particle_velocity[j] = (cylindrical_coord *) (calloc(number_of_steps,
 sizeof(cylindrical_coord)));
    timerecord[j] = (double *) (calloc(number_of_steps, sizeof(double)));
  }
}
```

```
/* Useful vector manipulation routines */
/* To use, you must have the cylindrical_coord data type defined */
/* In the program header file.                                   */
/* By bamf */
#include <math.h>
#include <stdio.h>
#include "particle.h"

/*#define SQ(X) ((X)*(X)) */

double dot(a, b)
/* I think this dot product formula is only valid for two similarly oriented */
/* Frames, so be careful using it for cylindrical coord */
cylindrical_coord *a, *b;
{
  double dot;

  dot = a->r * b->r + a->z * b->z + a->theta * b->theta;
  return(dot);
}

double magnitude(a)
/* Calculates the magnitude of a 3-d vector */
cylindrical_coord *a;
{
  double b;

  b = sqrt (SQ(a->r) + SQ(a->z) + SQ(a->theta));
  return(b);
}

scaleVector(a, b, c)
/* Multiplies vector a by scalar b and returns the result in c */
cylindrical_coord *a, *c;
double b;
{
  c->r = a->r * b;
  c->theta = a->theta * b;
  c->z = a->z * b;
}
```

```
/* This file contains analysis parameters for the program particle.c    */
/* It specifies the skimming ratios and cylinder lengths to be analyzed */
/* Negative lengths correspond to blank fields */
/* By bamf, 7/24/92 */

char alreadyfound[11];          /* Array of booleans; see algorithm for why */
double **eta, **trueeta;
double length[11],              /* Array of cylinder "cut off" lengths */
       sratio[11];              /* Array of skimming ratios */
int lstart;
int particle_caught[11][11];   /* Record of how many particles are caught */
                                /* Of form particle_caught[length][sratio] */
int lend = 11,
  send = 11;

/* Section Lengths */
length[0] = 0.001;
length[1] =  0.2;
length[2] =  0.4;
length[3] =  0.6;
length[4] =  0.8;
length[5] =  1.0;
length[6] =  0.0;
length[7] =  0.0;
length[8] =  0.0;
length[9] =  0.0;
length[10] =  0.0;

/* Skimming ratios */
/* Note that these are local values (i.e. no assumption is made that the */
/* flow is going down a cylinder;  it could be moving down a nozzle too  */
sratio[0] = 0.0;
sratio[1] =  0.02;
sratio[2] =  0.04;
sratio[3] =  0.06;
sratio[4] =  0.08;
sratio[5] =  0.10;
sratio[6] =  0.125;
sratio[7] =  0.15;
sratio[8] =  0.175;
sratio[9] =  0.20;
sratio[10] =  -1.0;

/* Allocate memory for the "eta" and "trueeta" arrays */
/* ("eta" is linear efficiency ratio, "trueeta" is the true efficiency */
eta = (double **) (calloc (lend, sizeof(double *)));
trueeta = (double **) (calloc (lend, sizeof(double *)));
     for (l = 0; l < lend; ++l) {
       eta[l] = (double *) (calloc (send, sizeof(double)));
       trueeta[l] = (double *) (calloc (send, sizeof(double)));
     }
```

```
/* This is the header file for particle.c */
/* Directories for reading and writing data */
#define DATADIRECTORY "Countdata2"
#define FLOWDIRECTORY "Flowdata2"
#define PLOTDIRECTORY "Plotdata"

/* These are flags for dlsode.  They designate the type of system and the */
/* Number of equations in it */
#define NONSTIFF 10
#define STIFF_USER_FULL_JACOBIAN 21
#define STIFF_AUTO_FULL_JACOBIAN 22
#define STIFF_USER_BANDED_JACOBIAN 24
#define STIFF_AUTO_BANDED_JACOBIAN 25
#define NUMBER_OF_EQUATIONS 6

/* Data type for State Vectors */
typedef struct {
  double x;
  double y;
} vector2;

typedef struct {
  double x;
  double y;
  double z;
} vector3;

typedef struct {
  double r;
  double z;
  double theta;
} cylindrical_coord;

typedef struct {
  double a;
  double b;
  double c;
  double d;
  double e;
} vector5;

/* Generic Constants */
#define EPSILON 1e-8
#define PI 3.14159265358979323846
#define FALSE 0
#define TRUE 1

/* Gas Characteristics */
#define RGAS 4157.3              /* J/kg-K */
#define GAMMA 1.4
#define GAS_MOLECULAR_WT 2.0

/* Default particle starting position */
#define DEFAULT_THETA 0.1
#define DEFAULT_R 0.1;

#define MAX_PARTICLE 300
#define MAX_STEP 400        /* Maximum number of time steps */

/* A useful Macro */
#define SQ(X) ((X)*(X))
```