NEW TREATMENT OF THE PULSAR EQUATION

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Michael Joseph Vidal

August 2014

ABSTRACT

In solving the pulsar equation, two methods have risen to the forefront, the CKF method (Contopoulos, Kazanas, and Fendt), and the TOTS method (Takamori, Okawa, Takamoto, and Suwa). Both methods are implemented by numerical relaxation, which creates problems at a singular surface known as the light cylinder. Furthermore, these methods give limited information about the problem. The CKF method will not tell you how singular an answer is, only what it will look like after iterative correction. The TOTS method, which foregoes iterative correction, has the potential to give more information, but it has only been tested once, and an extra physical quantity was unnecessarily restricted just to get the solution to converge.

We have replaced relaxation with Newton's method in the context of solving nonlinear equations. This technique is demonstrated by replicating the results of Michel 1973, Contopoulos et al. 1999, Takamori et al. 2012, Lovelace et al. 2006, and Contopoulos et al. 2014. These altered methods refine the original ideas and make clear exactly what place they have in searching for solutions.

We also introduce new investigative paths. We show how we can weed out solutions by revealing the singular behavior of a derivative, even if the function itself appears well-behaved. We also show how we can use a singular solution to generate a smooth solution by looking for smooth contour lines in a field of singular ones with the "lone contour method."

BIOGRAPHICAL SKETCH

Prior to university, Michael Joseph Vidal has always been a hobbyist programmer, independently learning coding through self-inspired pursuits, ranging from a Nintendo Game Boy emulator, to a Japanese Mosaic Puzzle solver, to a file archival experiment. His most recent project is the published "Physics Tutor," a mobile tutoring app for introductory physics problems available on Android.

In 2008, Michael entered Cornell University, where he would earn his Bachelor of Science. An Engineering Physics major, he has benefited from a diverse repertoire of study, with three undergraduate minors and coursework ranging from the Department of Mathematics to the College of Veterinary Medicine. While earning his bachelor's degree, Michael performed supersolid research under Professor John D. Reppy, where he assisted with the machining of a liquid helium torsional oscillator, along with data acquisition and analysis. Michael also served as a tutor for the College of Engineering, where he helped his peers through one-on-one instruction.

In 2012, Michael entered Cornell University's Applied Physics graduate program, where he began his research with Professor Richard V. E. Lovelace. Combining undergraduate and graduate programming instruction, Michael explored a new computer simulation method which enabled further progress in the investigation of neutron star pulsars.

Dedicated to my loving family.

Dedicated to Kimberly Jeanne Beccia and Jennifer Alexis Davis.

Dedicated to my professor and advisor, Richard V. E. Lovelace.

ACKNOWLEDGEMENTS

I would like to acknowledge Earl J. Kirkland for his programming advice and instruction which helped make this research possible.

I would like to acknowledge the School of Applied and Engineering Physics, the Department of Materials Science and Engineering, and the Department of Mathematics for exemplary education and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In the study of rotating neutron stars, or pulsars, there is much interest in the electromagnetic fields and particle wind that emanate from the surface. One key result is the Grad-Shafranov equation of [1], which links the star's rotation to the magnetic flux generated:

$$\left[1 - \left(\frac{r\Omega_*(\psi)}{c}\right)^2\right] \Delta^\star \psi - \frac{2r\Omega_*(\psi)^2}{c^2}\frac{\partial \psi}{\partial r} = -\tilde{F}(\psi) \tag{1.1}$$

Where:

$$\begin{aligned} \tilde{F}(\psi) &= \tilde{H}(\psi)\frac{d\tilde{H}(\psi)}{d\psi} \\ \Delta^\star &= \partial^2/\partial r^2 - (1/r)(\partial/\partial r) + \partial^2/\partial z^2 \end{aligned}$$

$\psi \equiv \psi(r,z)$ is the nonnegative magnetic flux, $\Omega_*(\psi)$ is the angular velocity of the star, c is the speed of light, and $\tilde{H}(\psi)$ is proportional to the current in the poloidal direction. The coordinates are cylindrical, except that by symmetry the angle is removed, leaving r and z. Solving for ψ would in turn allow us to calculate the electric and magnetic fields.

The coefficients of the highest derivative terms simultaneously vanish when $r = |c/\Omega_*(\psi)|$, a location known as the light cylinder. It is well known that many "solutions" returned by simulations have kinks or are otherwise singular at this location. To study this peculiarity, this equation has been simplified further. First, the angular velocity is assumed to be a constant ($\Omega_*(\psi) \longrightarrow \Omega_*$). Then, the following substitutions are used:

$$
\begin{aligned}
R &= \left(\frac{\Omega_*}{c}\right) r \\
Z &= \left(\frac{\Omega_*}{c}\right) z \\
H(\psi) &= \left(\frac{\Omega_*}{c}\right) \tilde{H}(\psi) \\
F(\psi) &= \left(\frac{\Omega_*}{c}\right)^2 \tilde{F}(\psi)
\end{aligned}
$$

Along with the definition of a linear operator:

$$\hat{L} \equiv (1 - R^2)\left(\frac{\partial^2}{\partial R^2} + \frac{\partial^2}{\partial Z^2}\right) - \left(\frac{1 + R^2}{R}\right)\frac{\partial}{\partial R}$$

The result is the so-called pulsar equation:

$$\hat{L}(\psi) = -F(\psi) \tag{1.2}$$

Where the light cylinder now occurs at $R = 1$.

Note that a solution which does exist everywhere will obey the following "smoothness condition" at $R = 1$:

$$\frac{\partial \psi}{\partial R} = \frac{1}{2} F(\psi) \tag{1.3}$$

This is not really a separate piece of information for a true solution, because it is just the pulsar equation with $R = 1$ plugged in. However, many authors single this point out because it is a useful criteria for weeding out potential solutions. Note that in theory, the smoothness condition is enough as is. However, we will show a case where the first derivative seems to behave, and the second derivative is instead used to rule out the solution. Indeed, using the smoothness condition, we show that often, if the solution exists at the light cylinder at all, then many derivatives with respect to R will exist on that surface too, not just the first (see Appendix D).

We also need to specify boundary conditions. For our purposes, the domain is the set of points where R and Z are both nonnegative. The first boundary is the positive Z-axis, where the value of ψ is agreed upon as zero. Boundary conditions for other edges are not always agreed upon, so throughout this paper various choices will be demonstrated.

Another feature of the pulsar equation is $F(\psi)$, which is a function of the unspecified $H(\psi)$. It is agreed upon that $H(0) = 0$, and it is often proposed that there exists some positive value ψ_{eq} (also called ψ_{open} by some authors) such that $H(\psi) = 0$ if $\psi \geq \psi_{eq}$. In most cases this produces closed loop contours. However, for $0 < \psi < \psi_{eq}$ we will in general have a nonlinear differential equation with an unspecified right-hand side.

Also, in looking at the pulsar equation, say that a solution ψ exists, so that:

$$\hat{L}(\psi) = -F(\psi) \tag{1.4}$$

We desire that for any constant K, $K\psi$ is also a solution. But this means:

$$-F(K\psi) = \hat{L}(K\psi) = K\hat{L}(\psi) = -KF(\psi) \tag{1.5}$$

This forces $F(K\psi) = KF(\psi)$. Seeing as how $F(\psi)$ can be nonlinear, this is not a trivial requirement. Most authors consider cases where this constant is just ψ_{eq} itself and thus accomplish these requirements by fixing:

$$H(\psi) = \psi f(\frac{\psi}{\psi_{eq}}) \tag{1.6}$$

$$F(\psi) = \psi f(\frac{\psi}{\psi_{eq}}) \left(f(\frac{\psi}{\psi_{eq}}) + \frac{\psi}{\psi_{eq}} f'(\frac{\psi}{\psi_{eq}}) \right) \tag{1.7}$$

Where f is a differentiable function. Note that this form of $H(\psi)$ is sufficient, but not necessary to fulfill the requirement in $F(\psi)$'s behavior. [1]

Even with a fixed boundary, this leaves room for debate as to which possibilities for $F(\psi)$ have physical solutions. One example of a debate is whether or not there exists a physical solution without an equatorial current sheet for a given boundary choice. This can be seen by looking at the behavior of $H(\psi)$:

$$\begin{aligned} \lim_{\psi \to \psi_{eq}^-} H(\psi) &= 0 \longleftrightarrow \text{“There is no equatorial current sheet.”} \\ \lim_{\psi \to \psi_{eq}^-} H(\psi) &\neq 0 \longleftrightarrow \text{“There exists an equatorial current sheet.”} \end{aligned}$$

Since $H(\psi) = 0$ for $\psi \geq \psi_{eq}$, the current sheet would indicate a discontinuous jump in $H(\psi)$.[2]

Moreover, as we will see, how to specify $F(\psi)$ is the *only* essential difference between existing numerical methods, and (along with the boundary) is the determinant of whether or not a smooth solution exists. So while ψ is the quantity we ultimately want, $F(\psi)$ is equally as important to consider.

As far as solving the equation itself, no matter how $F(\psi)$ is specified, if we restrict ourselves to $0 \leq R < 1$, then numerical relaxation is a viable method. However, doing this method "as is" will not be able to handle potential singularities at $R = 1$. There have been two alterations to deal with this problem. [2] splits the domain in the CKF method, whereas [3] splits the pulsar equation in the TOTS method. Both methods still use relaxation.

[1] We point out that knowing $H(\psi)$ defines $F(\psi)$ uniquely, and because we take $H(\psi)$ to be a nonpositive quantity, knowing $F(\psi)$ also defines $H(\psi)$ uniquely:
$H(\psi) = -\sqrt{2\int_0^\psi F(\psi')d\psi'}$

[2] If there is no equatorial current sheet, then $F(\psi_{eq}) = H(\psi_{eq}) = 0$. However, as pointed out in [2], if there is an equatorial current sheet, then $F(\psi_{eq})$ is a Dirac delta function and $H(\psi_{eq})$ is undefined. For our purposes, we can just say $F(\psi_{eq}) = H(\psi_{eq}) = 0$ either way.

In this paper we propose a new option: switching relaxation with Newton's method. While the abstract ideas of the older methods remain, using Newton's method provides a much more straightforward process conceptually, and eliminates the need for domain splitting, equation splitting, and some extraneous parameters.[3]

In Chapter 2, we remind the reader of the ideas of the older methods. In Chapter 3, we explain how to adapt Newton's method to this problem, being mindful of the boundaries and the light cylinder. We define the "Altered CKF method" and "Altered TOTS method" here. In Chapter 4 we demonstrate the altered methods with five previously investigated cases. In Chapter 5 we provide additional comments on the TOTS simulations of [3]. We will see how altering their method makes it much simpler, providing an opportunity to revisit their solutions. This chapter also serves as a warning, that although their method works, one must be cautious about what exactly that means, and what one should and should not expect from an answer. In Chapter 6 we provide additional comments on the jets simulations of [4]. We show how, with the help of the Altered TOTS method, a new investigative tool could possibly help derive functional forms for solutions starting from a singular answer.

[3]The altered methods do *not* end up merging completely. The treatment of $F(\psi)$ remains a fundamental point of difference.

CHAPTER 2

PREVIOUS METHODS

We give a very simplified summary of the two existing methods. For specific details about the methods, refer to [2] and [3].

2.1 CKF Method

1. Initialize ψ and $F(\psi)$ to values that favor numerical convergence, and split the domain into three regions: inside, outside, and on the light cylinder.[1]

2. Apply an iteration of numerical relaxation on the pulsar equation both inside and outside the light cylinder separately. The light cylinder serves as a boundary edge for both regions.

3. Correct the distribution of $F(\psi)$ as follows:
a) Determine $F(\psi)$ on the light cylinder by the smoothness condition (Equation 1.3). Also set ψ on the light cylinder as the average of the values to the immediate left and right.

b) For points elsewhere, because $F(\psi)$ only depends on ψ, $F(\psi)$ must be the same value everywhere on a field line, *including* the point where it crosses the light cylinder. Consider that on the light cylinder, $F(\psi)$ is known from a), so we can use this knowledge to set $F(\psi)$ everywhere along each field line.

4. Repeat steps 2 and 3 until convergence.

[1]Personal observations suggest that the light cylinder should lie exactly on the grid for all simulation methods.

2.2 TOTS Method

1. Split the pulsar equation into two equations:

Ampere's law:

$$\frac{\partial^2\psi}{\partial R^2} + \frac{\partial^2\psi}{\partial Z^2} - \frac{1}{R}\frac{\partial\psi}{\partial R} = -ST(R,Z) \tag{2.1}$$

Force-free condition:

$$R^2\left(\frac{\partial^2\psi}{\partial R^2} + \frac{\partial^2\psi}{\partial Z^2} - \frac{1}{R}\frac{\partial\psi}{\partial R}\right) + 2R\frac{\partial\psi}{\partial R} - F(\psi) = -ST(R,Z) \tag{2.2}$$

Note the new unknown term $ST(R,Z)$, which is the toroidal current.

2. Fix a functional form for $H(\psi)$ (and thus $F(\psi)$) and initialize ψ and $ST(R,Z)$ to values that favor numerical convergence (in this method, $F(\psi)$ is known, but ψ and $ST(R,Z)$ are unknown, so we still have two unknowns).

3. Apply an iteration of numerical relaxation on Ampere's law over the entire region (straight through the light cylinder).

4. Update $ST(R,Z)$ values (refer to [3]). Note that this is where the Force-free condition comes into play.

5. Repeat steps 3 and 4 until convergence.

CHAPTER 3

ALTERED METHODS

Both previous methods use different ideas to deal with the light cylinder, and both are implemented with the help of numerical relaxation. We believe that a benefit can be found by replacing their implementation with Newton's method and nonlinear equation solving.

3.1 Adapting Newton's Method

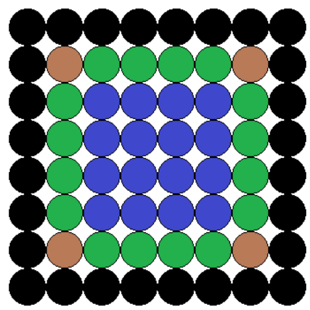
We remind the reader of the steps of Newton's method, with the context of using CKF's and TOTS's ideas. Similar to relaxation, we take the pulsar equation and convert it to a discrete version. For now, we consider $F(\psi) = 0$. We will consider the general case with $F(\psi) \neq 0$ later. The pulsar equation then becomes:

$$a_1 U_{j-1,k} + a_2 U_{j+1,k} + a_3 U_{j,k-1} + a_4 U_{j,k+1} + a_5 U_{j,k} = a_6$$

$$\begin{aligned}
a_1 &= 1 - j^2(\Delta R)^2 + \frac{1}{j} + j(\Delta R)^2 \\
a_2 &= 1 - j^2(\Delta R)^2 \\
a_3 &= \left(\frac{\Delta R}{\Delta Z}\right)^2 (1 - j^2(\Delta R)^2) \\
a_4 &= \left(\frac{\Delta R}{\Delta Z}\right)^2 (1 - j^2(\Delta R)^2) \\
a_5 &= \left[-2 - 2\left(\frac{\Delta R}{\Delta Z}\right)^2\right] (1 - j^2(\Delta R)^2) - \frac{1}{j} - j(\Delta R)^2 \\
a_6 &= 0
\end{aligned} \tag{3.1}$$

Where we have chosen to use backwards difference for the first derivative and central difference for the second derivatives. One is welcome to use different choices, and in particular, because we do not have to solve for $U_{j,k}$ directly in the equation we make, it is *not* required to have a_5 nonzero everywhere, as in relaxation. ΔR and ΔZ are the desired grid spacings. $R = j\Delta R$, $Z = k\Delta Z$, and discretization comes from indexing j and k as nonnegative integers.

Now, to demonstrate how to deal with boundaries, consider the following grid:



Boundary points are black.

For all interior points not touching a boundary ("interior interior" points, labeled blue), we can simply write an equation for that point in terms of itself and the surrounding points using Equation 3.1.

For all interior points adjacent to one boundary (edge points, labeled green), we write one equation for the interior point using Equation 3.1, and one equation for the adjacent boundary point of the form:

$$b_1 U_{j-1,k} + b_2 U_{j+1,k} + b_3 U_{j,k-1} + b_4 U_{j,k+1} + b_5 U_{j,k} = b_6 \tag{3.2}$$

The boundary equation is solved for that boundary point, and plugged into the interior point's equation. For example, listing the coefficients as a tuple

$(b_1, b_2, b_3, b_4, b_5, b_6)$, consider some typical bottom boundary conditions:

$$\psi(R_0, Z_0) = f_0 \longrightarrow (0, 0, 1, 0, 0, f_0)$$
$$\frac{\partial \psi}{\partial Z}|_{(R_0,Z_0)} = f_0 \longrightarrow (0, 0, -1, 0, 1, f_0 \Delta Z)$$

Where f_0 is a constant, and (R_0, Z_0) is a point on the bottom boundary.

For all interior points touching two boundaries (corner points, labeled brown), both boundary points are eliminated by boundary-derived equations.[1]

Thus far we have not made any accommodation for the light cylinder (other than placing the light cylinder exactly on the grid). But it is generally desired that the solution be smooth over the light cylinder. If N_{LC} is the grid number of the light cylinder, then consider the following overwrite to Equation 3.1:

If $N_{LC} - 1 \leq j \leq N_{LC} + 1$

Then $(a_1, a_2, a_3, a_4, a_5, a_6) = (-1/2, -1/2, 0, 0, 1, 0)$

This makes the solution smooth over the light cylinder.[2]

Note that for an N_RxN_Z grid, this amounts to solving $(N_R - 2)$x$(N_Z - 2)$ equations for $(N_R - 2)$x$(N_Z - 2)$ unknowns. Let $S = (N_R - 2)$x$(N_Z - 2)$, and let the variables be given by v_m with $0 \leq m \leq S - 1$. Note that we have a linear system of S equations for S variables. For convenience, let A and B be the coefficient matrix and right hand side vector for that system.

Now, these equations are linear, but we still need to include $F(\psi)$, which in general will not be linear. Reconsidering $F(\psi)$, one way to write each equation

[1] See Appendix A for more details about the boundaries.

[2] See Appendix B for more details about the need (or lack or need) for smoothing over the light cylinder.

is:[3]

$$eq_m : \sum_{n=0}^{S-1} A_{m,n} v_n - B_m + F_m = 0 \tag{3.3}$$

Note that $F(\psi)$ is a function of ψ only, so for any grid point represented by the variable v_m, we write $F_m \equiv F(v_m)$.

Now, for Newton's method, we need the Jacobian matrix. Fortunately, it is easy to write the Jacobian analytically, with elements:

$$J_{m,n} = \frac{\partial eq_m}{\partial v_n} = A_{m,n} + \delta_{m,n} \frac{dF_m}{dv_n} \tag{3.4}$$

In the CKF method, the derivative of F_m can be found by the chain rule, whereas in the TOTS method, the derivative of F_m can be found using the guessed formula of $F(\psi)$. We will see later that this term ends up being unimportant. For now, just note that the off-diagonal elements of J are fixed. Only the diagonal ones ever change.

One straightforward (but by no means the most efficient) way to perform one iteration of Newton's method is as follows:

[3] Even if $F(\psi)$ is linear or has a linear term, here we keep $F(\psi)$ entirely separate.

1. Update diagonal elements of J.

2. Update J inverse (J^{-1}).

3. Update all eq_m.

4. For each variable v_m, perform the following algorithm:

Input: v_{old}, the existing value of that variable.

Output: v_{new}, the new value of that variable.

$$\begin{aligned} t &:= \sum_{n=0}^{S-1} J^{-1}_{m,n} eq_n \\ t &:= v_{old} - t \\ v_{new} &:= (FRAC)(t) + (1 - FRAC)(v_{old}) \end{aligned}$$

Where FRAC is similar to the relaxation parameter. One key advantage we have noticed is that FRAC follows a very simple rule: Higher FRAC converges faster, and lower FRAC converges more carefully. The relaxation parameter, on the other hand, only vaguely followed that rule. There would often be exceptions, making it harder to work with.

Now, we take advantage of a major shortcut which makes the speed of Newton's method competitive with relaxation. Our observation is that the updates to J and J^{-1} do not play an important role. Thus, we can fix $J = A, J^{-1} = A^{-1}$ once. Also note that A only depends on the grid size, grid spacing, and choice of boundary conditions. Working with a fixed grid and boundary, one could store J^{-1} in memory, negating the need for steps 1 and 2 entirely.

Now, replacing numerical relaxation with Newton's method is straightforward. Consider these modified processes:

3.2 Altered CKF Method

1. Initialize ψ to 1 and $F(\psi)$ to 0 everywhere.[4]

2. Apply an iteration of Newton's method.

3. Correct the distribution of $F(\psi)$ as in the original CKF method (this step does not change).

4. Repeat steps 2 and 3 until convergence.

In our opinion, splitting the domain was an implementation detail, not a fundamental part of the CKF method. Their fundamental idea was to enforce the smoothness of ψ over the light cylinder, which we have ensured through the equations themselves. Thus, although we pay heed to the light cylinder in crafting the equations, once we perform Newton's method, light cylinder points are like any other.

It should be noted that when we demonstrate this altered method in Chapter 4, we deviate from the spirit of the original CKF method in two ways. First, CKF used a coordinate transformation to accommodate an infinite grid. To avoid this, we use a technique proposed by [4]. This allows us to deal with field lines that would cross the light cylinder, but cannot because the grid ends prematurely. Second, the original CKF method seeks solutions where ψ_{eq} is found iteratively, by solving inside the light cylinder and setting ψ_{eq} to whatever value is found in the lower right corner of the inside region for that iteration. However, we want to draw direct comparisons between CKF-like solutions and other so-

[4]This particular choice for initialization was chosen for convenience. The only real requirement is that the method converge to a solution where ψ is positive everywhere, making this a sensible choice.

lutions which take ψ_{eq} as an adjustable parameter, so we forego this requirement here and also take ψ_{eq} as an adjustable parameter.

If desired, one can work with CKF's original requirements. The coordinate transformation would simply amount to using a transformed set of equations and boundary conditions which are already laid out in [2]. As for ψ_{eq}, one could simply modify the equations for any affected boundary points to be equal to the appropriate corner point (this boundary does not strictly fit the form of the boundary equations we laid out prior, but the idea is still easy to implement. See Appendix A).

3.3 Altered TOTS Method

1. Fix a functional form for $H(\psi)$ (and thus $F(\psi)$).

2. Initialize ψ to 1 everywhere.

3. Keep applying iterations of Newton's method until convergence.

The TOTS method becomes much more direct. There is no need to split the pulsar equation or introduce the toroidal current as is done in [3]. Our belief is that these were again simply implementation details, and not fundamental parts of their method.

Also note that with domain splitting and toroidal currents aside, both CKF's and TOTS's ideas are more similar than one would believe looking at the steps of their original methods. The one fundamental difference in how they approach the problem is evident: one iterates to find $F(\psi)$, one sets $F(\psi)$ directly.

CHAPTER 4

DEMONSTRATION OF PRIOR RESULTS

To prove the usefulness of these altered methods, we will demonstrate the replications of five previously found results (Monopole, CKF, TOTS, Jets, and Null Sheet).

It should be noted that we use the following boundary conditions:

Left Edge:

$$\psi(R_0, Z_0) = 0$$

Top and Right Edge:

$$R_0 \frac{\partial \psi}{\partial R}|_{(R_0,Z_0)} + Z_0 \frac{\partial \psi}{\partial Z}|_{(R_0,Z_0)} = 0$$

Bottom Edge:

For Monopole:

$$\psi(R_0, Z_0) = \psi_{eq}$$

For Null Sheet:

$$\begin{aligned} \frac{\partial \psi}{\partial Z}|_{(R_0,Z_0)} &= 0 && R_0 \leq 1 \\ \frac{\partial \psi}{\partial Z}|_{(R_0,Z_0)} &= \frac{H(\psi(R_0, Z_0))}{(R_0^2 - 1)^{1/2}} && R_0 > 1 \end{aligned}$$

All other cases:

$$\begin{aligned} \frac{\partial \psi}{\partial Z}|_{(R_0,Z_0)} &= 0 && R_0 \leq 1 \\ \psi(R_0, Z_0) &= \psi_{eq} && R_0 > 1 \end{aligned} \tag{4.1}$$

Where ψ_{eq} is specified directly. $F(0) = 0$ and $F(\psi) = 0$ if $\psi \geq \psi_{eq}$. For $0 < \psi < \psi_{eq}$, the Altered CKF method finds $F(\psi)$ iteratively, but when using the Altered TOTS method, we specify $F(\psi)$:

Monopole:

$$\begin{aligned} H(\psi) &= \psi(\frac{\psi}{\psi_{eq}} - 2) \\ F(\psi) &= 2\psi(\frac{\psi}{\psi_{eq}} - 1)(\frac{\psi}{\psi_{eq}} - 2) \end{aligned}$$

TOTS:

$$\begin{aligned} H(\psi) &= \psi(\frac{\psi}{\psi_{eq}} - 1) \\ F(\psi) &= 2\psi(\frac{\psi}{\psi_{eq}} - 1)(\frac{\psi}{\psi_{eq}} - \frac{1}{2}) \end{aligned}$$

Jets:

$$\begin{aligned} H(\psi) &= \frac{k_H}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 2) \\ F(\psi) &= \frac{k_H^2}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 1)(\beta\frac{\psi}{\psi_{eq}} - 2) \end{aligned}$$

Null Sheet:

$$\begin{aligned} H(\psi) &= 1.07\psi\left(2 - \frac{\psi}{\psi_{eq}}\right)\left(1 - \frac{\psi}{\psi_{eq}}\right)^{0.4} \\ F(\psi) &= \frac{2}{5}(1.07)^2\psi\left[6\left(\frac{\psi}{\psi_{eq}}\right)^2 - 12\frac{\psi}{\psi_{eq}} + 5\right]\left(2 - \frac{\psi}{\psi_{eq}}\right)\left(1 - \frac{\psi}{\psi_{eq}}\right)^{-0.2} \end{aligned} \tag{4.2}$$

Note that we are not always using the outer boundary conditions originally considered by the authors. However, it is generally believed that the particular boundary conditions far away do not matter, and the ones for the left and bottom edge are agreed upon. For cases with a star, it is inserted as a 2x2 box in the lower left corner:

Jets:

$$\psi_{star} = \frac{1}{(R^2 + Z^2)^{\frac{1}{2}}}$$

All other cases:

$$\psi_{star} = \frac{R^2}{(R^2 + Z^2)^{\frac{3}{2}}} \tag{4.3}$$

Also, all simulations in this section are done with a grid spacing of 0.05 in both directions, and most simulations use a 40x40 grid (so they cover $0 \leq R, Z \leq 2$).[1]

[1]Note that it is not hard to enter in different boundary conditions if desired. Also everything said here assumes a domain with only nonnegative Z. These decisions need to be altered if one wanted to use all points in the closed half-plane $R \geq 0$ in a simulation.

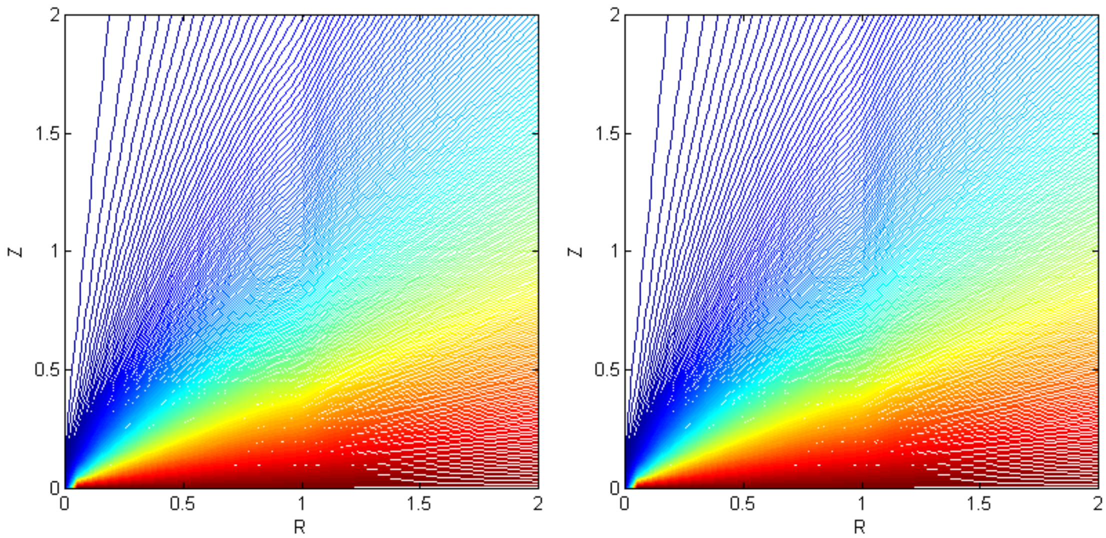


Figure 4.1: The Michel Monopole with $\psi_{eq} = 1$ found using both altered methods (CKF on left, TOTS on right).

4.1 Monopole

From [5], the monopole solution is reproduced with both altered methods in Figure 4.1.

4.2 CKF

From [2], and as seen in [4] and [6] (not an exhaustive list), reproduced using Altered CKF.

Figure 4.2 shows a more detailed version of a case in [2]. Figure 4.3 shows a variety of CKF solutions. There are three regions of ψ_{eq} of interest. Low ψ_{eq} has solutions mostly featuring closed contour loops, intermediate ψ_{eq} has "typical"

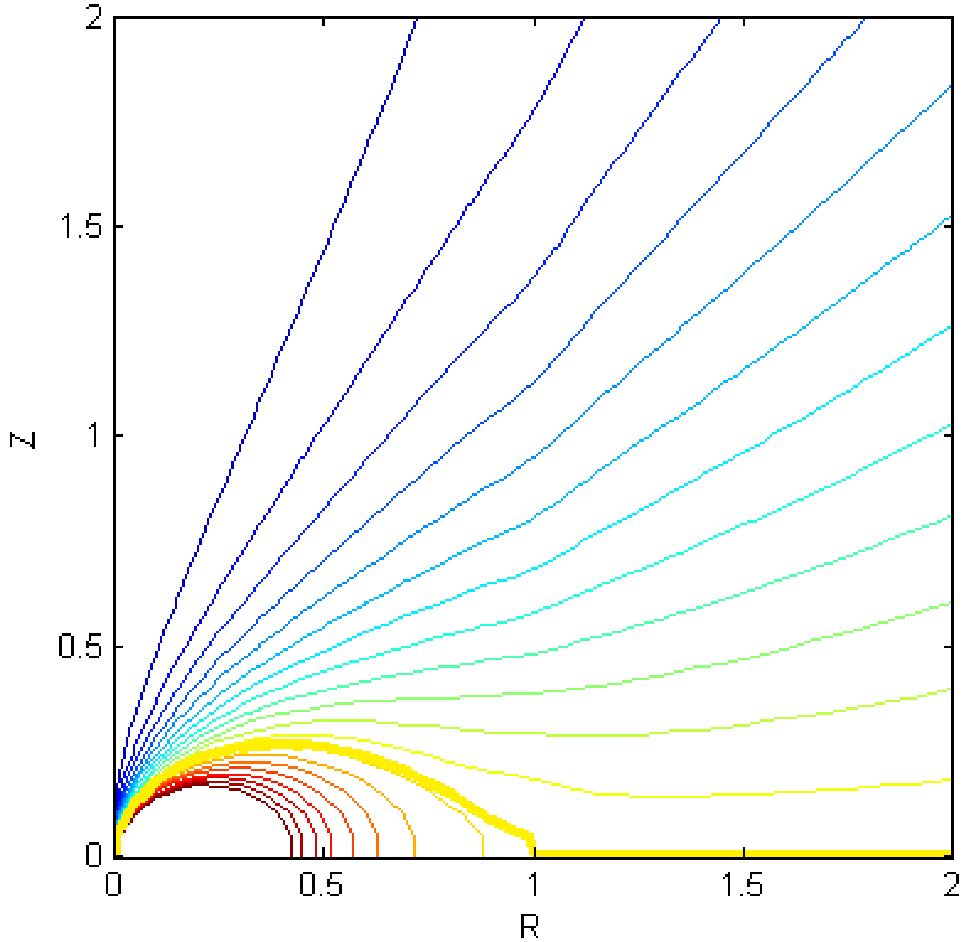


Figure 4.2: Direct replica of the result of [2] (obtained by setting $\psi_{eq} = 1.28$) which can be compared directly to their Figure 3.

CKF solutions, and high ψ_{eq} shows an introduction of field lines which do not emanate from the star at all.

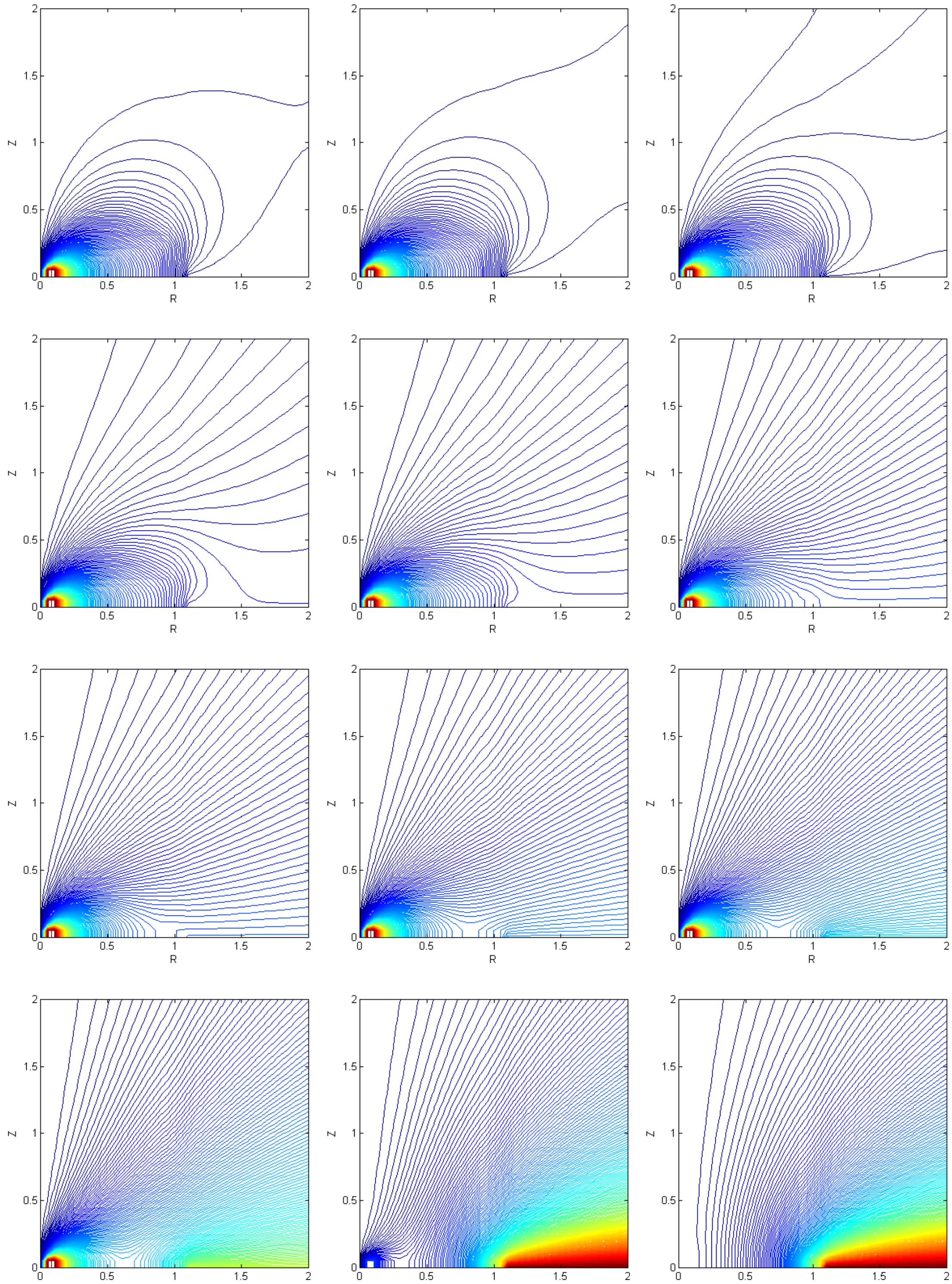


Figure 4.3: CKF solutions found with the Altered CKF method.
$\psi_{eq} = 0.001, 0.01, 0.1, 0.5, 0.8, 1.2, 1.4, 1.8, 2.4, 4.0, 40.0, 5000.0$

4.3 TOTS

From [3]. As previously stated, the Altered TOTS method does not need to introduce the toroidal current, so some parameters mentioned in their original paper are unnecessary.

Figure 4.4 shows a more detailed version of a case in [3]. Figure 4.5 shows a variety of TOTS solutions. From their equations, $F(\psi) = A^2\psi(\psi - \psi_{ret})(\psi - \psi_{eq})$, $A^2 = \frac{1}{r\psi_{eq}^2}$, and we used a fixed $r \equiv \frac{\psi_{ret}}{\psi_{eq}} = 0.5$, although other values $0.5 \leq r \leq 1.0$ they considered could be used without difficulty.

When using the same ψ_{eq} value, the TOTS solution generally looks like the CKF solution, but with a ripple near the light cylinder. In Chapter 5 we provide more analysis on this point.

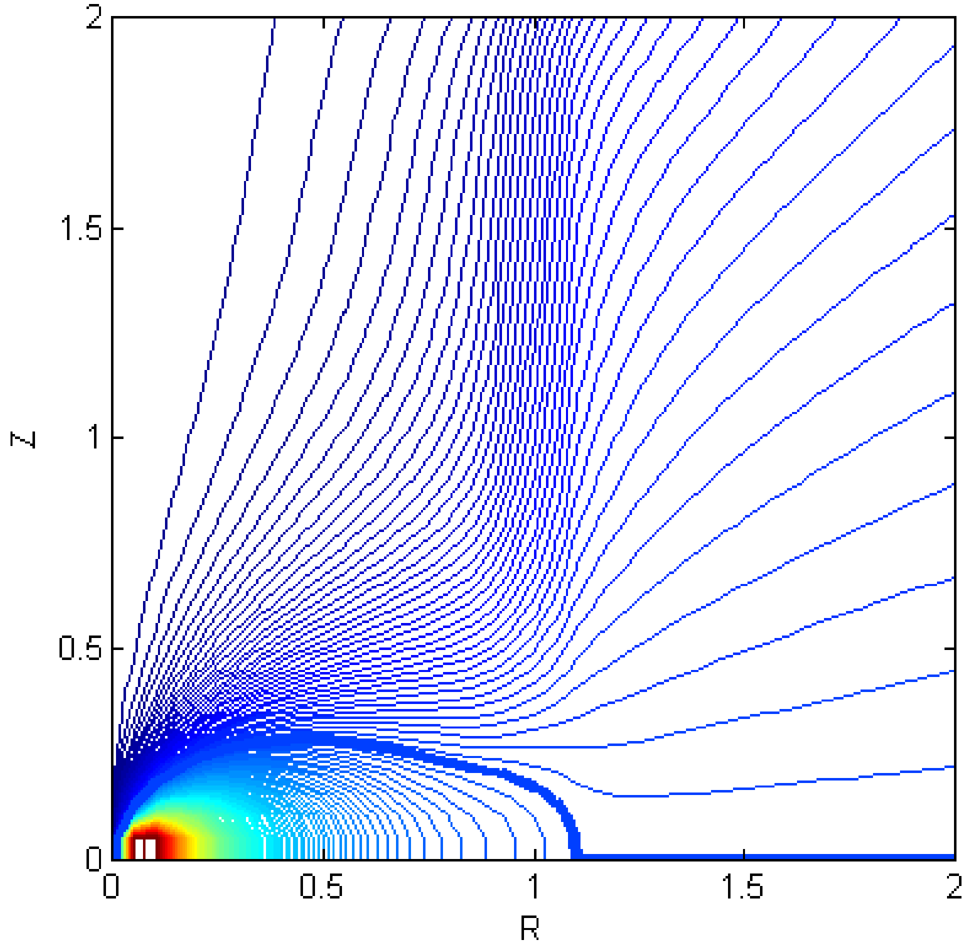


Figure 4.4: Direct replica of the result of [3] (r=0.5) with the same $\psi_{eq} = 1.225$ which can be compared directly to their Figure 3.

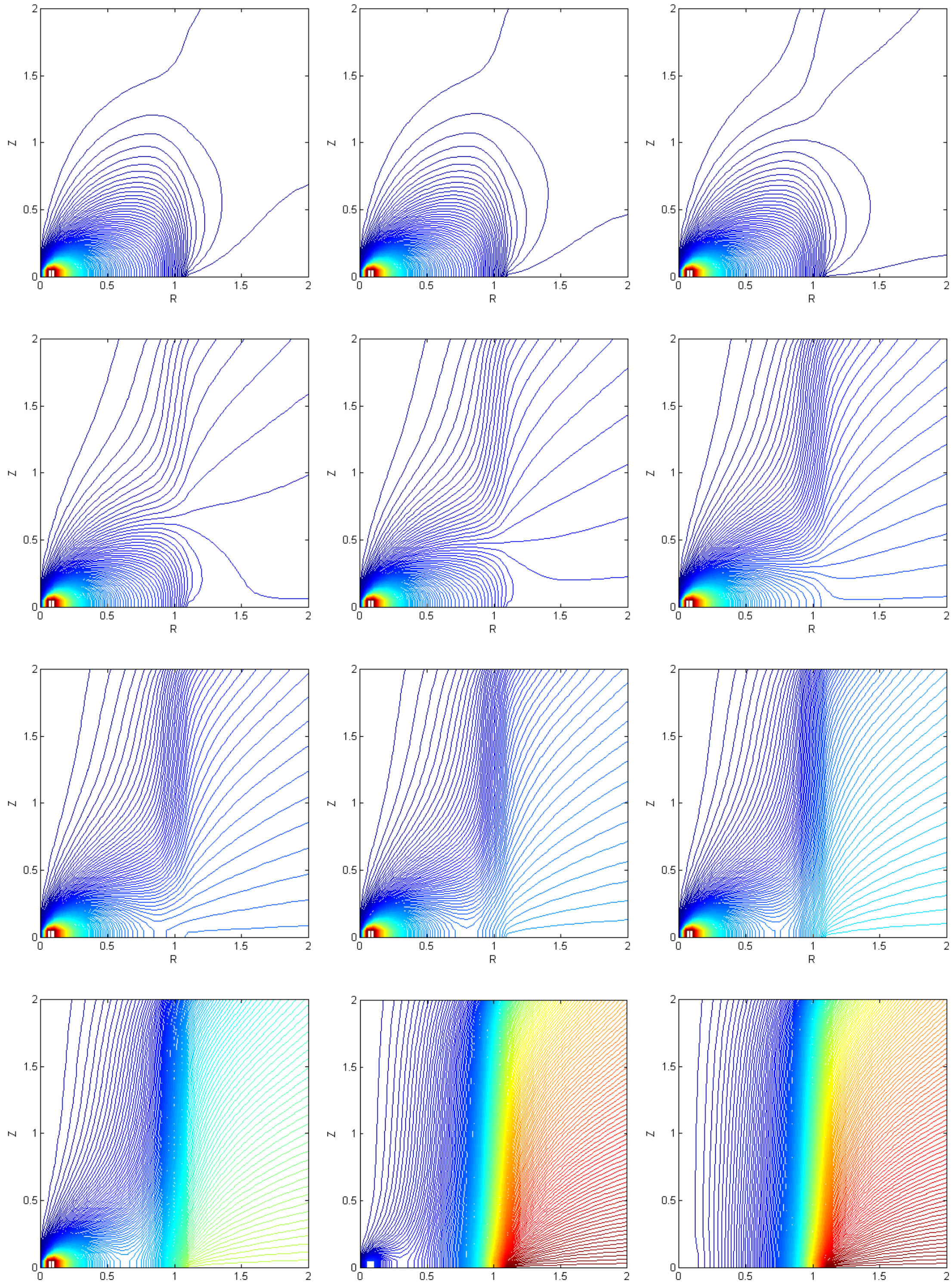


Figure 4.5: TOTS solutions found with the Altered TOTS method.
$\psi_{eq} = 0.001, 0.01, 0.1, 0.5, 0.8, 1.2, 1.4, 1.8, 2.4, 4.0, 40.0, 5000.0$

4.4 Jets

From [4] is a solution with collimated jets along the Z-axis. They use the CKF method, but for field lines that do not cross the light cylinder, they use:

$$H(\psi) = \frac{k_H}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 2) \tag{4.4}$$

$$F(\psi) = \frac{k_H^2}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 1)(\beta\frac{\psi}{\psi_{eq}} - 2) \tag{4.5}$$

Where k_H is an adjustable parameter, and β is determined iteratively using:

$$\beta = \frac{1}{2}\left\{3 - \left[1 + \frac{8\psi_c F_c}{k_H^2}\right]^{\frac{1}{2}}\right\} \tag{4.6}$$

ψ_c and F_c are the values of ψ and $F(\psi)$ at the upper right corner of the region inside the light cylinder. We note that β always ended up negligibly different from 1 in these particular simulations.

There was much difficulty in the numerical stability of such solutions when using relaxation. Using the Altered CKF method seems to alleviate these problems. To demonstrate this, in Figure 4.6 we show that one can modify both a monopole solution and a CKF-like solution by adding jets. An interesting feature we point out in Figure 4.7 is an "exclusion zone" where ψ approaches a constant value, consistent with an observation of [4].

If we take $\beta = 1$ for the time being, the form of $H(\psi)$ for jets is a generalization of the monopole $H(\psi)$ with a varying constant in front. Indeed (restricting ourselves inside the light cylinder) it is straightforward to show three types of curvature based on k_H in Figure 4.8. For $0 \leq |k_H| < 2$, jets curve to the right, for $|k_H| = 2$, we get the straight line monopole solution, and $|k_H| > 2$ we get jets that curve upwards. This would suggest that this jets idea could lead to a generaliza-

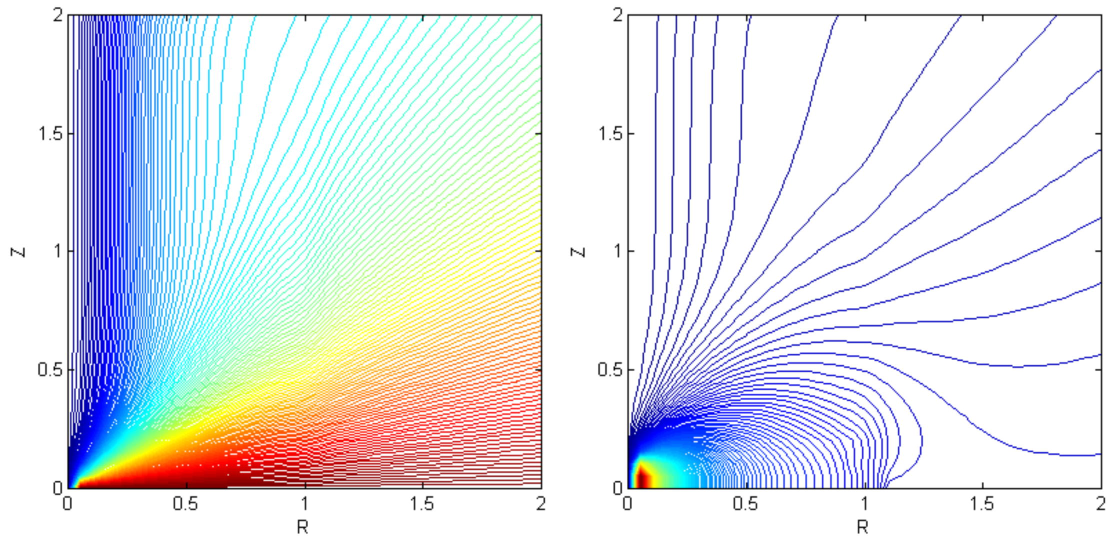


Figure 4.6: Jets solutions with $k_H = 5.48$ and $\psi_{eq} = 1$. Left is a modified monopole, right is a modified CKF-like solution.

tion of the original monopole solution. But of course, the monopole solution is smooth through the light cylinder, whereas contour lines that form a jet (curved in either direction) appear to never lead to a smooth solution. There is one possible loophole for the upward jets: perhaps there exists some ψ^* such that we can set $H(\psi) = \frac{k_H}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 2)$ for $0 \leq \psi \leq \psi^*$ only and have those field lines never touch the light cylinder at all, avoiding the smoothness requirements.

Whether or not this can be true remains an open question. On the one hand, personal observations suggest that no matter what value of ψ^* one chooses, there is always some grid range that would make that value of ψ^* cross the light cylinder and thus ruin the solution. One cannot use this idea to rule out every nonzero value of ψ^* because this would require an infinite grid. However, with a finite grid, one could rule out any arbitrarily small value of ψ^*, essentially doing the same thing.

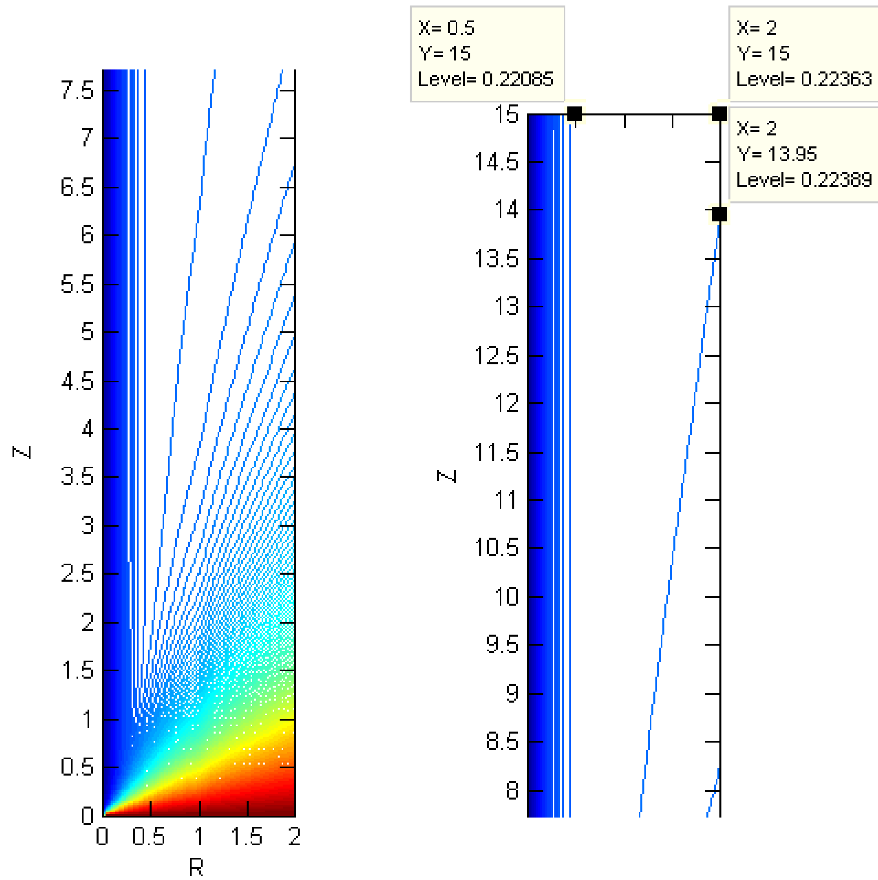


Figure 4.7: Monopole jets solution with $k_H = 8.48$ and $\psi_{eq} = 1$. The marked data points show very little change, indicating an "exclusion zone."

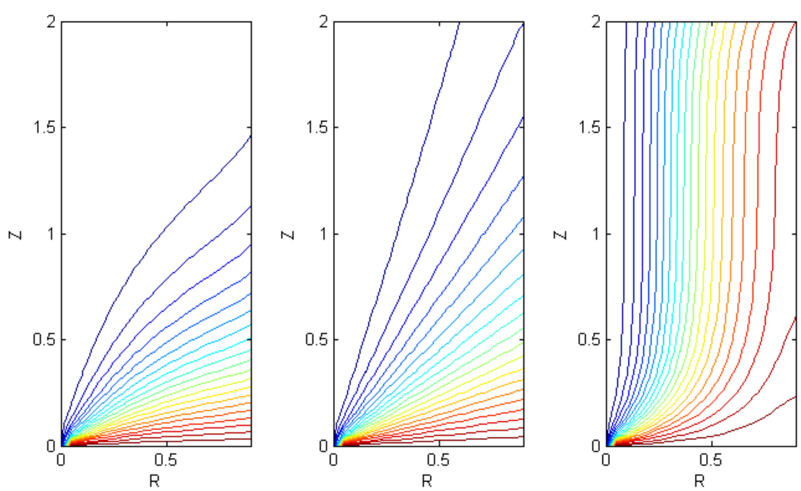


Figure 4.8: Jets solutions with $k_H = 0$, 2, and 5.48 and $\psi_{eq} = 1$.

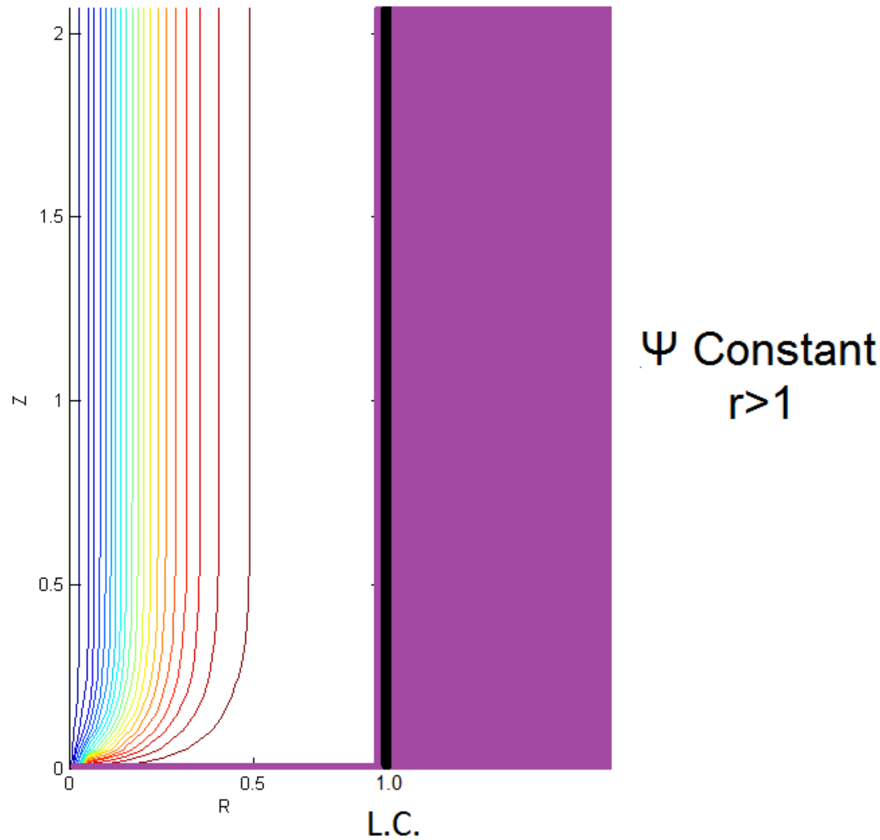


Figure 4.9: Theoretical idea for a jets solution.

On the other hand, imagine the theoretical picture in Figure 4.9. Consider if all the space within the light cylinder was filled with jets emanating from the origin. Further consider that there is one field line that travels from the origin along the equator, and then shoots up the light cylinder. Define this line as ψ_{eq}. Now past the light cylinder ψ is simply the constant value of ψ_{eq} (this is a more drastic example of an “exclusion zone”). Although this solution extends past the light cylinder in a trivial fashion, it does technically satisfy all of the boundary conditions. Further, this solution would be smooth over the light cylinder, and would have no jet line crossing the light cylinder ruining the solution. In terms of F, we would have:

$$F(\psi) = \frac{k_H^2}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 1)(\beta\frac{\psi}{\psi_{eq}} - 2) \qquad 0 \leq \psi < \psi_{eq} \qquad (4.7)$$

$$F(\psi) = 0 \qquad \psi \geq \psi_{eq} \qquad (4.8)$$

The two main differences between this theoretical solution and other jets solutions we have studied are that this theoretical solution has the entire region inside the light cylinder filled with jets, and has no jet contour crossing the light cylinder. One or both of these differences could be the key to having a viable jets solution. Of course, whether or not this is an actual solution to the pulsar equation remains to be seen.

One last issue is whether β really is just 1. Perhaps it is not a constant at all, but rather a function of ψ. In Chapter 6 we provide more analysis on this point.

4.5 Null Sheet

A recent case presented in [7] presents an opportunity to demonstrate a more complex boundary condition. This new case was similar to [2], but used a different bottom boundary condition (for R_0 past the light cylinder). We implemented this using:

$$(b_1, b_2, b_3, b_4, b_5) = (0, 0, -1, 0, 1) \tag{4.9}$$

$$b_6 = -\Delta Z \sqrt{\frac{2\int_0^{\psi(R_0,Z_0)} F(\psi')d\psi'}{j^2(\Delta R)^2 - 1}} \tag{4.10}$$

Where the integral is done using the trapezoidal rule. Unlike other cases, b_6 had to be updated every iteration. Figure 4.10 shows an attempt to replicate the behavior of [7] using the Altered CKF method. Convergence in this case was less favorable, suggesting that a finer and further-reaching grid would be required to study this case correctly. We do note that the behavior of the contour lines is consistent with what [7] saw.

As a further demonstration opportunity, we considered the fitting expression for $H(\psi)$ given in [7], which could then be fed into the Altered TOTS method. After our substitutions:

$$H(\psi) = 1.07\psi \left(2 - \frac{\psi}{\psi_{eq}}\right)\left(1 - \frac{\psi}{\psi_{eq}}\right)^{0.4} \tag{4.11}$$

$$F(\psi) = \frac{2}{5}(1.07)^2\psi \left[6\left(\frac{\psi}{\psi_{eq}}\right)^2 - 12\frac{\psi}{\psi_{eq}} + 5\right]\left(2 - \frac{\psi}{\psi_{eq}}\right)\left(1 - \frac{\psi}{\psi_{eq}}\right)^{-0.2} \tag{4.12}$$

Figure 4.11 shows the result. Although the graph has a slight light cylinder ripple, the behavior is similar. If nothing else, this suggests that this form for $H(\psi)$ is a reasonable approximation.

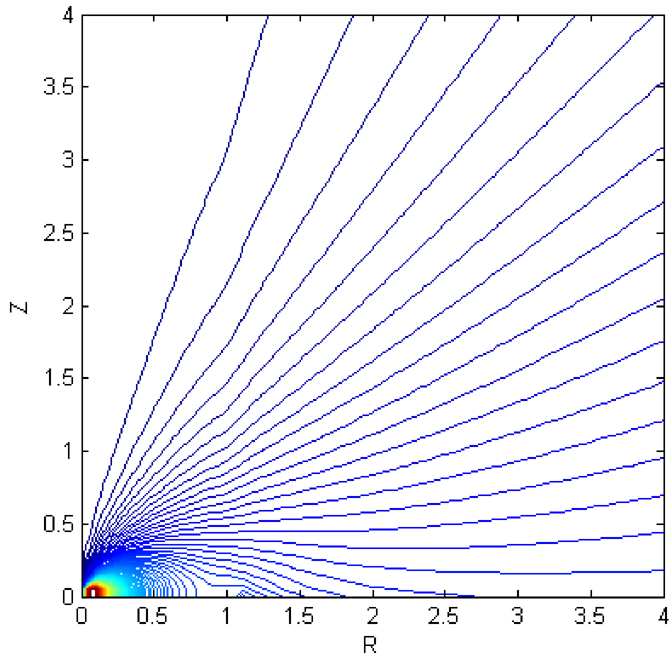


Figure 4.10: Null Sheet solution with $\psi_{eq} = 2.0$ using the Altered CKF method.

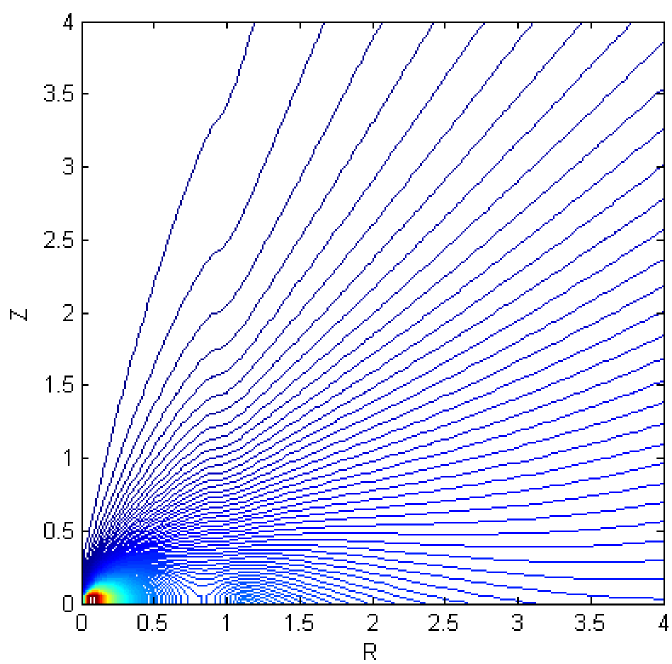


Figure 4.11: Null Sheet solution with $\psi_{eq} = 2.0$ using the Altered TOTS method.

CHAPTER 5

ADDITIONAL CRITIQUE ON TOTS SOLUTIONS

Here we show the cause of the curious ripples at the light cylinder of TOTS solutions. As we have mentioned, the Altered TOTS method is very straightforward conceptually. Make a guess for $F(\psi)$, and then the method will solve for ψ. Indeed, if you use $F(\psi)$ for the monopole, you get the correct monopole answer. Also, after generating a CKF answer, one can take the $F(\psi)$ values produced and plug them into Altered TOTS to generate the exact same CKF answer. Note that in both of these cases, one gets a smooth solution across the light cylinder.

In contrast, although similar to CKF answers, the graphs in Figures 4.4 and 4.5 have ripples at the light cylinder. However, so far it seems like a minor issue. Perhaps this is an artifact of using a coarse grid, or maybe the guess of $F(\psi)$ used simply gives a solution with field lines curving slightly upwards. Remember, this guess is meant to be an approximation, after all.

In a strict mathematical sense, however, we believe that the simulations actually return singular solutions, but that a combination of a coarse grid and unnecessary smoothing have disguised this fact. In Appendix B we will show the affect of taking away smoothing. However, someone doing simulations with smoothing may not see such a drastic effect, and may not even suspect the solution might be singular at all. Here we present another way of seeing singular behavior. In Figure 5.1 we repeat the use of the TOTS method for a finer and finer grid. The ripple region does appear to be shrinking in size. However, while ψ and $\frac{\partial\psi}{\partial R}$ seem to behave, $\frac{\partial^2\psi}{\partial R^2}$ is tending towards singular behavior. This creates a problem, for if ψ did exist at the light cylinder, $\frac{\partial^2\psi}{\partial R^2}$ (and higher derivatives) would have to exist there too (see Appendix D).

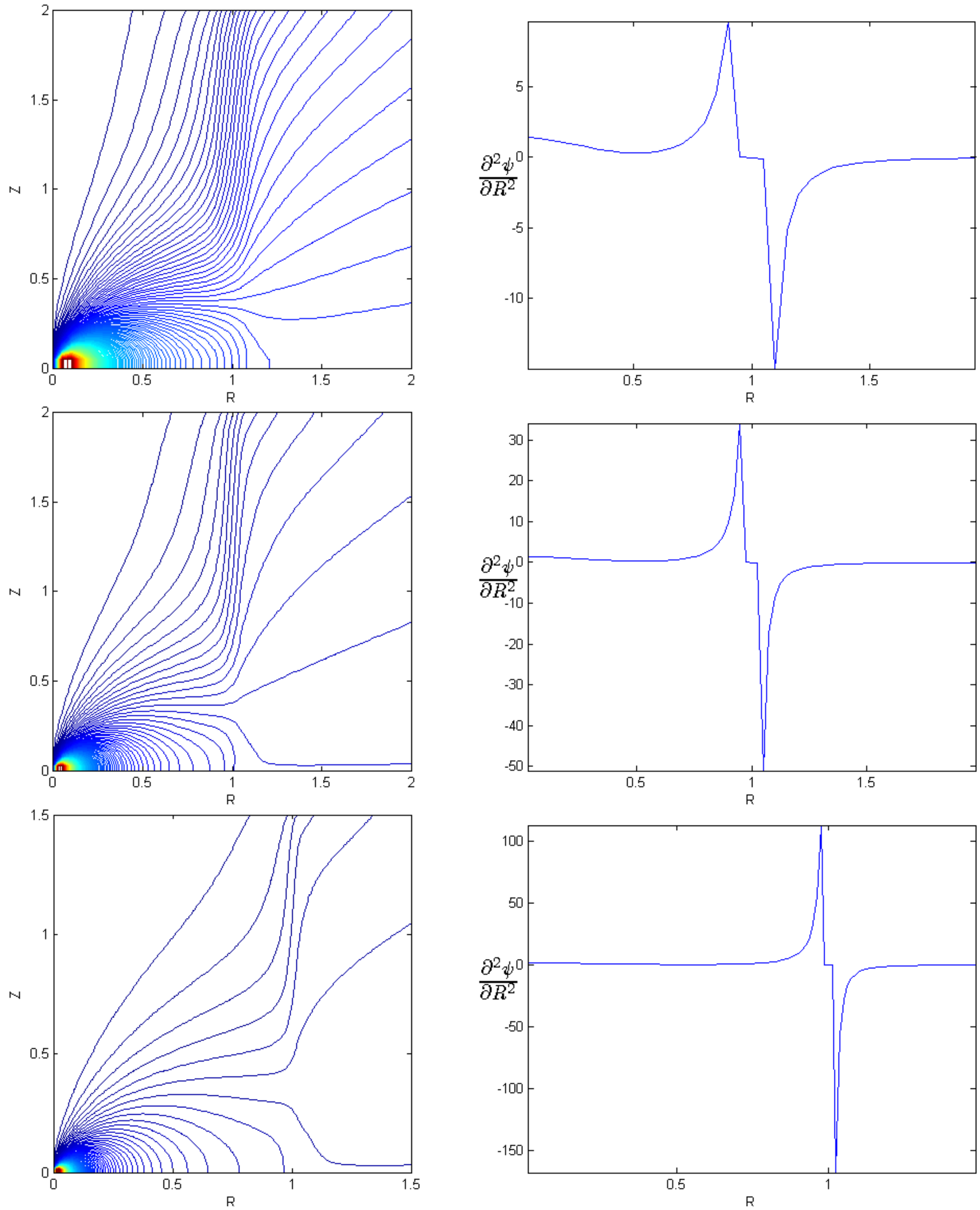


Figure 5.1: TOTS solutions with $\psi_{eq} = 1.0$ using the Altered TOTS method. The grid spacing in both directions is equal, and is 0.05, 0.025, and 0.0125. The second derivative vs. R is plotted next to the corresponding solution and is taken at $Z = 1$.

Do we believe the new method introduced in [3] was wrong? On the contrary, we believe it works exactly as advertised. Make a guess for $F(\psi)$, and then the method will solve for ψ, *whatever* the answer may be. There is no smoothing or iterative correcting here. You get whatever answer the equation has with that guess of $F(\psi)$, singular or otherwise.

One might wonder what exactly the takeaway message is. There are two ways to interpret this. The strict mathematical interpretation is that the guess of $F(\psi)$ made in [3] was wrong. Evidently, their guess corresponds to a singular solution, one that can exist on one side of the light cylinder or the other, but cannot go across and exist in all space. A solution cannot be "almost smooth." CKF solution is smooth, and TOTS solution is not.[1]

However, there is a more optimistic outlook. Their guess of $F(\psi)$ is meant to be an approximation of the (iteratively generated) CKF $F(\psi)$, and their solutions do look like CKF solutions. Furthermore, if given an exact form of $F(\psi)$ that corresponds to a known smooth solution, the method does return that smooth answer. So the method introduced in [3] clearly has some uses.

Now consider this observation. If you take a TOTS answer, and take the grid of ψ values and feed it as an initial guess to the Altered CKF method, then the answer is iteratively corrected to the CKF answer. Indeed, it is believed that this answer is always a unique answer. But this may hide valuable information. Using the TOTS method with an incorrect guess of $F(\psi)$ will give the actual, singular answer. The severity of the singular behavior could be an indication of how far off one is from a correct answer, and the lack of singular behavior

[1]Note that CKF plots of $F(\psi)$ appear to be well-behaved everywhere, and [3] guessed $F(\psi)$ as a polynomial, which is ∞-differentiable. By Appendix D, if they did find a smooth solution, all the derivatives of ψ with respect to R would have to exist at the light cylinder, ruling out any jumps in ψ or any of the R-derivatives.

could suggest that the guess is correct. In contrast, CKF seems to always give the same answer, no matter what you guess.[2]

With this in mind, in the following chapter, we show one way that the Altered TOTS method could be used as an investigative tool.

[2]Instead of initializing $F(\psi)$ to 0, one could initialize to a guess of $F(\psi)$, and then CKF would still reveal the number of iterations to converge, which may give some information. However, this would not be as straightforward to utilize.

CHAPTER 6

ADDITIONAL CRITIQUE ON JETS SOLUTIONS

Reconsider the guess of [4]:

$$H(\psi) = \frac{k_H}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 2) \tag{6.1}$$

$$F(\psi) = \frac{k_H^2}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 1)(\beta\frac{\psi}{\psi_{eq}} - 2) \tag{6.2}$$

In this chapter we take this guess for all $0 \leq \psi < \psi_{eq}$. It was previously assumed that β was a constant. Here, we ask if considering β as a function of ψ (and k_H) will shed any light on this case. This idea was first conceived by noticing a curious observation. In some simulations using the Altered TOTS method, the solution would be singular at the light cylinder, but would have one very specific ψ contour that would pass through smoothly. Further, the value of this "lone contour" would change when β was changed. By finding pairs of ψ and β values, the hope was to derive a functional form for $\beta(\psi)$ that when used, would ensure all ψ contours would pass smoothly through the light cylinder.[1]

Figure 6.1 and Table 6.1 show the points we found. Of course, there is an inherent error in using the above form of $F(\psi)$, since that formula assumes $\frac{d\beta}{d\psi} = 0$. One would have to vary both β and $\frac{d\beta}{d\psi}$ in the corrected $F(\psi)$ formula to remedy this. Nevertheless, the result of this "lone contour method" (see Appendix C) was both comforting and disappointing. We tried the functional fit:

$$\beta = \frac{2}{k_H} + 2\left(\frac{\psi_{eq}}{\psi}\right)\left(1 - \frac{2}{k_H}\right) \tag{6.3}$$

[1] There can be more than one "lone contour." For example, if a parameter p really is $(\psi - 4)(\psi - 5)$, then if we set $p = 2$, both $\psi = 3$ and $\psi = 6$ contours will be smooth. We still stand by the name because these contours are usually separate from each other and are surrounded by singular contours, so they would both be alone.

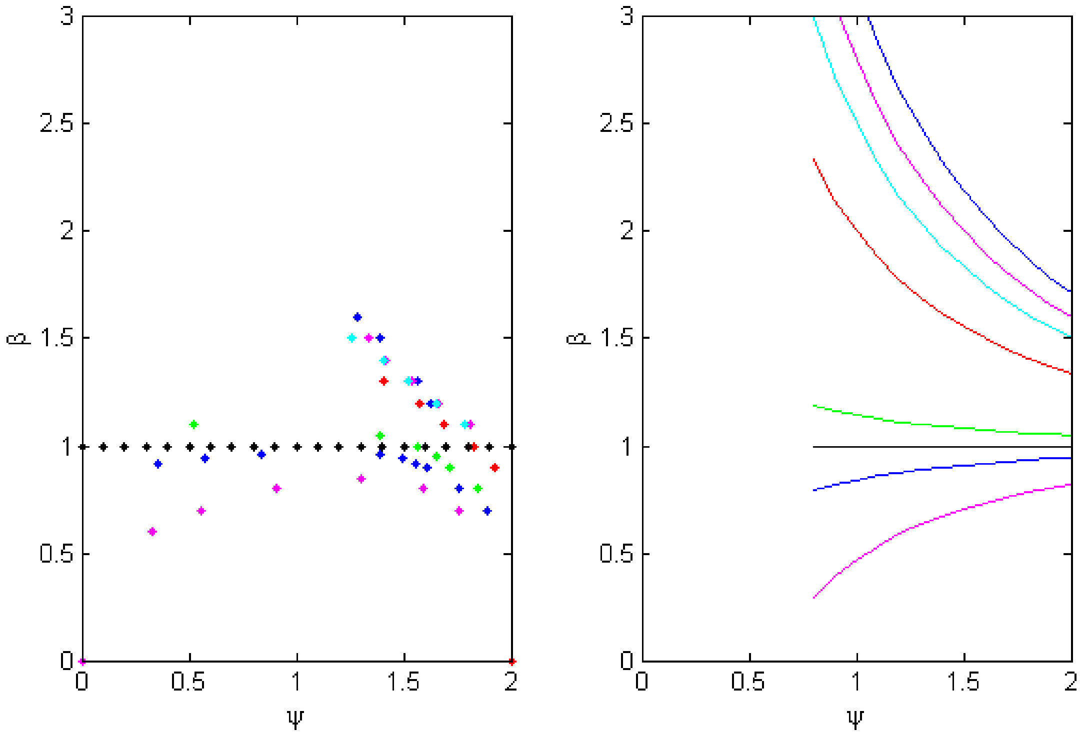


Figure 6.1: β vs. ψ, with $\psi_{eq} = 2$. From bottom to top, $k_H = 1.7$ (magenta), 1.9 (blue), 2.0 (black), 2.1 (green), 3.0 (red), 4.0 (cyan), 5.0 (magenta), and 7.0 (blue). $k_H = 2$ is the monopole solution ($\beta = 1$). The left plot is data estimated from the lone contour method, and the right plot is of Equation 6.3. See Table 6.1 for the complete list of data.

Which when plugged into $H(\psi)$ simply gives $H(\psi) = \psi(\frac{\psi}{\psi_{eq}} - 2)$, or the old monopole answer. This is identically true whether or not k_H is a function of ψ.

Now, we know this functional fit must work (the monopole answer is well established as being a valid solution), and it seems like the functional form of $\beta(\psi)$ that will work is unique. The obvious conclusion to jump to is that this fit that gives the monopole answer is the only fit that will work, ruling out any jet-like answers. If this is true, this would mean that if there is a new, non-monopole solution, it is not of the form that [4] guessed. As we allowed both k_H and β to vary, this means an entire 2-dimensional space of parameters is knocked out.

The only problem is that the fit does not do a very good job of passing through the data points. $k_H \leq 2$ is plausible, but $k_H > 2$ values have a larger and larger gap from the fitted curve as k_H increases. Further, as ψ approaches ψ_{eq}, the data points seem to keep decreasing, perhaps to a vertical asymptote. But we can easily see that using our fit:

$$\beta(\psi_{eq}) = 2\left(1 - \frac{1}{k_H}\right) \tag{6.4}$$

Which has very different behavior. Our interpretation of this is that the fit would work, except that one needs to vary both β and $\frac{d\beta}{d\psi}$ when searching for lone contours. After all, while a function and its derivative are obviously related, pointwise they are independent. Also, the bottom boundary artificially makes ψ_{eq} a smooth contour, which could skew the data points near ψ_{eq}.

There is one more parameter that we have already seen, the "r" in the TOTS $F(\psi)$. Revisiting Figures 4.4 and 4.5, we can see that despite the ripples at the light cylinder, there are contour lines that are able to pass through smoothly. These cases only vary ψ_{eq}, so we would expect the smooth $\frac{\psi}{\psi_{eq}}$ to be the same throughout all the simulations we have presented. We estimate some pairs of (ψ_{eq},ψ) to be (0.5,0.49299), (0.8,0.79135), and (1.2,1.19720), which all have a ratio close to 1. This was done with $r = 0.5$, so perhaps by varying r and finding this ratio, we could derive a relationship $r(\psi)$.

The primary reason for interest is that TOTS is approximating the CKF solution, which we *know* exists. So it really could be that finding $r(\psi)$ will help determine a closed form of the CKF solution. In the jets case, there was no definitive evidence that anything other than the monopole existed, so in retrospect it was not surprising that nothing new was found.

Table 6.1: Data obtained from the lone contour method. k_H and β are fixed in the program, while ψ is found from running the simulation.

k_H	ψ	β	k_H	ψ	β	k_H	ψ	β
7.0	1.2862	1.60	2.1	0.5194	1.10	1.0	1.5181	0.10
	1.3880	1.50		1.3889	1.05		1.2752	0.00
	1.4205	1.40		1.5625	1.00		1.7311	0.00
	1.5630	1.30		1.6537	0.95		1.1473	-0.10
	1.6239	1.20		1.7174	0.90		1.8280	-0.10
5.0	1.3358	1.50		1.8462	0.80		1.0404	-0.20
	1.4204	1.40	1.9	0.8405	0.96		1.9371	-0.20
	1.5413	1.30		1.3865	0.96		0.9499	-0.30
	1.6628	1.20		1.4937	0.94		0.8710	-0.40
	1.8135	1.10		0.5724	0.94		0.8102	-0.50
4.0	1.2560	1.50		0.3584	0.92		0.7476	-0.60
	1.4113	1.40		1.5578	0.92		0.6961	-0.70
	1.5214	1.30		1.6053	0.90		0.6474	-0.80
	1.6496	1.20		1.7615	0.80		0.6034	-0.90
	1.7839	1.10		1.8912	0.70		0.5619	-1.00
3.0	1.4046	1.30	1.7	1.2993	0.85	0.3	1.7811	-4.50
	1.5769	1.20		0.9098	0.80		1.3922	-4.50
	1.6879	1.10		1.5941	0.80		1.2471	-5.00
	1.8313	1.00		0.5570	0.70		1.0449	-6.00
	1.9221	0.90		1.7554	0.70		0.9152	-7.00
				0.3300	0.60		0.8054	-8.00
							0.7302	-9.00
							0.6405	-10.00
							0.5653	-11.00

CHAPTER 7

CONCLUSIONS

The altered methods augment the original methods by refining them and removing unnecessary steps which were introduced to accommodate numerical relaxation. We successfully have replicated prior results, and we strongly advocate that future researchers consider using Newton's method in place of relaxation, especially when using the TOTS method. We also see how the two ideas of CKF and TOTS have clearly different uses, and how the more recent TOTS method, rather than competing with CKF, can branch off into different investigative paths by using its unique ability to reveal, rather than correct and hide, singular solutions.

APPENDIX A

BOUNDARY CONDITIONS

We previously argued that in Equation 3.1, it is OK if coefficient a_5 vanishes. However, we must be careful about the boundary conditions. Although a small detail, it is important not to overlook a subtlety in dealing with boundary conditions where the term we wish to solve for does not exist because its coefficient vanishes. Using the example of the left boundary, consider these two cases:

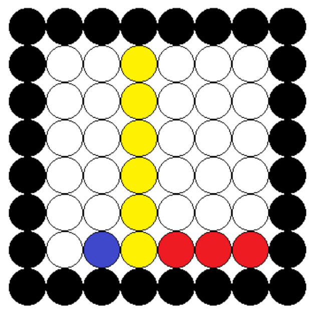
1. If the left boundary depends on the boundary point to the left of the edge point, then one can write:

$$b_1 U_{j-1,k} + b_2 U_{j+1,k} + b_3 U_{j,k-1} + b_4 U_{j,k+1} + b_5 U_{j,k} = b_6$$

Where b_1 will be nonzero to reflect the dependence on $U_{j-1,k}$. Then we can simply solve for $U_{j-1,k}$.

2. If the left boundary does not depend on the boundary point to the left of the edge point, then b_1 will be zero, and one cannot solve for $U_{j-1,k}$. However, in general $U_{j-1,k}$ will appear in the instance of Equation 3.1 that we wish to solve. For such boundary conditions, we can simply forget Equation 3.1 and write one custom equation. So, for example, if the left edge had the boundary condition $\frac{\partial\psi}{\partial Z}|_{(R_0,Z_0)} = 0$, then the equation would be $U_{j,k+1} - U_{j,k} = 0$. Here, the custom equation is in place of the usual technique of taking Equation 3.1 and plugging in the boundary equation. For the particular boundary conditions we demonstrated, this issue does not arise, but other boundary conditions that have been proposed may have to deal with this.

More complicated edge boundary conditions can also be accommodated by custom equations. For example, CKF's original mandate was that ψ_{eq} be found iteratively, rather than being a fixed parameter. Consider the following grid:



The light cylinder is marked in yellow, and the blue grid point is located at (j^*, k^*). To enforce CKF's mandate, we use Equation 3.1 at each red point, and plug in the custom boundary equation $U_{j,k-1} = U_{j^*,k^*}$.

The last boundary issue to consider is what happens at a corner. A corner point touches two boundaries, and typically this just means creating two boundary equations to eliminate both dependencies. Sometimes, however, there are complications. For example, with our choice of boundary conditions, consider the upper right corner. Both the top and right boundaries give you the same equation, $-jU_{j-1,k} + jU_{j+1,k} - kU_{j,k-1} + kU_{j,k+1} = 0$. One strategy would be to write something of the form $f_1(U_{j+1,k}, U_{j,k+1}) = f_2(U_{j-1,k}, U_{j,k-1}, U_{j,k})$ in an effort to eliminate both $U_{j+1,k}$ and $U_{j,k+1}$ with one equation. In general, however, there is no guarantee this would work.

The way we get around this is to note that, when dealing with the outer boundaries, we can really use any two boundary equations, as long as they are compatible with the actual boundary condition. So, just for this corner point, take these equations instead:

Top Boundary:

$$\frac{\partial\psi}{\partial Z}|_{(R_0,Z_0)} = 0 \longrightarrow (0,0,-1,1,0,0) \tag{A.1}$$

Right Boundary:

$$\frac{\partial\psi}{\partial R}|_{(R_0,Z_0)} = 0 \longrightarrow (-1,1,0,0,0,0) \tag{A.2}$$

This gives us two equations that can eliminate both boundary points. Note that these equations being true forces the actual boundary equations to be true.

APPENDIX B

SMOOTHING OVER THE LIGHT CYLINDER

In forming the nonlinear equations, a priori we do not have to give the light cylinder points any special treatment. However, even if the system of equations with our choice of boundary conditions converges to something, there is no guarantee that this "something" reached is smooth, or even a real solution at all. This is why simulation methods incorporate some sort of smoothing requirement. The original CKF method, for example, achieves a smooth solution through its iterative process. For us, we have overwritten the equations for points near the light cylinder.

One might wonder what would happen without any smoothing. Can the nonlinear equations be solved as is? Reconsider the overwrite that accommodated the light cylinder:

If $N_{LC} - Q \leq j \leq N_{LC} + Q$

Then $(a_1, a_2, a_3, a_4, a_5, a_6) = (-1/2, -1/2, 0, 0, 1, 0)$

Where $Q \leq -1$ would eliminate the overwrite completely (the condition would never be true and this would be skipped), $Q = 0$ would only affect the light cylinder itself, and $Q \geq 1$ forces smoothness further away ($Q > 1$ is usually not needed).

In every simulation up to this point, $Q = 1$ was used. But perhaps this smoothing step is completely unnecessary. Eliminating the smoothness overwrite completely would clean up the new methods further. In Figures B.1 and B.2 we present various solutions, with $Q = -1$, 0, and 1. All simulations in this section are done with a grid spacing of 0.025 in both directions, and use an

80x80 grid.[1]

The Altered CKF method seems to have slight trouble at $Q = -1$. Although some answers generally appear to be smooth over the light cylinder, it seems we do need additional smoothing to get rid of some ripples. However, the need for additional smoothing is an illusion. CKF demands that ψ at the light cylinder is the average of the values to the immediate left and right of it, and this is precisely the only difference between $Q = -1$ and $Q = 0$. So the Altered CKF method does not really need any additional smoothing. Using $Q = -1$ would just create a conflict of goals, tugging the solution in two different directions needlessly. We simply use $Q = 0$ as an easy way to enforce a part of the method without causing unnecessary complication.

As for the Altered TOTS method, we have previously argued that the TOTS solutions were indeed singular at the light cylinder, and for all values of Q we can see trouble. Furthermore, using an estimate of $F(\psi)$, as in the Null Sheet case, will also produce ripples for all Q. This makes sense, as the estimate itself is not likely to be an exact allowed function, so there will not be an exact smooth solution to go with it. We can also see how using a Q value that is too high may smooth over problems in the answer. But if the Altered TOTS method is used in a case where there is known to be a solution, then the method will work with $Q = -1$. For example, the monopole picture looks smooth. Also, if one uses the Altered CKF method to get an answer (with any Q), but then uses the iteratively found $F(\psi)$ in the Altered TOTS method with $Q = -1$, then you will get the exact same answer despite removing the smoothing. In other words, absolutely

[1] One might ask from a practical point of view why this matters at all. If you already know you will get a smooth answer, then using some smoothing is a sensible idea. However, if a solution is singular, smoothing can hide that fact. The point here is to show that there exists the option not to use smoothing.

no smoothing is required. Altered TOTS can simply solve the equations, exactly as they are, to get an answer. Smoothing is simply a convenience to get the answer in less iterations.

So in essence, both altered methods do not require much special treatment of the light cylinder (above and beyond what the CKF method already demanded). Any smoothing introduced is only to speed up simulations. It is comforting to know there is no theoretical need for it. The only real requirement we have not addressed is that the light cylinder lands exactly on the grid.

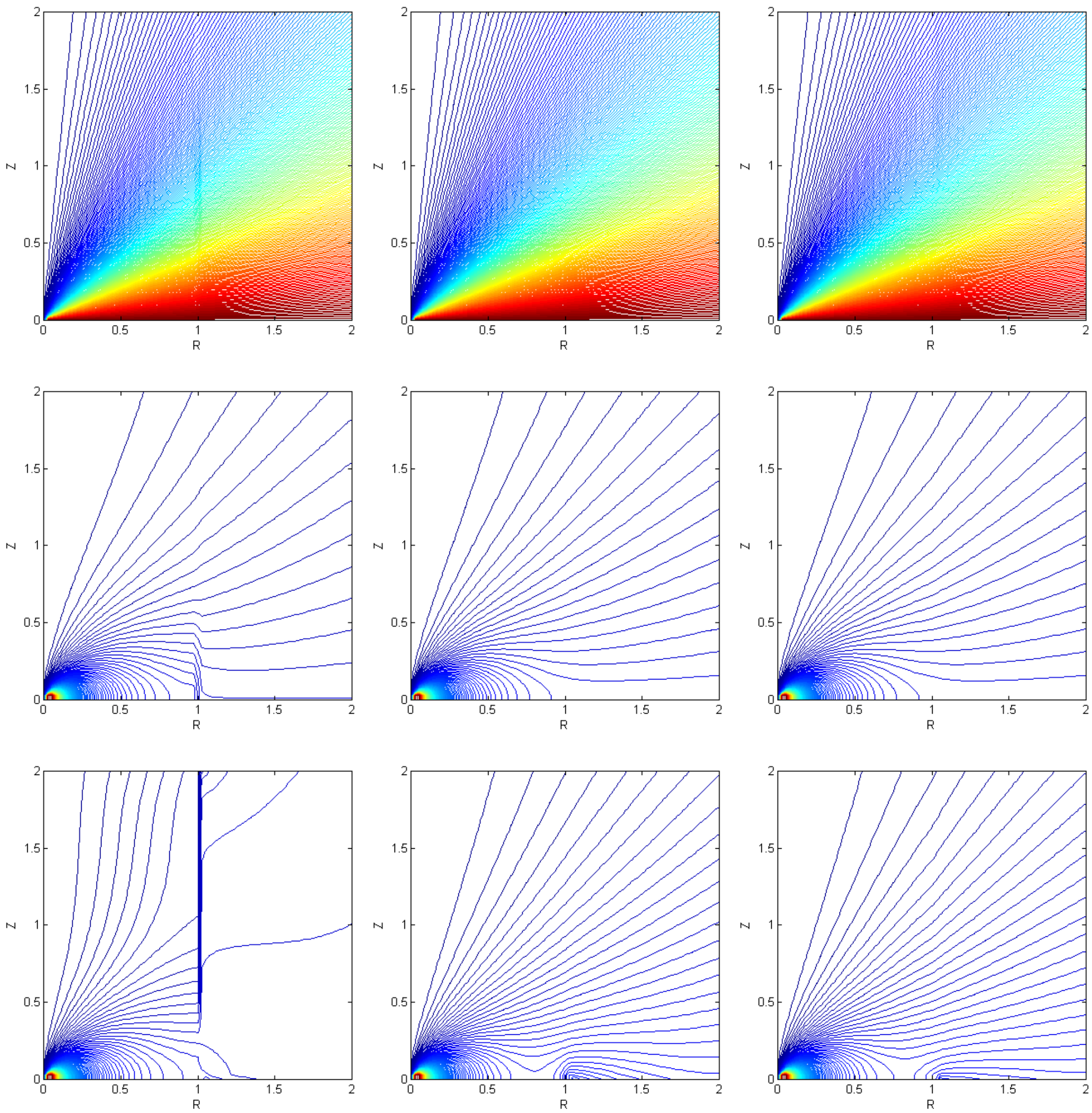


Figure B.1: Various solutions with the Altered CKF method. Top row is monopole ($\psi_{eq} = 1$), then CKF ($\psi_{eq} = 1.28$), then Null Sheet ($\psi_{eq} = 2$). Left column is $Q = -1$, then $Q = 0$, then $Q = 1$.

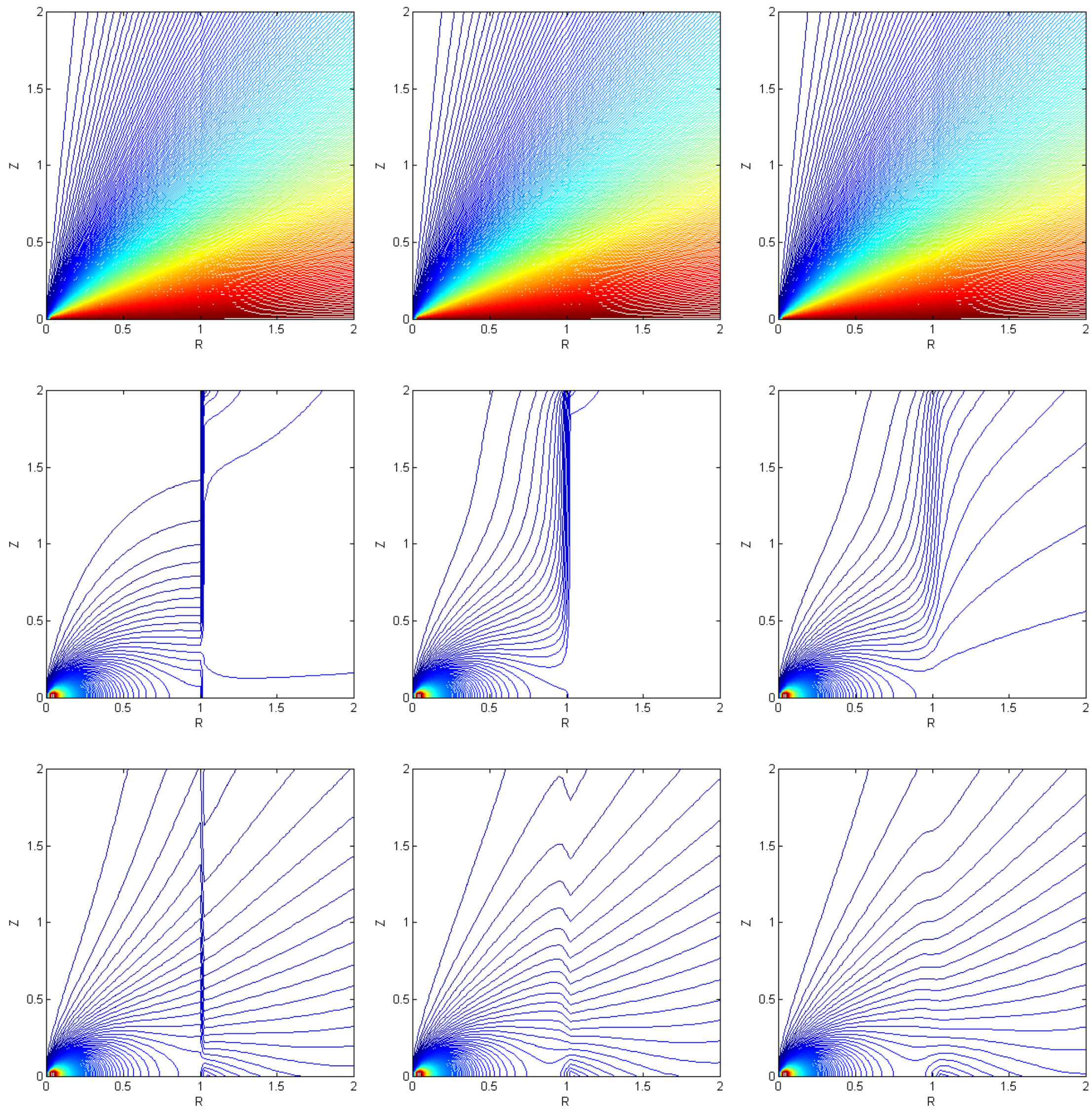


Figure B.2: Various solutions with the Altered TOTS method. Top row is monopole ($\psi_{eq} = 1$), then TOTS ($\psi_{eq} = 1.28$), then Null Sheet ($\psi_{eq} = 2$). Left column is $Q = -1$, then $Q = 0$, then $Q = 1$.

APPENDIX C

FUTURE OPPORTUNITIES

C.1 The Lone Contour Method

A new investigative method:
Write any guess for $F(\psi)$ that gives a singular answer and that has at least one parameter that can be varied. Run simulations with the Altered TOTS method and search for "lone" smooth contours by varying the parameters. Use this data to derive functional relationships between the parameters and smooth ψ values, in an effort to correct the original $F(\psi)$ guess.

Of course, one must exercise caution. For example, when we applied this method to the jets case, β was a function of both k_H and ψ. One could imagine that having many parameters could make this complicated. Another consideration is that it is not easy to estimate the lone contour's value. We show how we identify it in Figure C.1, but a lot of estimation is required. Our grid spacing of 0.0125 is not small enough, but we do feel that a finer grid would eventually provide any level of precision desired. A further reaching grid would also be useful to avoid undue influence from the boundary.

One thing this method has going for it is that we can take advantage of our shortcut of keeping J^{-1} fixed. Since we are only changing $F(\psi)$ and not the grid itself, we can simply read J^{-1} from a file, which is much faster than calculating a matrix inverse each simulation. Further, for a last bit of speed, we can run multiple simulations with J^{-1} in memory, and use the end result of one simulation as the initial guess for the next.

One last reminder is that one need not choose a form of $F(\psi)$ that generalizes the monopole. The jets guess does generalize it (reducing to the monopole for $k_H = 2$), but the TOTS guess does not. Whether or not there are separate families of solutions, or one correct general functional form is an open question.

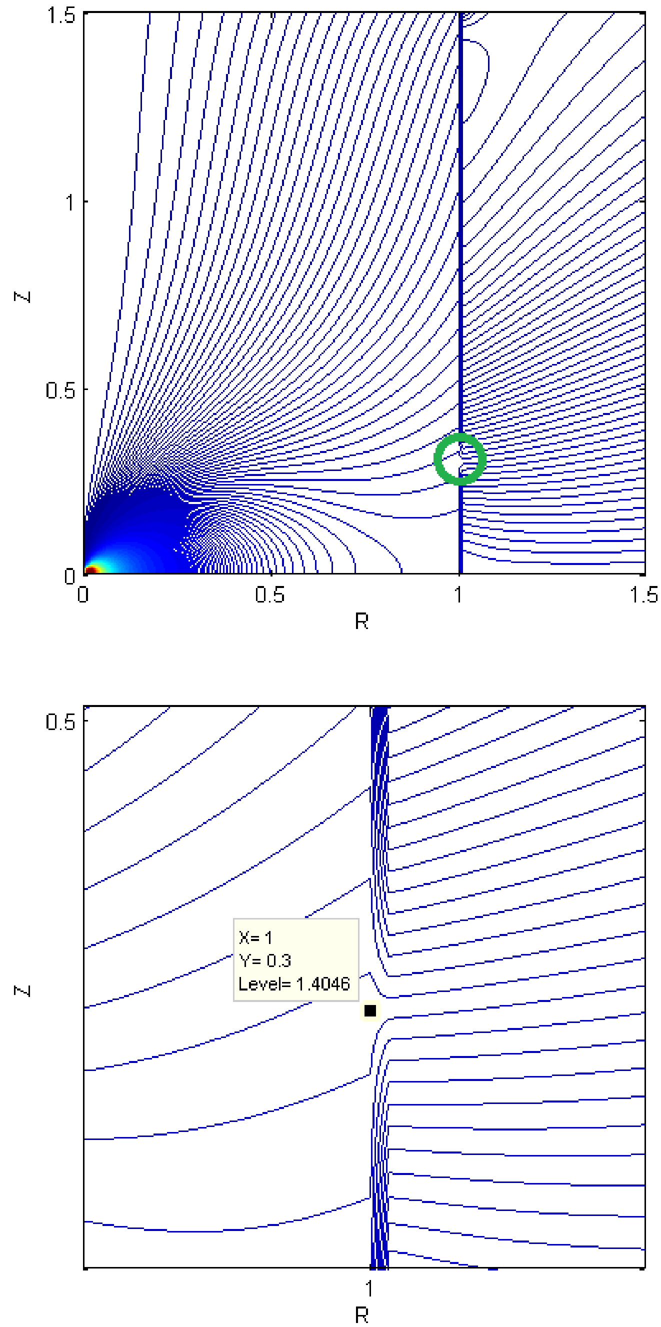


Figure C.1: Jets simulation ($Q = -1, \psi_{eq} = 2, k_H = 3, \beta = 1.3$), demonstrating a lone contour. The entire light cylinder is singular except for one location. Our grid spacing is 0.0125 in both directions, but a higher resolution is recommended for further study.

C.2 Unique Solution?

If we were to fix a boundary, choose a method (CKF or TOTS, original or altered), and reach a solution, is that solution unique? So far, the evidence says yes. However, most of that evidence is the observation that numerical relaxation reaches the same answer despite different initial values. However, there is an opportunity to get stronger evidence.

Newton's method reveals a new way of looking at this problem, i.e. as a set of nonlinear equations. Although Newton's method has converged to a unique solution for every problem considered thus far, writing the nonlinear equations provides an invitation for other solving techniques that could search for additional physical solutions. One possible avenue that was considered was homotopy, a process which utilizes Newton's method to trace out paths to each solution to the nonlinear equations, rather than simply converging to the one closest to the initial values. One could discard solutions with any nonpositive ψ value, and then look at any solutions that remain. This technique could either find other physical solutions, or simply provide further evidence of their nonexistence.

APPENDIX D

MATHEMATICAL PROOF

Let $C(n) \equiv$ "Let $(1, Z_0)$ be a point on the light cylinder ($Z_0 > 0$) where ψ exists. If the n^{th} derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$, then for $m \in \mathbb{Z}$, $\frac{\partial^m \psi}{\partial R^m}$ exists at $(1, Z_0)$ if $1 \leq m \leq n+1$."

Claim: For $n \in \mathbb{Z}_{\geq 0}$, $C(n)$ is true.

Proof: We use induction on n:

Base case: Prove $C(0)$

$(1, Z_0)$ is on the light cylinder, so the pulsar equation will give us the usual smoothness condition there:

$$\frac{\partial \psi}{\partial R} = \frac{1}{2} F(\psi) \tag{D.1}$$

Now, ψ exists at $(1, Z_0)$, and $F(\psi)$ exists at $\psi(1, Z_0)$, so simply plug in $\psi(1, Z_0)$ into $F(\psi)$ to get $F(\psi)$ at $(1, Z_0)$. Since this must exist, the other side of the smoothness equation must also exist. Hence, $\frac{\partial \psi}{\partial R}$ exists at $(1, Z_0)$.

Inductive Step: For $n > 0$, prove $C(n-1) \longrightarrow C(n)$

Here, we are assuming the n^{th} derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$, and this also means that the $(n-1)^{th}$ derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$. By the inductive assumption, we get to conclude that $\frac{\partial^m \psi}{\partial R^m}$ exists at $(1, Z_0)$ for $1 \leq m \leq n$. We just need to show that $\frac{\partial^{n+1} \psi}{\partial R^{n+1}}$ also exists at $(1, Z_0)$.

Once again consider the smoothness condition. With additional differentiation:

$$\frac{\partial^{n+1}\psi}{\partial R^{n+1}} = \frac{1}{2}\frac{\partial^n F(\psi)}{\partial R^n} \tag{D.2}$$

To apply the chain rule a variable number of times, we take advantage of a form of Faà di Bruno's formula:

$$\frac{\partial^n F(\psi)}{\partial R^n} = \sum_{j=1}^{n} \frac{\partial^j F(\psi)}{\partial \psi^j} B_{n,j}\left(\frac{\partial \psi}{\partial R}, \ldots, \frac{\partial^{n-j+1}\psi}{\partial R^{n-j+1}}\right) \tag{D.3}$$

Where $B_{n,j}$ are the Bell polynomials of combinatorial mathematics. The only detail about them that matters is that if the arguments all exist, then this polynomial will also exist.

Since $1 \leq j \leq n$, by our assumption and by plugging in $\psi(1, Z_0)$, all of the required derivatives of $F(\psi)$ will exist at $(1, Z_0)$. Also, for $1 \leq j \leq n$, we have $1 \leq n - j + 1 \leq n$ and as we previously argued, all of the required derivatives of ψ will exist at $(1, Z_0)$. So all terms on the right hand side exists at $(1, Z_0)$, meaning $\frac{\partial^n F(\psi)}{\partial R^n}$ exists at $(1, Z_0)$, and thus $\frac{\partial^{n+1}\psi}{\partial R^{n+1}}$ exists at $(1, Z_0)$, completing the proof.

Notes about the proof:

1: We previously fixed the light cylinder to be $R = 1$.

2: For convenience in writing the proof, the "zeroth derivative" of a function is understood to just be the function itself.

3: The claim is written in terms of $F(\psi)$. However it is straightforward to show that for $n \geq 0$:

"The n^{th} derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$." $\longleftrightarrow$

"The $(n+1)^{th}$ derivative of $H(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$."

APPENDIX E

$H(\psi)$ DICTIONARY

For reference, and especially for those who would like to employ the lone contour method, we would like to offer a centralized reference of $H(\psi)$ guesses (mostly from other authors), and any solutions known. This will give perspective on how these guesses are related, and which parameters they all have and where there is room to generalize more. Note that not all of these guesses have physical boundary conditions on $H(\psi)$ applied, but we do assume that ψ is always greater than or equal to the constant term. Also, we remind the reader that if $\psi(R, Z)$ and $F(\psi)$ fit the pulsar equation, then $k_1\psi(R, Z + k_2) + k_3$ and $k_1 F(\psi - k_3)$ must also fit.[1]

We have attempted to list the first source that proposed, in some way, each solution in our dictionary. Solutions marked "Original," to the best of our knowledge, are not mentioned in prior literature. The trivial solution is obvious so we will refrain from giving credit. Also, since the CKF solution is only known implicitly, it does not yet have a place in our dictionary.

[1] We do not show shifts of the star origin in the formulas explicitly, but from the pulsar equation, and as pointed out by [8], solutions should be unaffected by a shift in the Z coordinate. In other words, if we solve the pulsar equation, we are really finding a solution representing a star centered at any Z location, but at R=0. One could easily alter the pulsar equation by shifting the R coordinate. Then we could get answers centered anywhere.

"Trivial":

$$\psi = c_1 Z + c_2$$
$$H(\psi) = \text{ [Any Constant]}$$
$$F(\psi) = 0$$

"Real Monopole" [5]:

$$\psi = c_1 + \frac{c_2 Z}{\sqrt{R^2 + Z^2}} \quad (c_2 \neq 0)$$
$$H(\psi) = \pm c_2 \left(\frac{\psi - c_1}{c_2} + 1 \right) \left(\frac{\psi - c_1}{c_2} - 1 \right)$$
$$F(\psi) = 2(\psi - c_1) \left(\frac{\psi - c_1}{c_2} + 1 \right) \left(\frac{\psi - c_1}{c_2} - 1 \right)$$

"TOTS" [3]:

$$\psi = \psi_{eq}(c_1 + \text{[Unknown Non-constant Terms]}) \quad (r \neq 0)$$
$$H(\psi) = \pm \left(\frac{\psi - c_1}{\sqrt{2r}} \right) \sqrt{\left(\frac{\psi - c_1}{\psi_{eq}} \right)^2 - \frac{4}{3}(r+1) \left(\frac{\psi - c_1}{\psi_{eq}} \right) + 2r}$$
$$F(\psi) = \left(\frac{\psi - c_1}{r} \right) \left(\frac{\psi - c_1}{\psi_{eq}} - 1 \right) \left(\frac{\psi - c_1}{\psi_{eq}} - r \right)$$

"Jets" [4]:

$$\psi = \psi_{eq}(c_1 + \text{[Unknown Non-constant Terms]}) \quad (k_H \neq 0)$$
$$H(\psi) = \pm \frac{k_H}{2}(\psi - c_1) \left(\beta \frac{\psi - c_1}{\psi_{eq}} - 2 \right)$$
$$F(\psi) = \frac{k_H^2}{2}(\psi - c_1) \left(\beta \frac{\psi - c_1}{\psi_{eq}} - 1 + \frac{\beta'}{2} \frac{(\psi - c_1)^2}{\psi_{eq}} \right) \left(\beta \frac{\psi - c_1}{\psi_{eq}} - 2 \right)$$

"Null Sheet" [7]:

$$\psi = \psi_{eq}(c_1 + \text{[Unknown Non-constant Terms]})$$
$$H(\psi) = \pm 1.07(\psi - c_1)\left(2 - \frac{\psi - c_1}{\psi_{eq}}\right)\left(1 - \frac{\psi - c_1}{\psi_{eq}}\right)^{0.4}$$
$$F(\psi) = \frac{2}{5}(1.07)^2(\psi - c_1)\left[6\left(\frac{\psi - c_1}{\psi_{eq}}\right)^2 - 12\frac{\psi - c_1}{\psi_{eq}} + 5\right]$$
$$\left(2 - \frac{\psi - c_1}{\psi_{eq}}\right)\left(1 - \frac{\psi - c_1}{\psi_{eq}}\right)^{-0.2}$$

"Imaginary Monopole" (Original):

$$\psi = c_1 + \frac{c_2}{\sqrt{R^2 + Z^2}} \quad (c_2 \neq 0)$$
$$H(\psi) = \pm(\sqrt{-1})(\psi - c_1)\left(\frac{\psi - c_1}{c_2}\right)$$
$$F(\psi) = -2(\psi - c_1)\left(\frac{\psi - c_1}{c_2}\right)^2$$

"No Z dependance" (Original):

$$\psi = c_1 R^j + c_2 \quad (j \neq 0 \ \& \ c_1 \neq 0)$$
$$H(\psi) = \begin{cases} \pm\sqrt{(\psi - c_2)^2 + 2c_1^2 Log_e(c_2 - \psi)} & j = 1 \\ \\ \pm j(\psi - c_2)\sqrt{1 - \left(\frac{j-2}{j-1}\right)\left(\frac{\psi - c_2}{c_1}\right)^{-\frac{2}{j}}} & \text{else} \end{cases}$$
$$F(\psi) = (\psi - c_2)\left(j^2 - j(j-2)\left(\frac{\psi - c_2}{c_1}\right)^{-\frac{2}{j}}\right)$$

"Simple R and Z dependance" [8]:

$$\psi = c_1 R^2 Z + c_2 \quad (c_1 \neq 0)$$
$$H(\psi) = \pm 2(\psi - c_2)$$
$$F(\psi) = 4(\psi - c_2)$$

APPENDIX F

SOURCE CODE

We provide a complete listing of the code used. Eight files represent the main structural code, while eighteen files each represent a distinct simulation case. All code is in the C language, and was compiled with Pelles C 7.00.347.

F.1 Main Code

This section includes the source code for the main structure of the simulation.

- **matrix.c** is the main code and entry point. This contains a control panel which allows the user to change the grid parameters and other settings in one centralized location. The main program loop is here, along with all the data writing to files at the end of the simulation.
- **matrix_build.c** fills in the entries to the matrix and vector that represent the linear parts of the pulsar equation. This code assumes that variable relationships are only with nearest neighbors.
- **inverse.c** calculates the inverse of a matrix. Our shortcut ensures that we only need to call this once.
- **equation_solver.c** and **equation_solver_2case.c** use Newton's method to solve the nonlinear pulsar equation. Any nonlinearity comes from $F(\psi)$. The latter file allows the accommodation of double simulations, which start with one case and then switch to another, such as using the CKF method and then feeding the $F(\psi)$ values generated into the TOTS method.
- **reset_ckf.c** and **reset_tak.c** reset part of the bottom boundary to accommodate Null Sheet cases.
- **read_matrix.c** reads the inverse Jacobian matrix from file so that it does not have to be calculated, saving a significant amount of time. The simulation parameters used when the matrix was written cannot be changed, or else reading the matrix for that new simulation will result in nonsensical results. Also, this is not recommended for any Null Sheet cases.

F.1.1 matrix.c

```
//Matrix Builder
//Applies Newton's method to differential equations using a matrix.
//Corresponds to Vidal and Lovelace 2014 (Draft) and "New Treatment
    of the Pulsar Equation" (Master's Thesis).
//Created September 25, 2013 By Michael Joseph Vidal.

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h>
#include <math.h>
#include <string.h>
#include <time.h>

//Definitions for the types of simulations found in the program.
#define MONOPOLE 1
#define STANDARD 2
#define JETS 3
#define NULLSHEET 4

//This region of code serves as a control panel for the rest of the
    program. Comment or uncomment macros to get desired behavior.
typedef double T; //Choose your precision (float or double)

/*
Makes program read the matrix "J_INVERSE" from file instead of
    calculating it.
EVERY grid setting below must EXACTLY match the ones used in the file
     read.
READ may not work for Null Sheet cases.
*/
//#define READ

#define SHORTCUT //Assumes constant Jacobian Matrix.
T V_FRAC=0.1; //Similar to relaxation parameter.

//These only affect the jets case. So far, only ckf_jets.c has this
    implemented.
#define JETS1 //Use boundary used by Takamori in jets case (this one
    is the typical choice).
//#define JETS2 //Use boundary used by Lovelace in jets case.

#define SMOOTH 1 //Amount of smoothing w.r.t. the light cylinder.
    SMOOTH 1 is the typical value.

#define NUM_ITERATIONS_MAX 100000 //Maximum number of iterations,
    regardless of convergence.

//Grid Parameters.
#define N_MAX_X 42
```

```
#define N_MAX_Y 42
#define N_LC 20
#define DX 0.05 //Make sure N_LC*DX=1.
#define DY 0.05
#define N_S_X 2 //Make sure the star is square i.e. DX*N_S_X=DY*N_S_Y.
#define N_S_Y 2

//Other simulation parameters.
#define RATIO 0.5 //Only used in TOTS case.
#define P_OP 1.0 //PSI EQUATORIAL.

int toggle=0; //Only used in double simulation cases.
T BETA=1.0; //Only used in jets case. (Defining it here allows us to change it mid simulation.

//Sizes of various objects used in the program.
const int SIZE_T=sizeof(T);
const int SIZE_T_X=(N_MAX_X-2)*sizeof(T);
const int SIZE_T_Y=(N_MAX_Y-2)*sizeof(T);
const int SIZE_T_X_Y=(N_MAX_X-2)*(N_MAX_Y-2)*sizeof(T);
const int SIZE_T_XY=(N_MAX_X-2)*(N_MAX_Y-2)*sizeof(T);
const int SIZE_T_XY_XY=(N_MAX_X-2)*(N_MAX_X-2)*(N_MAX_Y-2)*(N_MAX_Y-2)*sizeof(T);

//Files that are written to. Some are extra files that may be useful for debugging or for secondary data.
FILE *fp_r,*fp_z,*fp_a,*fp_b,*fp_j,*fp_j2,*fp_hhp,*fp_hhpPRIME,*fp_v,*fp_vxy,*fp_P_HHP,*fp_change;

T L_func[(N_MAX_X-2)];
T L_X[(N_MAX_X-2)*(N_MAX_Y-2)];
T L_Y[(N_MAX_X-2)*(N_MAX_Y-2)];
T CHANGE[(N_MAX_X-2)][(N_MAX_Y-2)];

T **A;
T *B;
T **J;
T **J_INVERSE;
T **HHP;
T **HHP_PRIME;
T *v0;
T *v1;
T *eq;
T **h;
T *p;

T **a1,**a2,**a3,**a4,**a5,**a6;
T *bL1,*bL2,*bL3,*bL4,*bL5,*bL6;
T *bR1,*bR2,*bR3,*bR4,*bR5,*bR6;
T *bB1,*bB2,*bB3,*bB4,*bB5,*bB6;
T *bT1,*bT2,*bT3,*bT4,*bT5,*bT6;
```

```c
T ***c;

void initialize(void);
void star(void);
__forceinline void create_matrix(void);
void writeToFiles(void);
__forceinline int RN(int,int);
__forceinline int CN_X(int);
__forceinline int CN_Y(int);
__forceinline T f(int,int);
__forceinline void hhpSet(T**,T**);
void equationBuilder(int);
void inverse(T**,int,T**);
T** Make2DTArray(int,int);
void resetCKF(void);
void resetTak(void);
void read(void);

/*
Case file choices. Include EXACLTY ONE of these to choose the case.
*/
//Single simulation cases.
#include "ckf_monopole.c" //Pure Monopole case with CKF method.
//#include "ckf.c" //Pure CKF method from Contopoulos et al. 1999.
//#include "ckf_jets.c" //CKF case mixed with Jets case.
//#include "ckf_null.c" //Null Sheet case with CKF method.
//#include "tak_monopole_test.c" //Test of the Monopole case using the known answer.
//#include "tak_monopole.c" //Pure Monopole case with the TOTS method.
//#include "tak_monopole_jets.c" //Monopole case mixed with Jets case.
//#include "tak.c" //Pure TOTS method from Takamori et al. 2012.
//#include "tak_jets.c" //TOTS case mixed with Jets case.
//#include "tak_theory_jets.c" //Theoretical jets case.
//#include "tak_null.c" //Null Sheet case with TOTS method.

/*
Double simulation cases. There are two general situations:
You start with TOTS method, and then go to CKF method,
or, You start with CKF method, and then go to TOTS method.
Note: Doing TOTS first will give CKF the grid of PSI values. Doing CKF first will give TOTS the grid of HHP values.
*/
//#include "ckf_monopole_jets.c" //CKF Monopole, then Jets.
//#include "ckf_tak.c" //CKF, then TOTS.
//#include "ckf_null_tak.c" //CKF Null Sheet, then TOTS.
//#include "tak_ckf.c" //TOTS, then CKF.
//#include "tak_ckf_jets.c" //TOTS, then CKF Jets.
//#include "tak_ckf_null.c" //TOTS, then CFK Null Sheet.
//#include "tak_theory_jets_ckf.c" //TOTS Theoretical Jets case, then CKF.
```

```c

//The code that creates the matrix "A" and vector "b" for the linear system
#include "matrix_build.c"

//Additional code to reset part of the bottom boundary every iteration, which is neccessary only for Null Sheet cases.
#include "reset_ckf.c" //Only needed if ckf_null is an included case
#include "reset_tak.c" //Only needed if actual_null is an included case

//Code that calculates the inverse of a matrix.
#include "inverse.c"

//Code to solve the nonlinear equations.
#include "equation_solver.c" //Use with a "single simulation" case.
//#include "equation_solver_2case.c" //Use with a "double simulation" case.

//Code that reads the matrix "J_INVERSE" from file instead of calculating it.
#include "read_matrix.c"

int main(int argc, char *argv[])
{
        //Memory allocation and initialization.
        A=Make2DTArray((N_MAX_X-2)*(N_MAX_Y-2),(N_MAX_X-2)*(N_MAX_Y-2));
        B=malloc(SIZE_T_XY);
        J=Make2DTArray((N_MAX_X-2)*(N_MAX_Y-2),(N_MAX_X-2)*(N_MAX_Y-2));
        J_INVERSE=Make2DTArray((N_MAX_X-2)*(N_MAX_Y-2),(N_MAX_X-2)*(N_MAX_Y-2));
        v0=malloc(SIZE_T_XY);
        v1=malloc(SIZE_T_XY);
        eq=malloc(SIZE_T_XY);
        h=Make2DTArray((N_MAX_X-2)*(N_MAX_Y-2),(N_MAX_X-2)*(N_MAX_Y-2));
        p=malloc(SIZE_T_XY);

        HHP=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        HHP_PRIME=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);

        a1=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        a2=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        a3=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        a4=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        a5=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        a6=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);

        bL1=malloc(SIZE_T_Y);
        bL2=malloc(SIZE_T_Y);
```

```
        bL3=malloc(SIZE_T_Y);
        bL4=malloc(SIZE_T_Y);
        bL5=malloc(SIZE_T_Y);
        bL6=malloc(SIZE_T_Y);

        bR1=malloc(SIZE_T_Y);
        bR2=malloc(SIZE_T_Y);
        bR3=malloc(SIZE_T_Y);
        bR4=malloc(SIZE_T_Y);
        bR5=malloc(SIZE_T_Y);
        bR6=malloc(SIZE_T_Y);

        bB1=malloc(SIZE_T_X);
        bB2=malloc(SIZE_T_X);
        bB3=malloc(SIZE_T_X);
        bB4=malloc(SIZE_T_X);
        bB5=malloc(SIZE_T_X);
        bB6=malloc(SIZE_T_X);

        bT1=malloc(SIZE_T_X);
        bT2=malloc(SIZE_T_X);
        bT3=malloc(SIZE_T_X);
        bT4=malloc(SIZE_T_X);
        bT5=malloc(SIZE_T_X);
        bT6=malloc(SIZE_T_X);

        c=malloc(4*SIZE_T_X_Y);
        c[0]=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        c[1]=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        c[2]=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
        c[3]=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);

        //Gives a rough idea of how much memory is used while the program is running.
        printf("SIZE T MBytes=%f\n",(T)SIZE_T/(1024*1024));
        //Big stuff
        printf("Big Stuff\n");
        printf("MBytes A=%f\n",(T)SIZE_T_XY_XY/(1024*1024));
        printf("MBytes J=%f\n",(T)SIZE_T_XY_XY/(1024*1024));
        printf("MBytes J-1=%f\n",(T)SIZE_T_XY_XY/(1024*1024));
        printf("MBytes h=%f\n",(T)SIZE_T_XY_XY/(1024*1024));

        //Small stuff
        printf("\nSmall Stuff\n");
        printf("MBytes B=%f\n",(T)SIZE_T_XY/(1024*1024));
        printf("MBytes v,eqBlock=%f\n",3.0*SIZE_T_XY/(1024*1024));
        printf("MBytes p=%f\n",(T)SIZE_T_XY/(1024*1024));
        printf("MBytes hhpBlock=%f\n",2.0*SIZE_T_X_Y/(1024*1024));
        printf("MBytes aBlock=%f\n",6.0*SIZE_T_X_Y/(1024*1024));
        printf("MBytes bBlock=%f\n",12.0*(SIZE_T_X+SIZE_T_Y)/(1024*1024));
        printf("MBytes cBlock=%f\n",4.0*SIZE_T_X_Y/(1024*1024));
```

```
#ifndef READ
        //Initialize everything to zero
        for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
        {
                B[j]=0;
                for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
                {
                        A[j][k]=0;
                }
        }
#endif

        for(int x=0;x<N_MAX_X-2;x++)
        {
                for(int y=0;y<N_MAX_Y-2;y++)
                {
                        HHP[x][y]=0;
                        HHP_PRIME[x][y]=0;
                }
        }

        //Prepare files. Note that some files are for secondary information that is largely unused.
        fp_r=fopen("r.dat","w+"); //R Values.
        fp_z=fopen("z.dat","w+"); //Z Values.
        fp_a=fopen("a.dat","w+"); //A Matrix.
        fp_b=fopen("b.dat","w+"); //B Vector.
        fp_j=fopen("j.dat","w+"); //Jacobian Matrix.
#ifdef READ
        fp_j2=fopen("j2.dat","r+"); //Inverse Jacobian Matrix.
#else
        fp_j2=fopen("j2.dat","w+"); //Inverse Jacobian Matrix.
#endif
        fp_hhp=fopen("hhp.dat","w+"); //HHP Matrix.
        fp_hhpPRIME=fopen("hppPRIME.dat","w+"); //D(HHP)/D(PHI) Matrix.
        fp_v=fopen("v.dat","w+"); //PSI Values (written in a line).
        fp_vxy=fopen("vxy.dat","w+"); //PSI Values (written in a grid).
        fp_P_HHP=fopen("P_HHP.dat","w+"); //PSI vs. HHP.
        fp_change=fopen("change.dat","w+"); //Matrix of change of PSI between iterations.

        //Exits the program immediately if a file cannot be opened, to prevent unnecessary time wasting.
        if(NULL==fp_hhp||NULL==fp_r||NULL==fp_z||NULL==fp_v)
        {
                printf("Error - Cannot open file.\n");
                exit(0);
        }

```

```
        //Main program, separated into parts to show the user progress.
        printf("\n1\n");
        initialize(); //Case specifc discretization of the differential equation and boundary.
        printf("2\n");
        star(); //Case specific insertion of the star.
        printf("3\n");
        create_matrix(); //Creates the actual matrix "A" and vector "B".
        printf("4\n");
#ifdef READ
        read(); //Reads J_INVERSE.
#endif
        equationBuilder((N_MAX_X-2)*(N_MAX_Y-2)); //Creates equations from "A", "B", and "HHP" and applies Newton's method until convergence.
        printf("5\n");

        _fcloseall();
}

//There are two possible coordinate systems. (x,y) reflects a grid, and (s) is just all the grid points in a line.
//Converts between (x,y) and (s)
__forceinline int RN(int m, int n)
{
        return n*(N_MAX_X-2)+m;
}

//Converts between (s) and (x,y)
__forceinline int CN_X(int r)
{
        return r%(N_MAX_X-2);
}

__forceinline int CN_Y(int r)
{
        return r/(N_MAX_X-2);
}

void writeToFiles(void)
{
        rewind(fp_r);
        rewind(fp_z);
        rewind(fp_a);
        rewind(fp_b);
        rewind(fp_j);
        rewind(fp_j2);
        rewind(fp_hhp);
        rewind(fp_hhpPRIME);
        rewind(fp_v);
```

```
        rewind(fp_vxy);
        rewind(fp_P_HHP);
        rewind(fp_change);

        //Write actual R values.
        for (int j=0;j<=N_MAX_X-2;j++)
        {
                fprintf(fp_r, "%f\n",j*DX);
        }

        //Write actual Z values.
        for (int k=N_MAX_Y-2;k>=0;k--)
        {
                fprintf(fp_z, "%f\n",k*DY);
        }

        //Write "A" matrix .
        for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
        {
                for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
                {
                        fprintf(fp_a, "%40.20f ",A[j][k]);
                }
                fprintf(fp_a, "\n");
        }

        //Write "b" vector.
        for (int k=0;k<(N_MAX_X-2)*(N_MAX_Y-2);k++)
        {
                fprintf(fp_b, "%40.20f\n",B[k]);
        }

        //Write "J" matrix.
        for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
        {
                for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
                {
                           fprintf(fp_j, "%40.20f ",J[j][k]);
                        }
                        fprintf(fp_j, "\n");
        }

#ifndef READ
        //Write "J-1" matrix .
        for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
        {
                for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
                {
                           fprintf(fp_j2, "%40.20f ",J_INVERSE[j][k]);
                        }
                        fprintf(fp_j2, "\n");
```

```
        }
#endif

        //Write HPP.
        for (int k=N_MAX_Y-3;k>=0;k--)
        {
                for (int j=0;j<N_MAX_X-2;j++)
                {
                        fprintf(fp_hhp, "%40.20f ",HHP[j][k]);
                        }
                        fprintf(fp_hhp, "\n");
        }

        //Write D(HHP)/D(PHI).
        for (int k=N_MAX_Y-3;k>=0;k--)
        {
                for (int j=0;j<N_MAX_X-2;j++)
                {
                        fprintf(fp_hhpPRIME, "%40.20f ",HHP_PRIME[j][k]);
                        }
                fprintf(fp_hhpPRIME, "\n");
        }

        //Write "v" vector.
        fprintf(fp_v, "%40.20f ",0.0);
        for (int j=0;j<N_MAX_X-2;j++)
        {
                //Deal with bottom boundary.
#if TYPE==MONOPOLE
                //Monopole P=P_OP
                fprintf(fp_v, "%40.20f ",P_OP);
#else
                //Others
                if(j<N_LC+1)
                {
                        //All non-monopole cases Pz=0
                        fprintf(fp_v, "%40.20f ",v0[RN(j,0)]);
                }
                else
                {
#if (TYPE==STANDARD||TYPE==JETS)
                        //CKF, Takamori, Jets P=P_OP
                        fprintf(fp_v, "%40.20f ",P_OP);
#elif TYPE==NULL
                        //CKF_NULL, TAK_NULL H^2=(R^2-1)*(Pz)^2
                        fprintf(fp_v, "%40.20f ",v0[RN(j,0)]-L_func[j]);
#endif
                }

#endif
```

```
		}
		for (int k=0;k<N_MAX_Y-2;k++)
		{
			fprintf(fp_v, "%40.20f ",0.0);
			for (int j=0;j<N_MAX_X-2;j++)
			{
				fprintf(fp_v, "%40.20f ",v0[RN(j,k)]);
				}
		}

		//Write "v" grid.
		for (int k=N_MAX_Y-3;k>=0;k--)
		{
			for (int j=0;j<N_MAX_X-2;j++)
			{
				fprintf(fp_vxy, "%40.20f ",v0[RN(j,k)]);
				}
				fprintf(fp_vxy, "\n");
		}

		//Write P vs HHP list.
		for(int k=0;k<=N_MAX_Y-3;k++)
		{
			for(int j=0;j<=N_MAX_X-3;j++)
			{
				if(j>N_S_X||k>N_S_Y)
				{
					fprintf(fp_P_HHP,"%40.20f %40.20f\n",
					   v0[RN(j,k)],HHP[j][k]);
				}
			}
		}

		//Write "change of PSI" grid.
		for (int k=N_MAX_Y-3;k>=0;k--)
		{
			for (int j=0;j<N_MAX_X-2;j++)
			{
				fprintf(fp_change, "%40.20f ",CHANGE[j][k]);
				}
				fprintf(fp_change, "\n");
		}

		fflush(fp_r);
		fflush(fp_z);
		fflush(fp_a);
		fflush(fp_b);
		fflush(fp_j);
		fflush(fp_j2);
		fflush(fp_hhp);
		fflush(fp_hhpPRIME);
		fflush(fp_v);
```

```
        fflush(fp_vxy);
        fflush(fp_P_HHP);
        fflush(fp_change);
}

//Function to allocate memory for a 2D array of elements of type T.
T** Make2DTArray(int arraySizeX, int arraySizeY)
{
        T** theArray;
        theArray = (T**) malloc(arraySizeX*sizeof(T*));
        for (int i=0;i<arraySizeX;i++)
        {
                theArray[i] = (T*) malloc(arraySizeY*sizeof(T));
        }
        return theArray;
}
```

F.1.2 matrix_build.c

```c
__forceinline void create_matrix(void)
{
	//"Interior interior" points (points that do not touch a boundary)
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		for(int t=1;t<=N_MAX_Y-4;t++)
		{
			A[RN(s,t)][RN(s-1,t)]=a1[s][t];
			A[RN(s,t)][RN(s+1,t)]=a2[s][t];
			A[RN(s,t)][RN(s,t-1)]=a3[s][t];
			A[RN(s,t)][RN(s,t+1)]=a4[s][t];
			A[RN(s,t)][RN(s,t)]=a5[s][t];
			B[RN(s,t)]=a6[s][t]-DX*DX*f(s+1,t+1);
		}
	}

	//Left Edge
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=0;

		//A[RN(s,t)][RN(s-1,t)]=0;
		A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a1[s][t]*bL2[t]/bL1[t];
		A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a1[s][t]*bL3[t]/bL1[t];
		A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a1[s][t]*bL4[t]/bL1[t];
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a1[s][t]*bL5[t]/bL1[t];
		B[RN(s,t)]=a6[s][t]-a1[s][t]*bL6[t]/bL1[t]-DX*DX*f(s+1,t+1);
	}

	//Right Edge
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=N_MAX_X-3;

		A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a2[s][t]*bR1[t]/bR2[t];
		//A[RN(s,t)][RN(s+1,t)]=0;
		A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a2[s][t]*bR3[t]/bR2[t];
		A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a2[s][t]*bR4[t]/bR2[t];
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a2[s][t]*bR5[t]/bR2[t];
		B[RN(s,t)]=a6[s][t]-a2[s][t]*bR6[t]/bR2[t]-DX*DX*f(s+1,t+1);
	}
```

```
	//Bottom Edge
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		int t=0;

		A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a3[s][t]*bB1[s]/bB3[s];
		A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a3[s][t]*bB2[s]/bB3[s];
		//A[RN(s,t)][RN(s,t-1)]=0;
		A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a3[s][t]*bB4[s]/bB3[s];
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a3[s][t]*bB5[s]/bB3[s];
		B[RN(s,t)]=a6[s][t]-a3[s][t]*bB6[s]/bB3[s]-DX*DX*f(s+1,t+1);
	}

	//Top Edge
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		int t=N_MAX_Y-3;

		A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a4[s][t]*bT1[s]/bT4[s];
		A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a4[s][t]*bT2[s]/bT4[s];
		A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a4[s][t]*bT3[s]/bT4[s];
		//A[RN(s,t)][RN(s,t+1)]=0;
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a4[s][t]*bT5[s]/bT4[s];
		B[RN(s,t)]=a6[s][t]-a4[s][t]*bT6[s]/bT4[s]-DX*DX*f(s+1,t+1);
	}

	//Bottom Left Corner
	{
		int s=0;
		int t=0;

		//A[RN(s,t)][RN(s-1,t)]=0;
		A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a1[s][t]*bL2[t]/bL1[t]-a3[s][t]*bB2[s]/bB3[s];
		//A[RN(s,t)][RN(s,t-1)]=0;
		A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a1[s][t]*bL4[t]/bL1[t]-a3[s][t]*bB4[s]/bB3[s];
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a1[s][t]*bL5[t]/bL1[t]-a3[s][t]*bB5[s]/bB3[s];
		B[RN(s,t)]=a6[s][t]-a1[s][t]*bL6[t]/bL1[t]-a3[s][t]*bB6[s]/bB3[s]-DX*DX*f(s+1,t+1);
	}

```

```
	//Top Left Corner
	{
		int s=0;
		int t=N_MAX_Y-3;

		//A[RN(s,t)][RN(s-1,t)]=0;
		A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a1[s][t]*bL2[t]/bL1[t]-a4[s][t]*bT2[s]/bT4[s];
		A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a1[s][t]*bL3[t]/bL1[t]-a4[s][t]*bT3[s]/bT4[s];
		//A[RN(s,t)][RN(s,t+1)]=0;
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a1[s][t]*bL5[t]/bL1[t]-a4[s][t]*bT5[s]/bT4[s];
		B[RN(s,t)]=a6[s][t]-a1[s][t]*bL6[t]/bL1[t]-a4[s][t]*bT6[s]/bT4[s]-DX*DX*f(s+1,t+1);
	}

	//Bottom Right Corner
	{
		int s=N_MAX_X-3;
		int t=0;

		A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a2[s][t]*bR1[t]/bR2[t]-a3[s][t]*bB1[s]/bB3[s];
		//A[RN(s,t)][RN(s+1,t)]=0;
		//A[RN(s,t)][RN(s,t-1)]=0;
		A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a2[s][t]*bR4[t]/bR2[t]-a3[s][t]*bB4[s]/bB3[s];
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a2[s][t]*bR5[t]/bR2[t]-a3[s][t]*bB5[s]/bB3[s];
		B[RN(s,t)]=a6[s][t]-a2[s][t]*bR6[t]/bR2[t]-a3[s][t]*bB6[s]/bB3[s]-DX*DX*f(s+1,t+1);
	}

	//Top Right Corner
	{
		int s=N_MAX_X-3;
		int t=N_MAX_Y-3;

		A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a2[s][t]*bR1[t]/bR2[t]-a4[s][t]*bT1[s]/bT4[s];
		//A[RN(s,t)][RN(s+1,t)]=0;
		A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a2[s][t]*bR3[t]/bR2[t]-a4[s][t]*bT3[s]/bT4[s];
		//A[RN(s,t)][RN(s,t+1)]=0;
		A[RN(s,t)][RN(s,t)]=a5[s][t]-a2[s][t]*bR5[t]/bR2[t]-a4[s][t]*bT5[s]/bT4[s];
		B[RN(s,t)]=a6[s][t]-a2[s][t]*bR6[t]/bR2[t]-a4[s][t]*bT6[s]/bT4[s]-DX*DX*f(s+1,t+1);
	}
}
```

F.1.3 inverse.c

```
//Returns the inverse of the matrix stored in 2D array "m"
void inverse(T** m,int size,T** g)
{
	//Creates a copy of the input matrix so that the original is
	    not destroyed
	T **f;
	f=Make2DTArray((N_MAX_X-2)*(N_MAX_Y-2),(N_MAX_X-2)*(N_MAX_Y
	    -2));
	printf("MBytes f=%f\n",(T)SIZE_T_XY_XY/(1024*1024));

	//The memcpy calls won't work if "f" is dynamically allocated
	    , so we have to manually copy "f" below.
	//memcpy(f,m,(N_MAX_X-2)*(N_MAX_Y-2)*(N_MAX_X-2)*(N_MAX_Y-2))
	    ;

	//Initializes g as the identity matrix, which will turn into
	    the inverse of the original matrix
	for(int x=0;x<size;x++)
	{
		for(int y=0;y<size;y++)
		{
			g[x][y]=(x==y);
			f[x][y]=m[x][y];
		}
	}
	for(int j=0;j<size;j++)
	{
		printf("j=%i out of %i\n",j+1,size);
		/*
		//This code decides when to swap rows, and if the
		    matrix is singular.
		//The memcpy calls won't work if "f" is dynamically
		    allocated.
		//This needs to be rewritten
		if(fabs(f[j][j])<0.0000001)
		{
			//const char key=_getch();
			//exit(0);
			int isSingular=1;
			//Swap rows to avoid a zero pivot
			for(int s=j;s<size;s++)
			{
				if(fabs(f[s][j])>0.0000001)
				{
					T temp[(N_MAX_X-2)*(N_MAX_Y
					    -2)];

					isSingular=0;
					memcpy(temp,f+j,SIZE_T_XY);
					memcpy(f+j,f+s,SIZE_T_XY);
```

```
                        memcpy(f+s,temp,SIZE_T_XY);
                        memcpy(temp,g+j,SIZE_T_XY);
                        memcpy(g+j,g+s,SIZE_T_XY);
                        memcpy(g+s,temp,SIZE_T_XY);
                        break;
                    }
                }
                //if(isSingular)
                {
                    //printf("YOUR JACOBIAN IS SINGULAR !!!!! CANNOT GO ON!!!!!");
                    //printf("DET_J=%f\n",determinantAlt(J,(N_MAX-2)*(N_MAX-2)));
                    //printf("DET_A=%f\n",DET_A);
                    //exit(0);
                    //return;
                }
                //multAccum*=-f[j][j];
            }
            else
            {
                //multAccum*=f[j][j];
            }
            */
            T temp=1/f[j][j];
            for(int l=size-1;l>j;l--)
            {
                f[j][l]*=temp;
            }
            for(int l=j;l>=0;l--)
            {
                g[j][l]*=temp;
            }

            for(int k1=0;k1<j;k1++)
            {
                for(int k2=size-1;k2>j;k2--)
                {
                    f[k1][k2]-=f[k1][j]*f[j][k2];
                }
                for(int k2=j;k2>=0;k2--)
                {
                    g[k1][k2]-=f[k1][j]*g[j][k2];
                }
            }
            for(int k1=j+1;k1<size;k1++)
            {
                for(int k2=size-1;k2>j;k2--)
                {
                    f[k1][k2]-=f[k1][j]*f[j][k2];
                }
                for(int k2=j;k2>=0;k2--)
```

```
                    {
                        g[k1][k2]-=f[k1][j]*g[j][k2];
                    }
                }
            }
}
```

F.1.4 equation_solver.c

```
//Solves for PHI using Newton's method.
//Note: PHI is stored in v0 and v1.
void equationBuilder(int size)
{
        T vINIT[(N_MAX_X-2)*(N_MAX_Y-2)];

#ifndef READ
        for(int x=0;x<size;x++)
        {
                for(int y=0;y<size;y++)
                {
                        J[x][y]=A[x][y];
                        h[x][y]=A[x][y];
                }
        }
#endif

#ifdef SHORTCUT
#ifndef READ
        printf("pre inverse\n");
        inverse(J,size,J_INVERSE);
        printf("after inverse\n");
#endif
#endif

        //Our initial guess for every point is one.
        for(int i=0;i<size;i++)
        {
                vINIT[i]=1.0;
                v0[i]=vINIT[i];
                v1[i]=vINIT[i];
        }

        /*
        //An alternate way of setting initial values.
        //Our initial guess for every point is "random" between 0 and
            1.
        srand(time(NULL));
        for(int i=0;i<size;i++)
        {
                vINIT[i]=((T)rand())/RAND_MAX;
                v0[i]=vINIT[i];
                v1[i]=vINIT[i];
        }
        */

        T olddeltaALL=0;
        T deltaALL=0;
        for(int iiii=1;iiii<=NUM_ITERATIONS_MAX;iiii++)
        {
```

```
		olddeltaALL=deltaALL;
		deltaALL=0;
		T deltaMAX=0;

		//This code controls user interaction. All user input is direct keyboard hits, NOT typing text into a command line.
		//To the best of my knowledge, this code to check for user input does NOT slow the program down significantly.
		if(_kbhit())
		{
			//This call to _getch() will not pause the program because the user already hit a key,
			//but in general the other calls will pause the program and wait for input.
			const char key=_getch();
			if (key=='b'||key=='B')
			{
				//Simulation is paused.
				printf("\nWriting data to files ............\n");
				writeToFiles();
				printf("\nData has been written to files.\n\nEnter 'w' or 'W' to change the FRAC parameter.\nEnter 'e' or 'E' to change BETA.\nEnter 'r' or 'R' to reset the same simulation.\nEnter 'c' or 'C' to reset with another BETA value.\nEnter 'q' or 'Q' to quit.\nEnter anything else to continue.\n");

				const char key=_getch();
				if (key=='w'||key=='W')
				{
					//Yes, the code needs to be structured this way!
					while(1)
					{
						printf("Enter new V_FRAC value:\n");
						scanf("%lf",&V_FRAC);
						fflush(stdin);//NOT STANDARD :(
						rewind(stdin);//NOT STANDARD :(
						printf("\nV_FRAC=%lf - Is this OK?\nEnter 'y' or 'Y' to continue\nEnter
```

```
                         'a' or 'A' to try again.\n\n",V_FRAC);

                char key;
                while(1)
                {
                        key=_getch();
                        if (key=='y'||key=='Y'||key=='a'||key=='A')
                        {
                                break;
                        }
                }
                if (key=='y'||key=='Y')
                {
                        break;
                }
        }
}
else if (key=='e'||key=='E')
{
        while(1)
        {
                printf("Enter new BETA value:\n");
                scanf("%lf",&BETA);
                fflush(stdin);//NOT STANDARD :(
                rewind(stdin);//NOT STANDARD :(
                printf("\nBETA=%lf - Is this OK?\nEnter 'y' or 'Y' to continue\nEnter 'a' or 'A' to try again.\n\n",BETA);

                char key;
                while(1)
                {
                        key=_getch();
                        if (key=='y'||key=='Y'||key=='a'||key=='A')
```

```
                            {
                                break;
                            }
                        }
                        if (key=='y'||key=='Y')
                        {
                            break;
                        }
                    }
                }
                else if (key=='r'||key=='R')
                {
                    //Completely reset simulation.
                    for(int i=0;i<size;i++)
                    {
                        vINIT[i]=1;
                        v0[i]=vINIT[i];
                        v1[i]=vINIT[i];
                        iiii=1;
                    }
                }
                else if (key=='c'||key=='C')
                {
                    //Completely reset simulation with different BETA value.
                    for(int i=0;i<size;i++)
                    {
                        vINIT[i]=1;
                        v0[i]=vINIT[i];
                        v1[i]=vINIT[i];
                        iiii=1;
                    }

                    while(1)
                    {
                        printf("Enter new BETA value:\n");
                        scanf("%lf",&BETA);
                        fflush(stdin);//NOT STANDARD :(
                        rewind(stdin);//NOT STANDARD :(
                        printf("\nBETA=%lf - Is this OK?\nEnter 'y' or 'Y' to continue\nEnter 'a' or 'A' to try again.\n\n",BETA);
```

```

						char key;
						while(1)
						{
							key=_getch();
							if (key=='y'||key=='Y'||key=='a'||key=='A')
							{
								break;
							}
						}
						if (key=='y'||key=='Y')
						{
							break;
						}
					}
				}
				else if (key=='q'||key=='Q')
				{
					//Quits the program.
					printf("* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");
					printf("%i out of %i iterations were completed.\n",iiii,NUM_ITERATIONS_MAX);
					_fcloseall();
					exit(0);
				}
				else
				{
					printf("\n\nCONTINUE\n\n");
				}
			}
		}

		printf("\n         ITERATION=%i    V_FRAC=%f    BETA=%f\n",iiii,V_FRAC,BETA);

#ifndef SHORTCUT
		//Updates the Jacobian for each iteration of Newton's Method.
		for(int x=0;x<size;x++)
		{
			p[x]=DX*DX*HHP_PRIME[CN_X(x)][CN_Y(x)];
			J[x][x]=h[x][x]+p[x];
```

```
        }

        inverse(J,size,J_INVERSE);
#endif
        //Calculates the nonlinear equations that we are trying to solve to be zero.
        for(int i=0;i<size;i++)
        {
            eq[i]=0;
            for(int k=0;k<size;k++)
            {
                eq[i]+=A[i][k]*v0[k];
            }
            eq[i]+=-B[i]+HHP[CN_X(i)][CN_Y(i)]*DX*DX;
        }

        int s_max=0;
        //Newton's method.
        for(int i=0;i<size;i++)
        {
            //We could do better by not including the star in our solver, since the values are already known.
            //if((CN_X(i)>(N_S_X-1))||(CN_Y(i)>(N_S_Y-1)))
            {
                //Update all variables.
                v1[i]=0;
                for(int j=0;j<size;j++)
                {
                    v1[i]+=J_INVERSE[i][j]*eq[j];
                }
                v1[i]=v0[i]-v1[i];

                //Keeps track of changes between iterations.
                deltaALL+=fabs(v0[i]-v1[i]);
                CHANGE[CN_X(i)][CN_Y(i)]=fabs(v0[i]-v1[i]);
                if((CN_X(i)>(N_S_X-1))||(CN_Y(i)>(N_S_Y-1)))
                {
                    if (fabs(v0[i]-v1[i])>deltaMAX)
                    {
                        s_max=i;
                        deltaMAX=fabs(v0[i]-v1[i]);
                    }
                }
            }
        }
```

```
			printf("Max change at %i, or (%i,%i)\n",s_max,CN_X(s_max),CN_Y(s_max));
			printf("deltaALL=%f\n",deltaALL);
			printf("olddeltaALL=%f\n",olddeltaALL);
			printf("deltaMAX=%f\n",deltaMAX);

			//Similar effect to the relaxation parameter.
			for(int i=0;i<size;i++)
			{
				v0[i]=(V_FRAC)*(v1[i])+(1-V_FRAC)*(v0[i]);
			}

			//Automatically end if a certain convergence level is reached.
			//User is provided with options as to what to do next.
			if(iiii>=5&&deltaMAX<0.000001)
			{
				printf("STABLE BREAK\n");
				printf("\nWriting data to files............\n");
				writeToFiles();

				printf("\nData has been written to files.\n\nEnter 'r' or 'R' to reset the same simulation.\nEnter 'c' or 'C' to reset with another BETA value.\nEnter 'e' or 'E' to continue with another BETA value.\nEnter anything else to quit.\n");
				const char key=_getch();

				if (key=='r'||key=='R')
				{
					//Completely reset simulation.
					for(int i=0;i<size;i++)
					{
						vINIT[i]=1;
						v0[i]=vINIT[i];
						v1[i]=vINIT[i];
					}
					iiii=0;
				}
				else if (key=='c'||key=='C')
				{
					//Completely reset simulation with different BETA value.
					for(int i=0;i<size;i++)
					{
						vINIT[i]=1;
						v0[i]=vINIT[i];
						v1[i]=vINIT[i];
					}
```

```
				iiii=0;

				while(1)
				{
					printf("Enter new BETA value:\n");
					scanf("%lf",&BETA);
					fflush(stdin);//NOT STANDARD :(
					rewind(stdin);//NOT STANDARD :(
					printf("\nBETA=%lf - Is this OK?\nEnter 'y' or 'Y' to continue\nEnter 'a' or 'A' to try again.\n\n",BETA);

					char key;
					while(1)
					{
						key=_getch();
						if (key=='y'||key=='Y'||key=='a'||key=='A')
						{
							break;
						}
					}
					if (key=='y'||key=='Y')
					{
						break;
					}
				}
			}
			else if (key=='e'||key=='E')
			{
				//Continue simulation with different BETA value.
				iiii=0; //This is reset here so that the maximum iteration limit isn't reached prematurely if you switch BETA a lot.
				while(1)
				{
					printf("Enter new BETA value:\n");
					scanf("%lf",&BETA);
					fflush(stdin);//NOT STANDARD :(
					rewind(stdin);//NOT STANDARD :(
					printf("\nBETA=%lf - Is this OK?\nEnter 'y' or 'Y' to
```

```
                                    continue\nEnter 'a' or 'A'
                                     to try again.\n\n",BETA);

                                char key;
                                while(1)
                                {
                                        key=_getch();
                                        if (key=='y'||key=='Y
                                            '||key=='a'||key==
                                            'A')
                                        {
                                                break;
                                        }
                                }
                                if (key=='y'||key=='Y')
                                {
                                        break;
                                }
                        }
                }
                else
                {
                        //Quits the program.
                        printf("* * * * * * * * * * * * * * *
                             * * * * * * * * * * * * * * * *
                             * *\n");
                        printf("%i out of %i iterations were
                           completed.\n",iiii,
                           NUM_ITERATIONS_MAX);
                        _fcloseall();
                        exit(0);
                }
        }

        //Update HHP values.
        hhpSet(HHP,HHP_PRIME);
    }
}
```

F.1.5 equation_solver_2case.c

```
//Solves for PHI using Newton's method.
//Note: PHI is stored in v0 and v1.
void equationBuilder(int size)
{
        T vINIT[(N_MAX_X-2)*(N_MAX_Y-2)];

#ifndef READ
        for(int x=0;x<size;x++)
        {
                for(int y=0;y<size;y++)
                {
                        J[x][y]=A[x][y];
                        h[x][y]=A[x][y];
                }
        }
#endif

#ifdef SHORTCUT
#ifndef READ
        printf("pre inverse\n");
        inverse(J,size,J_INVERSE);
        printf("after inverse\n");
#endif
#endif

        //Our initial guess for every point is one
        for(int i=0;i<size;i++)
        {
                vINIT[i]=1.0;
                v0[i]=vINIT[i];
                v1[i]=vINIT[i];
        }

        /*
        //An alternate way of setting initial values.
        //Our initial guess for every point is "random" between 0 and
            1.
        srand(time(NULL));
        for(int i=0;i<size;i++)
        {
                vINIT[i]=((T)rand())/RAND_MAX;
                v0[i]=vINIT[i];
                v1[i]=vINIT[i];
        }
        */

        T olddeltaALL=0;
        T deltaALL=0;
        for(int iiii=1;iiii<=NUM_ITERATIONS_MAX;iiii++)
        {
```

```
			olddeltaALL=deltaALL;
			deltaALL=0;
			T deltaMAX=0;

			//This code controls user interaction. All user input is direct keyboard hits, NOT typing text into a command line.
			//To the best of my knowledge, this code to check for user input does NOT slow the program down significantly.
			if(_kbhit())
			{
				//This call to _getch() will not pause the program because the user already hit a key,
				//but in general the other calls will pause the program and wait for input.
				const char key=_getch();
				if (key=='b'||key=='B')
				{
					//Simulation is paused.
					printf("\nWriting data to files ............\n");
					writeToFiles();
					printf("\nData has been written to files.\n\nEnter 'w' or 'W' to change the FRAC parameter.\nEnter 'q' or 'Q' to quit.\nEnter anything else to continue.\n");

					const char key=_getch();
					if (key=='w'||key=='W')
					{
						//Yes, the code needs to be structured this way!
						while(1)
						{
							printf("Enter new V_FRAC value:\n");
							scanf("%lf",&V_FRAC);
							fflush(stdin);//NOT STANDARD :(
							rewind(stdin);//NOT STANDARD :(
							printf("\nV_FRAC=%lf - Is this OK?\nEnter 'y' or 'Y' to continue\nEnter 'a' or 'A' to try again.\n\n",V_FRAC);

```

```
				char key;
				while(1)
				{
					key=_getch();
					if (key=='y'||key=='Y'||key=='a'||key=='A')
					{
						break;
					}
				}
				if (key=='y'||key=='Y')
				{
					break;
				}
			}
		}
		else if (key=='q'||key=='Q')
		{
			//If you are at the first case of a double simulation, it switches to the second case.
			//Otherwise, it quits the program.
			if(!toggle)
			{
				printf("\n\n\nSwitching Case\n\n\n");
				toggle=1;
				iiii=0;
				for(int i=0;i<size;i++)
				{
					vINIT[i]=1.0;
					v0[i]=vINIT[i];
					v1[i]=vINIT[i];
				}
			}
			else
			{
				printf("* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
*\n");
                        printf("%i out of %i iterations were completed.\n",iiii,NUM_ITERATIONS_MAX);
                        _fcloseall();
                        exit(0);
                    }
                }
                else
                {
                    printf("\n\nCONTINUE\n\n");
                }
            }
        }

        printf("\n         ITERATION=%i    V_FRAC=%f\n",iiii,V_FRAC);

#ifndef SHORTCUT
        //Updates the Jacobian for each iteration of Newton's Method.
        for(int x=0;x<size;x++)
        {
            p[x]=DX*DX*HHP_PRIME[CN_X(x)][CN_Y(x)];
            J[x][x]=h[x][x]+p[x];
        }

        inverse(J,size,J_INVERSE);
#endif
        //Calculates the nonlinear equations that we are trying to solve to be zero.
        for(int i=0;i<size;i++)
        {
            eq[i]=0;
            for(int k=0;k<size;k++)
            {
                eq[i]+=A[i][k]*v0[k];
            }
            eq[i]+=-B[i]+HHP[CN_X(i)][CN_Y(i)]*DX*DX;
        }

        int s_max=0;
        //Newton's method.
        for(int i=0;i<size;i++)
        {
            //We could do better by not including the star in our solver, since the values are already known.
            //if((CN_X(i)>(N_S-1))||(CN_Y(i)>(N_S-1)))
```

```
            {
                //Update all variables.
                v1[i]=0;
                for(int j=0;j<size;j++)
                {
                    v1[i]+=J_INVERSE[i][j]*eq[j];
                }
                v1[i]=v0[i]-v1[i];

                //Keeps track of changes between iterations.
                deltaALL+=fabs(v0[i]-v1[i]);
                CHANGE[CN_X(i)][CN_Y(i)]=fabs(v0[i]-v1[i]);
                if((CN_X(i)>(N_S_X-1))||(CN_Y(i)>(N_S_Y-1)))
                {
                    if (fabs(v0[i]-v1[i])>deltaMAX)
                    {
                        s_max=i;
                        deltaMAX=fabs(v0[i]-v1[i]);
                    }
                }
            }
        }
        printf("Max change at %i, or (%i,%i)\n",s_max,CN_X(s_max),CN_Y(s_max));
        printf("deltaALL=%f\n",deltaALL);
        printf("olddeltaALL=%f\n",olddeltaALL);
        printf("deltaMAX=%f\n",deltaMAX);

        //Similar effect to the relaxation parameter.
        for(int i=0;i<size;i++)
        {
            v0[i]=(V_FRAC)*(v1[i])+(1-V_FRAC)*(v0[i]);
        }

        //Automatically end/toggle if a certain convergence level is reached.
        if(iiii>=5&&deltaMAX<0.000001)
        {
            if(!toggle)
            {
                printf("\n\n\nSwitching Case\n\n\n");
                toggle=1;
                iiii=0;
                for(int i=0;i<size;i++)
                {
                    vINIT[i]=1.0;
                    v0[i]=vINIT[i];
```

```
                                        v1[i]=vINIT[i];
                                }
                        }
                        else
                        {
                                printf("STABLE BREAK\n");
                                printf("\nWriting data to files
                                    ............\n");
                                writeToFiles();
                                break;
                        }
                }

                //Update HHP values.
                hhpSet(HHP,HHP_PRIME);
        }
}
```

F.1.6 reset_ckf.c

```c
void InsertionSort(void)
{
	//This is the common "insertion sort" algorithm for sorting an array. Here, we sort L_X in descending order, and carry L_Y "along for the ride."
	//This means that L_X vs. L_Y pairs are preserved. We are simply ordering the pairs based on L_X values.
	int i,j;
	T tempX,tempY;
	for(j=1;j<(N_MAX_X-2)*(N_MAX_Y-2);j++)// Start with 1, not 0, because first value in any array is "automatically" smaller than everything before it.
	{
		//At every loop iteration, we consider the value L_X[j]. Everything before this value is already in descending order.
		tempX = L_X[j];//Store current value in consideration.
		tempY = L_Y[j];//All sorting is done with L_X values only, not L_Y.
		for(i=j-1;(i>=0)&&(L_X[i]<tempX);i--)//Values smaller than tempX rise, and tempX "sinks" until it hits a value greater than itself.
		{
			L_X[i+1] = L_X[i];
			L_Y[i+1] = L_Y[i];
		}
		L_X[i+1] = tempX;
		L_Y[i+1] = tempY;
	}
	return;
}

void resetCKF(void)
{
	int size=(N_MAX_X-2)*(N_MAX_Y-2);
	//We need to get PSI vs. HHP so we can take integral via trapezoidal method.
	for(int i=0;i<size;i++)
	{
		if(CN_Y(i)>0)
		{
			L_X[i]=v0[i];
			L_Y[i]=fabs(HHP[CN_X(i)][CN_Y(i)]); //Do we really need to take the absolute value.
		}
	}
	InsertionSort(); //This makes L_X sorted in DESCENDING order (L_X[0] is largest, L_X[size-1] is smallest).

```

```
//Bottom Edge (past light cylinder).
for(int s=N_LC;s<=N_MAX_X-4;s++)
{
	T sum=0;
	if(v0[s]<P_OP)
	{
		for(int i=size-1;i>0;i--)
		{
			if(L_X[i]>v0[s]){break;}
			sum+=0.5*(L_X[i-1]-L_X[i])*(L_Y[i]+L_Y[i-1]);
		}
		sum=fabs(sum);//This ensures that the term under the square root is positive. We shouldn't really need this...
	}

	//Note: H is negative, so we want -sqrt(H^2).
	bB1[s]=0;
	bB2[s]=0;
	bB3[s]=-1;
	bB4[s]=0;
	bB5[s]=1;
	bB6[s]=-DY*sqrt(2*sum/((s+1)*(s+1)*DX*DX-1));

	L_func[s]=-DY*sqrt(2*sum/((s+1)*(s+1)*DX*DX-1)); // This is used just so that we can add the bottom boundary to plots.
}

//Bottom Right Corner.
int s=N_MAX_X-3;
T sum=0;
if(v0[s]<P_OP)
{
	for(int i=size-1;i>0;i--)
	{
		if(L_X[i]>v0[s]){break;}
		sum+=0.5*(L_X[i-1]-L_X[i])*(L_Y[i]+L_Y[i-1]);
	}
}

int t=0;
int i=s+1;
int j=t+1;

//Note: H is negative, so we want -sqrt(H^2).
bR1[t]=-i;
bR2[t]=i;
bR3[t]=0;
bR4[t]=j;
bR5[t]=-j;
```

```
        bR6[t]=DY*j*sqrt(2*sum/(i*i*DX*DX-1));

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=-1;
        bB4[s]=0;
        bB5[s]=1;
        bB6[s]=-DY*sqrt(2*sum/(i*i*DX*DX-1));

        L_func[s]=-DY*sqrt(2*sum/((s+1)*(s+1)*DX*DX-1));

        create_matrix();

        for(int x=0;x<size;x++)
        {
                for(int y=0;y<size;y++)
                {
                        J[x][y]=A[x][y];
                        h[x][y]=A[x][y];
                }
        }
}
```

F.1.7 reset_tak.c

```
void resetTak(void)
{
        T size=(N_MAX_X-2)*(N_MAX_Y-2);

        //Bottom Edge (past light cylinder).
        for(int s=N_LC;s<=N_MAX_X-4;s++)
        {
                int t=0;
                //Note: H is negative, so we want -sqrt(H^2).
                if(v0[RN(s+1,t+1)]>P_OP){v0[RN(s+1,t+1)]=P_OP;}
                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=-1;
                bB4[s]=0;
                bB5[s]=1;
                bB6[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/
                    P_OP)*(pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt
                    ((s+1)*(s+1)*DX*DX-1);

                //This is used just so that we can add the bottom
                    boundary to plots.
                L_func[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)
                    ]/P_OP)*(pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/
                    sqrt((s+1)*(s+1)*DX*DX-1);
        }

        //Bottom Right Corner.
        int s=N_MAX_X-3;
        int t=0;
        int i=s+1;
        int j=t+1;

        //Note: H is negative, so we want -sqrt(H^2).
        if(v0[RN(s+1,t+1)]>P_OP){v0[RN(s+1,t+1)]=P_OP;}
        bR1[t]=-i;
        bR2[t]=i;
        bR3[t]=0;
        bR4[t]=j;
        bR5[t]=-j;
        bR6[t]=DY*j*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/P_OP)*(
            pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt((s+1)*(s+1)*DX
            *DX-1);

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=-1;
        bB4[s]=0;
        bB5[s]=1;
        bB6[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/P_OP)*(pow
            (fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt((s+1)*(s+1)*DX*DX
```

```
        -1);

    //This is used just so that we can add the bottom boundary to
        plots.
    L_func[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/P_OP)*(
        pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt((s+1)*(s+1)*DX
        *DX-1);

    create_matrix();

    for(int x=0;x<size;x++)
    {
        for(int y=0;y<size;y++)
        {
            J[x][y]=A[x][y];
            h[x][y]=A[x][y];
        }
    }
}
```

F.1.8 read_matrix.c

```c
void read(void)
{
        rewind(fp_j2);

        int size=(N_MAX_X-2)*(N_MAX_Y-2);
        for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
        {
                printf("%i out of %i\n",j+1,size);
                for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
                {
                        fscanf(fp_j2, "%lf ",&J_INVERSE[j][k]);
                }
        }

        fclose(fp_j2);
}
```

F.2 Case Files

This section includes the source code for each case file. Every case file is structured the same way, with the discretization choices first, then the boundary conditions, then the choice of star, and finally the method for setting $F(\psi)$ and any other nonhomogeneous terms. Our hope in including these case files is that readers can understand the particular choices made in crafting the equations and the boundaries, along with the subtle decisions that a simulator must make. We hope that by providing every line of code we will answer any questions about the work done and will aid those who wish to pursue the work further.

Single simulation cases:

- ckf_monopole.c - Pure Monopole case with the CKF method.
- ckf.c - Pure CKF case from Contopoulos et al. 1999.
- ckf_jets.c - CKF case mixed with Jets case.
- ckf_null.c - Null Sheet case with the CKF method.
- tak_monopole_test.c - Test of the Monopole case using the known answer.
- tak_monopole.c - Pure Monopole case with the TOTS method.
- tak_monopole_jets.c - Monopole case mixed with Jets case.
- tak.c - Pure TOTS case from Takamori et al. 2012.
- tak_jets.c - TOTS case mixed with Jets case.
- tak_theory_jets.c - Theoretical Jets case.
- tak_null.c - Null Sheet case with the TOTS method.

Double simulation cases:

- ckf_monopole_jets.c - CKF Monopole, then Jets.
- ckf_tak.c - CKF, then TOTS.
- ckf_null_tak.c - CKF Null Sheet, then TOTS.
- tak_ckf.c - TOTS, then CKF.
- tak_ckf_jets.c - TOTS, then CKF Jets.
- tak_ckf_null.c - TOTS, then CKF Null Sheet.
- tak_theory_jets_ckf.c - TOTS Theoretical Jets case, then CKF.

F.2.1 ckf_monopole.c

```c
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE MONOPOLE
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```
			a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
			a6[s][t]=0;

			//Value near LC is average of values to the left and right.
			if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
			{
				a1[s][t]=-0.5;
				a2[s][t]=-0.5;
				a3[s][t]=0;
				a4[s][t]=0;
				a5[s][t]=1;
				a6[s][t]=0;
			}

			/*
			//Unused
			//Polynomial coefficients of HHP (constant term taken care of in function "f")
			//c[#] is the coefficient for (Pij)^(#+1)
			c[0][s][t]=4;
			c[1][s][t]=-6;
			c[2][s][t]=2;
			*/
		}
	}

	//Left Boundary P=0
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=0;

		bL1[t]=1;
		bL2[t]=0;
		bL3[t]=0;
		bL4[t]=0;
		bL5[t]=0;
		bL6[t]=0;
	}

	//Right Boundary RPr+ZPz=0
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=N_MAX_X-3;

		bR1[t]=-i;
		bR2[t]=i;
		bR3[t]=-j;
		bR4[t]=j;
		bR5[t]=0;
		bR6[t]=0;
```

```
        }

        //Bottom Boundary P=P_OP
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;
```

```
                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
```

```
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void){}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        T HHP_L[N_MAX_Y-2];
        for(int KP=0;KP<N_MAX_Y-2;KP++)
        {
                HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
        }

        //At r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder).
        //For 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.

        for(int jj=0;jj<=N_LC-2;jj++)
        {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        {
                                const T PT=v0[RN(jj,kk)];

                                if(PT<v0[RN(N_LC-2,0)])
                                {
                                        if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                                        {
                                                HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                                        }
                                        else
                                        {
                                                for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                                                const T Q1=PT-v0[RN(N_LC-1,KP+1)];
```

```
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                    }
                }
                else
                {
                    HHP[jj][kk]=0;
                }
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
    }

    for(int jj=N_LC-1;jj==N_LC-1;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            HHP[N_LC-1][kk]=HHP_L[kk];
        }
    }

    for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            int KP=0;
            const T PT=v0[RN(jj,kk)];

            for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

            const T Q1=PT-v0[RN(N_LC-1,KP+1)];
            const T Q2=PT-v0[RN(N_LC-1,KP)];
            HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
    }

    for(int jj=N_LC-1;jj==N_LC-1;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            v0[RN(N_LC-1,kk)]=0.5*(v0[RN(N_LC-2,kk)]+v0[RN(N_LC,kk)]);
        }
    }

```

```
	//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			HHP_PRIME[x][y]=0.1*sqrt(x*x+y*y);
		}
	}
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0;
}

#undef i
#undef j
```

F.2.2 ckf.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE STANDARD
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
                    a3[s][t]=1-i*i*DX*DX;
                    a4[s][t]=1-i*i*DX*DX;
                    a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                    a6[s][t]=0;
                    */

                    //1st derivative is backward difference, DX and DY independent
                    a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                    a2[s][t]=1-i*i*DX*DX;
                    a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                    a6[s][t]=0;

                    //Value near LC is average of values to the left and right.
                    if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                    {
                            a1[s][t]=-0.5;
                            a2[s][t]=-0.5;
                            a3[s][t]=0;
                            a4[s][t]=0;
                            a5[s][t]=1;
                            a6[s][t]=0;
                    }

                    /*
                    //Unused
                    //Polynomial coefficients of HHP (constant term taken care of in function "f")
                    //c[#] is the coefficient for (Pij)^(#+1)
                    c[0][s][t]=0;
                    c[1][s][t]=0;
                    c[2][s][t]=0;
                    */
            }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=0;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;
    }
```

```

//Right Boundary RPr+ZPz=0
for(int t=1;t<=N_MAX_Y-4;t++)
{
        int s=N_MAX_X-3;

        bR1[t]=-i;
        bR2[t]=i;
        bR3[t]=-j;
        bR4[t]=j;
        bR5[t]=0;
        bR6[t]=0;
}

//Bottom Boundary
//Outside star, inside light cylinder Pz=0
for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
{
        int t=0;

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=1;
        bB4[s]=0;
        bB5[s]=-1;
        bB6[s]=0;
}

//Outside light cylinder P=P_OP which is specified from the
   start.
for(int s=N_LC-1;s<=N_MAX_X-4;s++)
{
        int t=0;

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=1;
        bB4[s]=0;
        bB5[s]=0;
        bB6[s]=P_OP;
}

//Top Boundary RPr+ZPz=0
for(int s=1;s<=N_MAX_X-4;s++)
{
        int t=N_MAX_Y-3;

        bT1[s]=-i;
        bT2[s]=i;
        bT3[s]=-j;
        bT4[s]=j;
        bT5[s]=0;
```

```
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
```

```
		bR5[t]=0;
		bR6[t]=j*P_OP;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=0;
		bB6[s]=P_OP;
	}

	//Top Right Corner
	{
		int s=N_MAX_X-3;
		int t=N_MAX_Y-3;

		bR1[t]=-1;
		bR2[t]=1;
		bR3[t]=0;
		bR4[t]=0;
		bR5[t]=0;
		bR6[t]=0;

		bT1[s]=0;
		bT2[s]=0;
		bT3[s]=-1;
		bT4[s]=1;
		bT5[s]=0;
		bT6[s]=0;
	}
}

//This function inserts the star.
void star(void)
{
	for(int s=0;s<N_S_X;s++)
	{
		const T R=DX*i;
		for(int t=0;t<N_S_Y;t++)
		{
			const T Z=DY*j;
			a1[s][t]=0;
			a2[s][t]=0;
			a3[s][t]=0;
			a4[s][t]=0;
			a5[s][t]=1;
			a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
		}
	}
}
```

```
//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	T HHP_L[N_MAX_Y-2];
	for(int KP=0;KP<N_MAX_Y-2;KP++)
	{
		HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
	}

	for(int jj=N_LC-1;jj==N_LC-1;jj++)
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			v0[RN(N_LC-1,kk)]=0.5*(v0[RN(N_LC-2,kk)]+v0[RN(N_LC,kk)]);
		}
	}

	//at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
	//for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
	for(int jj=0;jj<=N_S_X-1;jj++)
	{
		for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
		{
			int KP=0;
			{
				const T PT=v0[RN(jj,kk)];

				if(PT<v0[RN(N_LC-2,0)])
				{
					if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
					{
						HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
					}
					else
					{
						for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

						const T Q1=PT-v0[RN(N_LC-1,KP+1)];
```

```
						const T Q2=PT-v0[RN(N_LC-1,KP)];
						HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
					}
				}
				else
				{
					HHP[jj][kk]=0;
				}
			}
			if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
		}
	}

	//for NR_S<r<rLC
	for(int jj=N_S_X;jj<=N_LC-2;jj++)
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			int KP=0;
			{
				const T PT=v0[RN(jj,kk)];

				if(PT<v0[RN(N_LC-2,0)])
				{
					if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
					{
						HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
					}
					else
					{
						for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

						const T Q1=PT-v0[RN(N_LC-1,KP+1)];
						const T Q2=PT-v0[RN(N_LC-1,KP)];
						HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
					}
				}
				else
				{
```

```
                                        HHP[jj][kk]=0;
                                }
                        }
                        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
                }
        }

        for(int jj=N_LC-1;jj==N_LC-1;jj++)
        {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                        HHP[N_LC-1][kk]=HHP_L[kk];
                }
        }

        for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
        {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        const T PT=v0[RN(jj,kk)];

                        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        //if(fabs(Q1-Q2)>0.00001)
                        {
                                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                        }
                        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
                }
        }

        //TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
        for(int x=0;x<N_MAX_X-2;x++)
        {
                for(int y=0;y<N_MAX_Y-2;y++)
                {
                        T part1,part2;
                        if(x==N_MAX_X-3)
                        {
                                part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
                        }
                        else
                        {
                                part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
```

```
			}
			if(y==N_MAX_Y-3)
			{
				part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(x,y)]-v0[RN(x,y-1)])/(DY*DY);
			}
			else
			{
				part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(x,y+1)]-v0[RN(x,y)])/(DY*DY);
			}
			HHP_PRIME[x][y]=part1+part2;
		}
	}
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0;
}

#undef i
#undef j
```

F.2.3 ckf_jets.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE STANDARD
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                        a6[s][t]=0;

                        //Value near LC is average of values to the left and right.
                        if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                        {
                                a1[s][t]=-0.5;
                                a2[s][t]=-0.5;
                                a3[s][t]=0;
                                a4[s][t]=0;
                                a5[s][t]=1;
                                a6[s][t]=0;
                        }

                        /*
                        //Unused
                        //Polynomial coefficients of HHP (constant term taken care of in function "f")
                        //c[#] is the coefficient for (Pij)^(#+1)
                        c[0][s][t]=0;
                        c[1][s][t]=0;
                        c[2][s][t]=0;
                        */
                }
        }

        //Left Boundary P=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;
        }

        //Right Boundary
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;
#ifdef JETS1
                //RPr+ZPz=0
                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
```

```
                bR6[t]=0;
#endif
#ifdef JETS2
                //Prr=0
                bR1[t]=1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=-2;
                bR6[t]=0;
#endif
        }

        //Bottom Boundary
        //Outside star, inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=-1;
                bB6[s]=0;
        }

        //Outside light cylinder P=P_OP which is specified from the start.
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;
#ifdef JETS1
                // RPr+ZPz=0
                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
```

```
                bT6[s]=0;
#endif
#ifdef JETS2
                //Prr=0
                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=1;
                bT4[s]=1;
                bT5[s]=-2;
                bT6[s]=0;
#endif
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

#ifdef JETS1
                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
```

```
#endif
#ifdef JETS2
                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=1;
                bT4[s]=1;
                bT5[s]=-2;
                bT6[s]=0;
#endif
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

#ifdef JETS1
                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;
#endif
#ifdef JETS2
                bR1[t]=1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=-2;
                bR6[t]=0;
#endif

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

#ifdef JETS1
                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
```

```
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
#endif
#ifdef JETS2
                bR1[t]=1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=-2;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=1;
                bT4[s]=1;
                bT5[s]=-2;
                bT6[s]=0;
#endif
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        //a6[s][t]=R*R/pow(R*R+Z*Z,1.5); //Typical
                            choice for the star.
                        a6[s][t]=1.0/pow(R*R+Z*Z,0.5); //Lovelace
                            used this choice instead for the jets case
                            .
                }
        }
}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
```

```
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        T KH=2.48;
        //T BETA=0.9995;

        T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
        //T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));
        //T BETA=2.0;
        printf("BETA=%f\n",BETA);

        T HHP_L[N_MAX_Y-2];
        for(int KP=0;KP<N_MAX_Y-2;KP++)
        {
                HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
        }

        //at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
        //for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
        for(int jj=0;jj<=N_S_X-1;jj++)
        {
                for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        {
                                const T PT=v0[RN(jj,kk)];

                                if(PT<v0[RN(N_LC-2,0)])
                                {
                                        if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                                        {
                                                const T PS=PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                                                HHP[jj][kk]=KH*KH*(PS-0.5*BETA*PS*PS)*(1-BETA*PS);
                                        }
                                        else if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                                        {
                                                HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                                        }
```

```
                else
                {
                    for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                    const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                    const T Q2=PT-v0[RN(N_LC-1,KP)];
                    HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                }
            }
            else
            {
                HHP[jj][kk]=0;
            }
        }
        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
    }
}

//for NR_S<r<rLC
for(int jj=N_S_X;jj<=N_LC-2;jj++)
{
    for(int kk=0;kk<=N_MAX_Y-3;kk++)
    {
        int KP=0;
        {
            const T PT=v0[RN(jj,kk)];

            if(PT<v0[RN(N_LC-2,0)])
            {
                if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                {
                    const T PS=PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                    HHP[jj][kk]=KH*KH*(PS-0.5*BETA*PS*PS)*(1-BETA*PS);
                }
                else if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                {
                    HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
```

```
                    }
                    else
                    {
                        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                    }
                }
                else
                {
                    HHP[jj][kk]=0;
                }
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
    }

    for(int jj=N_LC-1;jj==N_LC-1;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            HHP[N_LC-1][kk]=HHP_L[kk];
        }
    }

    for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            int KP=0;
            const T PT=v0[RN(jj,kk)];

            for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

            const T Q1=PT-v0[RN(N_LC-1,KP+1)];
            const T Q2=PT-v0[RN(N_LC-1,KP)];
            //if(fabs(Q1-Q2)>0.00001)
            {
                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
```

```
        }

        //TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
        for(int x=0;x<N_MAX_X-2;x++)
        {
                for(int y=0;y<N_MAX_Y-2;y++)
                {
                        T part1,part2;
                        if(x==N_MAX_X-3)
                        {
                                part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
                        }
                        else
                        {
                                part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
                        }
                        if(y==N_MAX_Y-3)
                        {
                                part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(x,y)]-v0[RN(x,y-1)])/(DY*DY);
                        }
                        else
                        {
                                part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(x,y+1)]-v0[RN(x,y)])/(DY*DY);
                        }
                        HHP_PRIME[x][y]=part1+part2;
                }
        }
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
        return 0;
}

#undef i
#undef j
```

F.2.4 ckf_null.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE NULLSHEET
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
            a3[s][t]=1-i*i*DX*DX;
            a4[s][t]=1-i*i*DX*DX;
            a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
            a6[s][t]=0;
            */

            //1st derivative is backward difference, DX and DY independent
            a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
            a2[s][t]=1-i*i*DX*DX;
            a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
            a6[s][t]=0;

            //Value near LC is average of values to the left and right.
            if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
            {
                a1[s][t]=-0.5;
                a2[s][t]=-0.5;
                a3[s][t]=0;
                a4[s][t]=0;
                a5[s][t]=1;
                a6[s][t]=0;
            }

            /*
            //Unused
            //Polynomial coefficients of HHP (constant term taken care of in function "f")
            //c[#] is the coefficient for (Pij)^(#+1)
            c[0][s][t]=0;
            c[1][s][t]=0;
            c[2][s][t]=0;
            */
        }
    }

    //The bottom right coner and bottom edge past the light cylinder are changed elsewhere, so the choices for these are largely unimportant.
    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
        int s=0;

        bL1[t]=1;
        bL2[t]=0;
        bL3[t]=0;
        bL4[t]=0;
```

```
                bL5[t]=0;
                bL6[t]=0;
        }

        //Right Boundary RPr+ZPz=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=0;
        }

        //Bottom Boundary
        //Outside star, inside light cylinder Pz=0
        for(int s=1;((s<=N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=-1;
                bB6[s]=0;
        }

        //Outside light cylinder H^2=(R^2-1)*(Pz)^2
        //To begin the simulation, just use P=P_OP
        for(int s=N_LC;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
```

```
			bT3[s]=-j;
			bT4[s]=j;
			bT5[s]=0;
			bT6[s]=0;
		}

		//Bottom Left Corner
		{
			int s=0;
			int t=0;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;

			bB1[s]=0;
			bB2[s]=0;
			bB3[s]=1;
			bB4[s]=0;
			bB5[s]=0;
			bB6[s]=P_OP;
		}

		//Top Left Corner
		{
			int s=0;
			int t=N_MAX_Y-3;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;

			bT1[s]=0;
			bT2[s]=i;
			bT3[s]=-j;
			bT4[s]=j;
			bT5[s]=0;
			bT6[s]=0;
		}

		//Bottom Right Corner
		{
			int s=N_MAX_X-3;
			int t=0;

			bR1[t]=-i;
```

```
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
                }
        }
```

```
}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        T HHP_L[N_MAX_Y-2];
        for(int KP=0;KP<N_MAX_Y-2;KP++)
        {
                HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
                    N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
        }

        //at r<rL, there are many possible cases (closed field lines,
            open field lines that do or do not cross the light
            cylinder)
        //for 0<r<=NR_S, we stay OUTSIDE star, because the star value
            is known and does not need to be altered.
        for(int jj=0;jj<=N_S_X-1;jj++)
        {
                for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        {
                                const T PT=v0[RN(jj,kk)];

                                if(PT<v0[RN(N_LC-2,0)])
                                {
                                        if(PT<v0[RN(N_LC-2,N_MAX_Y-3)
                                            ])
                                        {
                                                HHP[jj][kk]=HHP_L[
                                                    N_MAX_Y-3]*PT/v0[
                                                    RN(N_LC-2,N_MAX_Y
                                                    -3)];
                                        }
                                        else
                                        {
                                                for(;KP<N_MAX_Y-4&&PT
                                                    <v0[RN(N_LC-2,KP)
                                                    ];KP++);

                                                const T Q1=PT-v0[RN(
                                                    N_LC-1,KP+1)];
                                                const T Q2=PT-v0[RN(
                                                    N_LC-1,KP)];
                                                HHP[jj][kk]=(Q1*HHP_L
                                                    [KP]-Q2*HHP_L[KP
                                                    +1])/(Q1-Q2);
                                        }
                                }
                                else
```

```
                {
                    HHP[jj][kk]=0;
                }
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
    }

    //for NR_S<r<rLC
    for(int jj=N_S_X;jj<=N_LC-2;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            int KP=0;
            {
                const T PT=v0[RN(jj,kk)];

                if(PT<v0[RN(N_LC-2,0)])
                {
                    if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                    {
                        HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                    }
                    else
                    {
                        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                    }
                }
                else
                {
                    HHP[jj][kk]=0;
                }
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
    }

    for(int jj=N_LC-1;jj==N_LC-1;jj++)
```

```
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			HHP[N_LC-1][kk]=HHP_L[kk];
		}
	}

	for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			int KP=0;
			const T PT=v0[RN(jj,kk)];

			for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

			const T Q1=PT-v0[RN(N_LC-1,KP+1)];
			const T Q2=PT-v0[RN(N_LC-1,KP)];
			//if(fabs(Q1-Q2)>0.00001)
			{
				HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
			}
			if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
		}
	}

	//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			T part1,part2;
			if(x==N_MAX_X-3)
			{
				part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
			}
			else
			{
				part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
			}
			if(y==N_MAX_Y-3)
			{
				part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(x,y)]-v0[RN(x,y-1)])/(DY*DY);
			}
			else
			{
```

```
                                        part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                                            x,y+1)]-v0[RN(x,y)])/(DY*DY);
                              }
                              HHP_PRIME[x][y]=part1+part2;
                    }
          }

          resetCKF();
}

//This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
          return 0;
}

#undef i
#undef j
```

F.2.5 tak_monopole_test.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE MONOPOLE
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```
                    a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
                        -1/i-i*DX*DX;
                    a6[s][t]=0;

                    //Value near LC is average of values to the
                        left and right.
                    if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                    {
                            a1[s][t]=-0.5;
                            a2[s][t]=-0.5;
                            a3[s][t]=0;
                            a4[s][t]=0;
                            a5[s][t]=1;
                            a6[s][t]=0;
                    }

                    //Polynomial coefficients of HHP (constant
                        term taken care of in function "f")
                    //c[#] is the coefficient for (Pij)^(#+1)
                    c[0][s][t]=0;
                    c[1][s][t]=0;
                    c[2][s][t]=0;
            }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=0;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;
    }

    //Right Boundary RPr+ZPz=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=N_MAX_X-3;

            bR1[t]=-i;
            bR2[t]=i;
            bR3[t]=-j;
            bR4[t]=j;
            bR5[t]=0;
            bR6[t]=0;
    }

    //Bottom Boundary P=P_OP
```

```
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
```

```
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
```

```
}

//This function inserts the star.
void star(void){}

__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*
			    v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x
			    ,y)]*v0[RN(x,y)]*v0[RN(x,y)];
			HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN
			    (x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)
			    ];
		}
	}
}

//This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	T R=m*DX;
	T Z=n*DY;
	T Q=P_OP*(1-Z/sqrt(Z*Z+R*R));
	return 2*Q*(Q/P_OP-1)*(Q/P_OP-2);
}

#undef i
#undef j
```

F.2.6 tak_monopole.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE MONOPOLE
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```
                    a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                    a6[s][t]=0;
                    */

                    //1st derivative is backward difference, DX and DY independent
                    a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                    a2[s][t]=1-i*i*DX*DX;
                    a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                    a6[s][t]=0;

                    //Value near LC is average of values to the left and right.
                    if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                    {
                            a1[s][t]=-0.5;
                            a2[s][t]=-0.5;
                            a3[s][t]=0;
                            a4[s][t]=0;
                            a5[s][t]=1;
                            a6[s][t]=0;
                    }

                    //Polynomial coefficients of HHP (constant term taken care of in function "f")
                    //c[#] is the coefficient for (Pij)^(#+1)
                    c[0][s][t]=4;
                    c[1][s][t]=-6;
                    c[2][s][t]=2;
            }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=0;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;
    }

    //Right Boundary RPr+ZPz=0
    for(int t=1;t<=N_MAX_Y-4;t++)
```

```
	{
		int s=N_MAX_X-3;

		bR1[t]=-i;
		bR2[t]=i;
		bR3[t]=-j;
		bR4[t]=j;
		bR5[t]=0;
		bR6[t]=0;
	}

	//Bottom Boundary P=P_OP
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		int t=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=0;
		bB6[s]=P_OP;
	}

	//Top Boundary RPr+ZPz=0
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		int t=N_MAX_Y-3;

		bT1[s]=-i;
		bT2[s]=i;
		bT3[s]=-j;
		bT4[s]=j;
		bT5[s]=0;
		bT6[s]=0;
	}

	//Bottom Left Corner
	{
		int s=0;
		int t=0;

		bL1[t]=1;
		bL2[t]=0;
		bL3[t]=0;
		bL4[t]=0;
		bL5[t]=0;
		bL6[t]=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
```

```
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
    }

    //Top Left Corner
    {
            int s=0;
            int t=N_MAX_Y-3;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;

            bT1[s]=0;
            bT2[s]=i;
            bT3[s]=-j;
            bT4[s]=j;
            bT5[s]=0;
            bT6[s]=0;
    }

    //Bottom Right Corner
    {
            int s=N_MAX_X-3;
            int t=0;

            bR1[t]=-i;
            bR2[t]=i;
            bR3[t]=0;
            bR4[t]=j;
            bR5[t]=0;
            bR6[t]=j*P_OP;

            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
    }

    //Top Right Corner
    {
            int s=N_MAX_X-3;
            int t=N_MAX_Y-3;

            bR1[t]=-1;
            bR2[t]=1;
```

```
			bR3[t]=0;
			bR4[t]=0;
			bR5[t]=0;
			bR6[t]=0;

			bT1[s]=0;
			bT2[s]=0;
			bT3[s]=-1;
			bT4[s]=1;
			bT5[s]=0;
			bT6[s]=0;
		}
}

//This function inserts the star.
void star(void){}

//__forceinline void hhpSet(T HHP[(N_MAX_X-2)][(N_MAX_Y-2)],T HHP_PRIME[(N_MAX_X-2)][(N_MAX_Y-2)])
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
		for(int x=0;x<N_MAX_X-2;x++)
		{
			for(int y=0;y<N_MAX_Y-2;y++)
			{
				HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]*v0[RN(x,y)];
				HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
			}
		}
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
		return 0.0;
}

#undef i
#undef j
```

F.2.7 tak_monopole_jets.c

```c
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE MONOPOLE
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
	//Equation coefficients
	for(int s=0;s<=N_MAX_X-3;s++)
	{
		for(int t=0;t<=N_MAX_Y-3;t++)
		{
			/*
			//These equations represent an alternate finite difference choice for the first derivative.
			//The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
			//1st derivative is central difference
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=1-i*i*DX*DX;
			a4[s][t]=1-i*i*DX*DX;
			a5[s][t]=-4+4*i*i*DX*DX;
			a6[s][t]=0;
			*/

			/*
			//1st derivative is backward difference
			a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
			a2[s][t]=1-i*i*DX*DX;
			a3[s][t]=1-i*i*DX*DX;
			a4[s][t]=1-i*i*DX*DX;
			a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
			a6[s][t]=0;
			*/

			//1st derivative is backward difference, DX and DY independent
			a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
			a2[s][t]=1-i*i*DX*DX;
			a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```
                a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
                    -1/i-i*DX*DX;
                a6[s][t]=0;

                //Value near LC is average of values to the
                    left and right.
                if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                {
                        a1[s][t]=-0.5;
                        a2[s][t]=-0.5;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=0;
                }

                //Polynomial coefficients of HHP (constant
                    term taken care of in function "f")
                //c[#] is the coefficient for (Pij)^(#+1)
                c[0][s][t]=4;
                c[1][s][t]=-6;
                c[2][s][t]=2;
        }
}

//Left Boundary P=0
for(int t=1;t<=N_MAX_Y-4;t++)
{
        int s=0;

        bL1[t]=1;
        bL2[t]=0;
        bL3[t]=0;
        bL4[t]=0;
        bL5[t]=0;
        bL6[t]=0;
}

//Right Boundary RPr+ZPz=0
for(int t=1;t<=N_MAX_Y-4;t++)
{
        int s=N_MAX_X-3;

        bR1[t]=-i;
        bR2[t]=i;
        bR3[t]=-j;
        bR4[t]=j;
        bR5[t]=0;
        bR6[t]=0;
}

//Bottom Boundary P=P_OP
```

```
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
```

```
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;

            bT1[s]=0;
            bT2[s]=i;
            bT3[s]=-j;
            bT4[s]=j;
            bT5[s]=0;
            bT6[s]=0;
        }

        //Bottom Right Corner
        {
            int s=N_MAX_X-3;
            int t=0;

            bR1[t]=-i;
            bR2[t]=i;
            bR3[t]=0;
            bR4[t]=j;
            bR5[t]=0;
            bR6[t]=j*P_OP;

            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
        }

        //Top Right Corner
        {
            int s=N_MAX_X-3;
            int t=N_MAX_Y-3;

            bR1[t]=-1;
            bR2[t]=1;
            bR3[t]=0;
            bR4[t]=0;
            bR5[t]=0;
            bR6[t]=0;

            bT1[s]=0;
            bT2[s]=0;
            bT3[s]=-1;
            bT4[s]=1;
            bT5[s]=0;
            bT6[s]=0;
        }
```

```
}

//This function inserts the star.
void star(void){}

__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			if(v0[RN(x,y)]>=P_OP)
			{
				HHP[x][y]=0;
			}
			else if(v0[RN(x,y)]<v0[RN(N_LC-2,N_MAX_Y-3)])
			{
				T KH=2.48;
				//T BETA=0.9995;
				//T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
				//T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));
				HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA*v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x,y)]/P_OP-2);
			}
			else
			{
				HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]*v0[RN(x,y)];
				HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
			}
		}
	}
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0.0;
}

#undef i
#undef j
```

F.2.8 tak.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE STANDARD
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
                    a3[s][t]=1-i*i*DX*DX;
                    a4[s][t]=1-i*i*DX*DX;
                    a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                    a6[s][t]=0;
                    */

                    //1st derivative is backward difference, DX and DY independent
                    a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                    a2[s][t]=1-i*i*DX*DX;
                    a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                    a6[s][t]=0;

                    //Value near LC is average of values to the left and right.
                    if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                    {
                            a1[s][t]=-0.5;
                            a2[s][t]=-0.5;
                            a3[s][t]=0;
                            a4[s][t]=0;
                            a5[s][t]=1;
                            a6[s][t]=0;
                    }

                    //Polynomial coefficients of HHP (constant term taken care of in function "f")
                    //c[#] is the coefficient for (Pij)^(#+1)
                    const T A_A=1/(RATIO*P_OP*P_OP);
                    const T P_RET=RATIO*P_OP;

                    c[0][s][t]=A_A*P_OP*P_RET;
                    c[1][s][t]=-A_A*(P_OP+P_RET);
                    c[2][s][t]=A_A;
            }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=0;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;
    }
```

```
        //Right Boundary RPr+ZPz=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=0;
        }

        //Bottom Boundary
        //Outside star, inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=-1;
                bB6[s]=0;
        }

        //Outside light cylinder P=P_OP which is specified from the
           start.
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
```

```
			bT6[s]=0;
		}

		//Bottom Left Corner
		{
			int s=0;
			int t=0;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;

			bB1[s]=0;
			bB2[s]=0;
			bB3[s]=1;
			bB4[s]=0;
			bB5[s]=0;
			bB6[s]=P_OP;
		}

		//Top Left Corner
		{
			int s=0;
			int t=N_MAX_Y-3;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;

			bT1[s]=0;
			bT2[s]=i;
			bT3[s]=-j;
			bT4[s]=j;
			bT5[s]=0;
			bT6[s]=0;
		}

		//Bottom Right Corner
		{
			int s=N_MAX_X-3;
			int t=0;

			bR1[t]=-i;
			bR2[t]=i;
			bR3[t]=0;
			bR4[t]=j;
```

```
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
                }
        }
}
```

```
//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			if(v0[RN(x,y)]>=P_OP)
			{
				HHP[x][y]=0;
				HHP_PRIME[x][y]=0;
			}
			else
			{
				HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]*v0[RN(x,y)];
				HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
			}
		}
	}
}

//This represents the part of HHP that is a function of "r" and "z" (NOT "Phi"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0;
}

#undef i
#undef j
```

F.2.9 tak_jets.c

```c
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE STANDARD
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
	//Equation coefficients
	for(int s=0;s<=N_MAX_X-3;s++)
	{
		for(int t=0;t<=N_MAX_Y-3;t++)
		{
			/*
			//These equations represent an alternate finite difference choice for the first derivative.
			//The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
			//1st derivative is central difference
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=1-i*i*DX*DX;
			a4[s][t]=1-i*i*DX*DX;
			a5[s][t]=-4+4*i*i*DX*DX;
			a6[s][t]=0;
			*/

			/*
			//1st derivative is central difference, DX and DY independent
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
			a6[s][t]=0;
			*/

			/*
			//1st derivative is backward difference
			a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
			a2[s][t]=1-i*i*DX*DX;
```

```
            a3[s][t]=1-i*i*DX*DX;
            a4[s][t]=1-i*i*DX*DX;
            a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
            a6[s][t]=0;
            */

            //1st derivative is backward difference, DX and DY independent
            a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
            a2[s][t]=1-i*i*DX*DX;
            a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
            a6[s][t]=0;

            //Value near LC is average of values to the left and right.
            if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
            {
                a1[s][t]=-0.5;
                a2[s][t]=-0.5;
                a3[s][t]=0;
                a4[s][t]=0;
                a5[s][t]=1;
                a6[s][t]=0;
            }

            //Polynomial coefficients of HHP (constant term taken care of in function "f")
            //c[#] is the coefficient for (Pij)^(#+1)
            const T A_A=1/(RATIO*P_OP*P_OP);
            const T P_RET=RATIO*P_OP;

            c[0][s][t]=A_A*P_OP*P_RET;
            c[1][s][t]=-A_A*(P_OP+P_RET);
            c[2][s][t]=A_A;
        }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
        int s=0;

        bL1[t]=1;
        bL2[t]=0;
        bL3[t]=0;
        bL4[t]=0;
        bL5[t]=0;
        bL6[t]=0;
    }
```

```
        //Right Boundary RPr+ZPz=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=0;
        }

        //Bottom Boundary
        //Outside star, inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=-1;
                bB6[s]=0;
        }

        //Outside light cylinder P=P_OP which is specified from the
            start.
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
```

```
			bT6[s]=0;
		}

		//Bottom Left Corner
		{
			int s=0;
			int t=0;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;

			bB1[s]=0;
			bB2[s]=0;
			bB3[s]=1;
			bB4[s]=0;
			bB5[s]=0;
			bB6[s]=P_OP;
		}

		//Top Left Corner
		{
			int s=0;
			int t=N_MAX_Y-3;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;

			bT1[s]=0;
			bT2[s]=i;
			bT3[s]=-j;
			bT4[s]=j;
			bT5[s]=0;
			bT6[s]=0;
		}

		//Bottom Right Corner
		{
			int s=N_MAX_X-3;
			int t=0;

			bR1[t]=-i;
			bR2[t]=i;
			bR3[t]=0;
			bR4[t]=j;
```

```
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
                }
        }
}
```

```
//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        for(int x=0;x<N_MAX_X-2;x++)
        {
                for(int y=0;y<N_MAX_Y-2;y++)
                {
                        if(v0[RN(x,y)]>=P_OP)
                        {
                                HHP[x][y]=0;
                                HHP_PRIME[x][y]=0;
                        }
                        else if (v0[RN(x,y)]<v0[RN(N_LC-2,N_MAX_Y-3)])
                        {

                                T KH=2.0;
                                //T BETA=0.9995;
                                //T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
                                //T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));

                                HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA*v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x,y)]/P_OP-2);
                        }
                        else
                        {
                                HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]*v0[RN(x,y)];
                                HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
                        }
                }
        }
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
        return 0;
}

#undef i
#undef j
```

F.2.10 tak_theory_jets.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph
    will be of.
If there is more than one type of simulation mixed together, this
    must match whatever the bottom boundary is of.
*/
#define TYPE JETS
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        //These equations represent an alternate
                            finite difference choice for the first
                            derivative.
                        /*
                        //The upper right corner is impossible to
                            solve for using this choice, so I don't
                            recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX
                            and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```
				a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
				a6[s][t]=0;

				//Value near LC is average of values to the left and right.
				if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
				{
					a1[s][t]=-0.5;
					a2[s][t]=-0.5;
					a3[s][t]=0;
					a4[s][t]=0;
					a5[s][t]=1;
					a6[s][t]=0;
				}

				//Polynomial coefficients of HHP (constant term taken care of in function "f")
				//c[#] is the coefficient for (Pij)^(#+1)
				c[0][s][t]=4;
				c[1][s][t]=-6;
				c[2][s][t]=2;
			}
		}

		//Left Boundary P=0
		for(int t=1;t<=N_MAX_Y-4;t++)
		{
			int s=0;

			bL1[t]=1;
			bL2[t]=0;
			bL3[t]=0;
			bL4[t]=0;
			bL5[t]=0;
			bL6[t]=0;
		}

		//Right Boundary P=P_OP
		for(int t=1;t<=N_MAX_Y-4;t++)
		{
			int s=N_MAX_X-3;

			bR1[t]=0;
			bR2[t]=1;
			bR3[t]=0;
			bR4[t]=0;
			bR5[t]=0;
			bR6[t]=P_OP;
		}

		//Bottom Boundary P=P_OP
```

```
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary
        //Inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=0;
                bT4[s]=1;
                bT5[s]=-1;
                bT6[s]=0;
        }

        //Outside light cylinder P=P_OP which is specified from the start.
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=0;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=P_OP;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

```

```
            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
    }

    //Top Left Corner
    {
            int s=0;
            int t=N_MAX_Y-3;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;

            bT1[s]=0;
            bT2[s]=0;
            bT3[s]=0;
            bT4[s]=1;
            bT5[s]=-1;
            bT6[s]=0;
    }

    //Bottom Right Corner
    {
            int s=N_MAX_X-3;
            int t=0;

            bR1[t]=0;
            bR2[t]=1;
            bR3[t]=0;
            bR4[t]=0;
            bR5[t]=0;
            bR6[t]=P_OP;

            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
    }

    //Top Right Corner
    {
            int s=N_MAX_X-3;
            int t=N_MAX_Y-3;
```

```
            bR1[t]=0;
            bR2[t]=1;
            bR3[t]=0;
            bR4[t]=0;
            bR5[t]=0;
            bR6[t]=P_OP;

            bT1[s]=0;
            bT2[s]=0;
            bT3[s]=0;
            bT4[s]=1;
            bT5[s]=0;
            bT6[s]=P_OP;
        }
}

//This function inserts the star.
void star(void){}

//__forceinline void hhpSet(T HHP[(N_MAX_X-2)][(N_MAX_Y-2)],T
    HHP_PRIME[(N_MAX_X-2)][(N_MAX_Y-2)])
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        for(int x=0;x<N_MAX_X-2;x++)
        {
                for(int y=0;y<N_MAX_Y-2;y++)
                {
                        if(v0[RN(x,y)]>P_OP){HHP[x][y]=0.0;}
                        else if (v0[RN(x,y)]<=0.0){HHP[x][y]=0;v0[RN(
                            x,y)]=0.0;}
                        else
                        {
                                T KH=6.6;
                                T BETA=1.6;
                                HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA
                                    *v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x
                                    ,y)]/P_OP-2);
                        }
                }
        }
}

//This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
        return 0.0;
}

#undef i
#undef j
```

F.2.11 tak_null.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE NULLSHEET
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                        a6[s][t]=0;

                        //Value near LC is average of values to the left and right.
                        if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                        {
                                a1[s][t]=-0.5;
                                a2[s][t]=-0.5;
                                a3[s][t]=0;
                                a4[s][t]=0;
                                a5[s][t]=1;
                                a6[s][t]=0;
                        }

                        /*
                        //Unused
                        //Polynomial coefficients of HHP (constant term taken care of in function "f")
                        //c[#] is the coefficient for (Pij)^(#+1)
                        c[0][s][t]=0;
                        c[1][s][t]=0;
                        c[2][s][t]=0;
                        */
                }
        }

        //The bottom right coner and bottom edge past the light cylinder are changed elsewhere, so the choices for these are largely unimportant.
        //Left Boundary P=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
```

```
		bL5[t]=0;
		bL6[t]=0;
	}

	//Right Boundary RPr+ZPz=0
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=N_MAX_X-3;

		bR1[t]=-i;
		bR2[t]=i;
		bR3[t]=-j;
		bR4[t]=j;
		bR5[t]=0;
		bR6[t]=0;
	}

	//Bottom Boundary
	//Outside star, inside light cylinder Pz=0
	for(int s=1;((s<=N_LC-1)&&(s<=N_MAX_X-4));s++)
	{
		int t=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=-1;
		bB6[s]=0;
	}

	//Outside light cylinder H^2=(R^2-1)*(Pz)^2
	//To begin the simulation, just use P=P_OP
	for(int s=N_LC;s<=N_MAX_X-4;s++)
	{
		int t=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=0;
		bB6[s]=P_OP;
	}

	//Top Boundary RPr+ZPz=0
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		int t=N_MAX_Y-3;

		bT1[s]=-i;
		bT2[s]=i;
```

```
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
```

```
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
                }
        }
```

```
}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			if(v0[RN(x,y)]>=P_OP){HHP[x][y]=0;}
			else
			{
				//All choices should work, but the last one is the most general.
				//Original P_OP=1
				//HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[RN(x,y)])*pow(1-v0[RN(x,y)],0.4))*(0.428*(5+6*v0[RN(x,y)]*(v0[RN(x,y)]-2))/(pow(fabs(1-v0[RN(x,y)]),0.6)));
				//Alternate P_OP=1
				//HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[RN(x,y)])*pow(1-v0[RN(x,y)],0.4))*(2.568*(v0[RN(x,y)]-1.40825)*(v0[RN(x,y)]-0.591752))/(pow(fabs(1-v0[RN(x,y)]),0.6));
				//General P_OP
				HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[RN(x,y)]/P_OP)*pow(1-v0[RN(x,y)]/P_OP,0.4))*(0.428*(5*P_OP*P_OP-12*P_OP*v0[RN(x,y)]+6*v0[RN(x,y)]*v0[RN(x,y)])/(P_OP*P_OP*pow(fabs(1-v0[RN(x,y)]/P_OP),0.6)));
			}
			HHP_PRIME[x][y]=0;
		}
	}
	resetTak();
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0;
}

#undef i
#undef j
```

F.2.12 ckf_monopole_jets.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph
    will be of.
If there is more than one type of simulation mixed together, this
    must match whatever the bottom boundary is of.
*/
#define TYPE MONOPOLE
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate
                            finite difference choice for the first
                            derivative.
                        //The upper right corner is impossible to
                            solve for using this choice, so I don't
                            recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX
                            and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
                            ;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
            a3[s][t]=1-i*i*DX*DX;
            a4[s][t]=1-i*i*DX*DX;
            a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
            a6[s][t]=0;
            */

            //1st derivative is backward difference, DX and DY independent
            a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
            a2[s][t]=1-i*i*DX*DX;
            a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
            a6[s][t]=0;

            //Value near LC is average of values to the left and right.
            if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
            {
                a1[s][t]=-0.5;
                a2[s][t]=-0.5;
                a3[s][t]=0;
                a4[s][t]=0;
                a5[s][t]=1;
                a6[s][t]=0;
            }

            //Polynomial coefficients of HHP (constant term taken care of in function "f")
            //c[#] is the coefficient for (Pij)^(#+1)
            c[0][s][t]=4;
            c[1][s][t]=-6;
            c[2][s][t]=2;
        }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
        int s=0;

        bL1[t]=1;
        bL2[t]=0;
        bL3[t]=0;
        bL4[t]=0;
        bL5[t]=0;
        bL6[t]=0;
    }

    //Right Boundary RPr+ZPz=0
    for(int t=1;t<=N_MAX_Y-4;t++)
```

```
        {
                int s=N_MAX_X-3;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=0;
        }

        //Bottom Boundary P=P_OP
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
```

```
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
```

```
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void){}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        T KH=8.48;
        if (toggle)
        {
                printf("Takamori\n");

                for(int x=0;x<N_MAX_X-2;x++)
                {
                        for(int y=0;y<N_MAX_Y-2;y++)
                        {
                                if(v0[RN(x,y)]<v0[RN(N_LC-2,N_MAX_Y-3)])
                                {
                                        //T KH=2.48;
                                        //T BETA=0.9995;
                                        T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
                                        T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));
                                        HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(v0[RN(x,y)]-1)*(v0[RN(x,y)]-2);
                                        HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
                                }
                                else
                                {
```

```
                        HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]*v0[RN(x,y)];
                        HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
                    }
                }
            }
        }
        else
        {
            printf("CKF\n");
            //T KH=2.48;
            //T BETA=0.9995;

            T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
            T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));

            T HHP_L[N_MAX_Y-2];
            for(int KP=0;KP<N_MAX_Y-2;KP++)
            {
                HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
            }

            //At r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder).
            //For 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.

            for(int jj=0;jj<=N_LC-2;jj++)
            {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                    int KP=0;
                    {
                        const T PT=v0[RN(jj,kk)];

                        if(PT<v0[RN(N_LC-2,0)])
                        {
                            if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                            {
```

```
                    const T PS=PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                    HHP[jj][kk]=KH*KH*(PS-0.5*BETA*PS*PS)*(1-BETA*PS);
                }
                else
                {
                    for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                    const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                    const T Q2=PT-v0[RN(N_LC-1,KP)];
                    HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                }
            }
            else
            {
                HHP[jj][kk]=0;
            }
        }
        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
    }
}

for(int jj=N_LC-1;jj==N_LC-1;jj++)
{
    for(int kk=0;kk<=N_MAX_Y-3;kk++)
    {
        HHP[N_LC-1][kk]=HHP_L[kk];
    }
```

```
		}

		for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
		{
			for(int kk=0;kk<=N_MAX_Y-3;kk++)
			{
				int KP=0;
				const T PT=v0[RN(jj,kk)];

				for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

				const T Q1=PT-v0[RN(N_LC-1,KP+1)];
				const T Q2=PT-v0[RN(N_LC-1,KP)];
				HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
				if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
			}
		}

		//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
		for(int x=0;x<N_MAX_X-2;x++)
		{
			for(int y=0;y<N_MAX_Y-2;y++)
			{
				HHP_PRIME[x][y]=0.1*sqrt(x*x+y*y);
			}
		}
	}
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0;
}

#undef i
#undef j
```

F.2.13 ckf_tak.c

```c
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE STANDARD
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
            a3[s][t]=1-i*i*DX*DX;
            a4[s][t]=1-i*i*DX*DX;
            a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
            a6[s][t]=0;
            */

            //1st derivative is backward difference, DX and DY independent
            a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
            a2[s][t]=1-i*i*DX*DX;
            a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
            a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
            a6[s][t]=0;

            //Value near LC is average of values to the left and right.
            if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
            {
                a1[s][t]=-0.5;
                a2[s][t]=-0.5;
                a3[s][t]=0;
                a4[s][t]=0;
                a5[s][t]=1;
                a6[s][t]=0;
            }

            //Polynomial coefficients of HHP (constant term taken care of in function "f")
            //c[#] is the coefficient for (Pij)^(#+1)
            c[0][s][t]=0;
            c[1][s][t]=0;
            c[2][s][t]=0;
        }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
        int s=0;

        bL1[t]=1;
        bL2[t]=0;
        bL3[t]=0;
        bL4[t]=0;
        bL5[t]=0;
        bL6[t]=0;
    }

    //Right Boundary RPr+ZPz=0
    for(int t=1;t<=N_MAX_Y-4;t++)
```

```
    {
        int s=N_MAX_X-3;

        bR1[t]=-i;
        bR2[t]=i;
        bR3[t]=-j;
        bR4[t]=j;
        bR5[t]=0;
        bR6[t]=0;
    }

    //Bottom Boundary
    //Outside star, inside light cylinder Pz=0
    for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
    {
        int t=0;

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=1;
        bB4[s]=0;
        bB5[s]=-1;
        bB6[s]=0;
    }

    //Outside light cylinder P=P_OP which is specified from the start.
    for(int s=N_LC-1;s<=N_MAX_X-4;s++)
    {
        int t=0;

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=1;
        bB4[s]=0;
        bB5[s]=0;
        bB6[s]=P_OP;//Needs to be fixed
    }

    //Top Boundary RPr+ZPz=0
    for(int s=1;s<=N_MAX_X-4;s++)
    {
        int t=N_MAX_Y-3;

        bT1[s]=-i;
        bT2[s]=i;
        bT3[s]=-j;
        bT4[s]=j;
        bT5[s]=0;
        bT6[s]=0;
    }
```

```
        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

```

```
			bB1[s]=0;
			bB2[s]=0;
			bB3[s]=1;
			bB4[s]=0;
			bB5[s]=0;
			bB6[s]=P_OP;
	}

	//Top Right Corner
	{
			int s=N_MAX_X-3;
			int t=N_MAX_Y-3;

			bR1[t]=-1;
			bR2[t]=1;
			bR3[t]=0;
			bR4[t]=0;
			bR5[t]=0;
			bR6[t]=0;

			bT1[s]=0;
			bT2[s]=0;
			bT3[s]=-1;
			bT4[s]=1;
			bT5[s]=0;
			bT6[s]=0;
	}
}

//This function inserts the star.
void star(void)
{
	for(int s=0;s<N_S_X;s++)
	{
			const T R=DX*i;
			for(int t=0;t<N_S_Y;t++)
			{
					const T Z=DY*j;
					a1[s][t]=0;
					a2[s][t]=0;
					a3[s][t]=0;
					a4[s][t]=0;
					a5[s][t]=1;
					a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
			}
	}
}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
```

```
		if(toggle)
		{
			printf("Takamori\n");
			//SMOOTH=-1; //Opportunity to change smoothness mid simulation. Change code so that SMOOTH is a variable, and not a macro.
			return;
		}
		printf("CKF\n");

		T HHP_L[N_MAX_Y-2];
		for(int KP=0;KP<N_MAX_Y-2;KP++)
		{
			HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
		}

		//at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
		//for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
		for(int jj=0;jj<=N_S_X-1;jj++)
		{
			for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
			{
				int KP=0;
				{
					const T PT=v0[RN(jj,kk)];

					if(PT<v0[RN(N_LC-2,0)])
					{
						if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
						{
							HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
						}
						else
						{
							for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

							const T Q1=PT-v0[RN(N_LC-1,KP+1)];
							const T Q2=PT-v0[RN(N_LC-1,KP)];
							HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
```

```
                                                +1])/(Q1-Q2);
                                        }
                                }
                                else
                                {
                                        HHP[jj][kk]=0;
                                }
                        }
                        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
                }
        }

        //for NR_S<r<rLC
        for(int jj=N_S_X;jj<=N_LC-2;jj++)
        {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        {
                                const T PT=v0[RN(jj,kk)];

                                if(PT<v0[RN(N_LC-2,0)])
                                {
                                        if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                                        {
                                                HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                                        }
                                        else
                                        {
                                                for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                                                const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                                                const T Q2=PT-v0[RN(N_LC-1,KP)];
                                                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                                        }
                                }
                                else
                                {
                                        HHP[jj][kk]=0;
                                }
                        }
                        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
```

```
		}
	}

	for(int jj=N_LC-1;jj==N_LC-1;jj++)
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			HHP[N_LC-1][kk]=HHP_L[kk];
		}
	}

	for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			int KP=0;
			const T PT=v0[RN(jj,kk)];

			for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

			const T Q1=PT-v0[RN(N_LC-1,KP+1)];
			const T Q2=PT-v0[RN(N_LC-1,KP)];
			//if(fabs(Q1-Q2)>0.00001)
			{
				HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
			}
			if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
		}
	}

	//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
	for(int x=0;x<N_MAX_X-2;x++)
	{
		for(int y=0;y<N_MAX_Y-2;y++)
		{
			T part1,part2;
			if(x==N_MAX_X-3)
			{
				part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
			}
			else
			{
				part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
			}
			if(y==N_MAX_Y-3)
			{
```

```
                                        part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
                                            x,y)]-v0[RN(x,y-1)])/(DY*DY);
                              }
                              else
                              {
                                        part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                                            x,y+1)]-v0[RN(x,y)])/(DY*DY);
                              }
                              HHP_PRIME[x][y]=part1+part2;
                    }
          }
}

//This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
          return 0;
}

#undef i
#undef j
```

F.2.14 ckf_null_tak.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE NULLSHEET
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
	//Equation coefficients
	for(int s=0;s<=N_MAX_X-3;s++)
	{
		for(int t=0;t<=N_MAX_Y-3;t++)
		{
			/*
			//These equations represent an alternate finite difference choice for the first derivative.
			//The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
			//1st derivative is central difference
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=1-i*i*DX*DX;
			a4[s][t]=1-i*i*DX*DX;
			a5[s][t]=-4+4*i*i*DX*DX;
			a6[s][t]=0;
			*/

			/*
			//1st derivative is central difference, DX and DY independent
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
			a6[s][t]=0;
			*/

			/*
			//1st derivative is backward difference
			a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
			a2[s][t]=1-i*i*DX*DX;
```

```
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                        a6[s][t]=0;

                        //Value near LC is average of values to the left and right.
                        if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                        {
                                a1[s][t]=-0.5;
                                a2[s][t]=-0.5;
                                a3[s][t]=0;
                                a4[s][t]=0;
                                a5[s][t]=1;
                                a6[s][t]=0;
                        }

                        /*
                        //Unused
                        //Polynomial coefficients of HHP (constant term taken care of in function "f")
                        //c[#] is the coefficient for (Pij)^(#+1)
                        c[0][s][t]=0;
                        c[1][s][t]=0;
                        c[2][s][t]=0;
                        */
                }
        }

        //The bottom right coner and bottom edge past the light cylinder are changed elsewhere, so the choices for these are largely unimportant.
        //Left Boundary P=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
```

```
		bL5[t]=0;
		bL6[t]=0;
	}

	//Right Boundary RPr+ZPz=0
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=N_MAX_X-3;

		bR1[t]=-i;
		bR2[t]=i;
		bR3[t]=-j;
		bR4[t]=j;
		bR5[t]=0;
		bR6[t]=0;
	}

	//Bottom Boundary
	//Outside star, inside light cylinder Pz=0
	for(int s=1;((s<N_LC+1)&&(s<=N_MAX_X-4));s++)
	{
		int t=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=-1;
		bB6[s]=0;
	}

	//Outside light cylinder H^2=(R^2-1)*(Pz)^2
	//To begin the simulation, just use P=P_OP
	for(int s=N_LC+1;s<=N_MAX_X-4;s++)
	{
		int t=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=0;
		bB6[s]=P_OP;
	}

	//Top Boundary RPr+ZPz=0
	for(int s=1;s<=N_MAX_X-4;s++)
	{
		int t=N_MAX_Y-3;

		bT1[s]=-i;
		bT2[s]=i;
```

```
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=1;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
```

```
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
                }
        }
```

```
}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        if (toggle){printf("Takamori\n");return;}
        printf("CKF\n");

        //T Slopes[NZ_MAX+1];
        T HHP_L[N_MAX_Y-2];
        for(int KP=0;KP<N_MAX_Y-2;KP++)
        {
                HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
        }

        //at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
        //for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
        for(int jj=0;jj<=N_S_X-1;jj++)
        {
                for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        {
                                const T PT=v0[RN(jj,kk)];

                                if(PT<v0[RN(N_LC-2,0)])
                                {
                                        if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                                        {
                                                HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                                        }
                                        else
                                        {
                                                for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                                                const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                                                const T Q2=PT-v0[RN(N_LC-1,KP)];
                                                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
```

```
                                        +1])/(Q1-Q2);
                    }
                }
                else
                {
                    HHP[jj][kk]=0;
                }
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
    }

    //for NR_S<r<rLC
    for(int jj=N_S_X;jj<=N_LC-2;jj++)
    {
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
            int KP=0;
            {
                const T PT=v0[RN(jj,kk)];

                if(PT<v0[RN(N_LC-2,0)])
                {
                    if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                    {
                        HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                    }
                    else
                    {
                        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                    }
                }
                else
                {
                    HHP[jj][kk]=0;
                }
            }
            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
```

```
                }
        }

        for(int jj=N_LC-1;jj==N_LC-1;jj++)
        {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                        HHP[N_LC-1][kk]=HHP_L[kk];
                }
        }

        for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
        {
                for(int kk=0;kk<=N_MAX_Y-3;kk++)
                {
                        int KP=0;
                        const T PT=v0[RN(jj,kk)];

                        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++)
                            ;

                        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        //if(fabs(Q1-Q2)>0.00001)
                        {
                                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
                                    +1])/(Q1-Q2);
                        }
                        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
                }
        }

        //TODO - HHP_PRIME is largely unnecessary, and this
            calculation may be wrong.
        for(int x=0;x<N_MAX_X-2;x++)
        {
                for(int y=0;y<N_MAX_Y-2;y++)
                {
                        T part1,part2;
                        if(x==N_MAX_X-3)
                        {
                                part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(
                                    x,y)]-v0[RN(x-1,y)])/(DX*DX);
                        }
                        else
                        {
                                part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(
                                    x+1,y)]-v0[RN(x,y)])/(DX*DX);
                        }
                        if(y==N_MAX_Y-3)
                        {
```

```
                                part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
                                    x,y)]-v0[RN(x,y-1)])/(DY*DY);
                        }
                        else
                        {
                                part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                                    x,y+1)]-v0[RN(x,y)])/(DY*DY);
                        }
                        HHP_PRIME[x][y]=part1+part2;
                }
        }

        resetCKF();
}

//This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
        return 0;
}

#undef i
#undef j
```

F.2.15 tak_ckf.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE STANDARD
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
					a3[s][t]=1-i*i*DX*DX;
					a4[s][t]=1-i*i*DX*DX;
					a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
					a6[s][t]=0;
					*/

					//1st derivative is backward difference, DX and DY independent
					a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
					a2[s][t]=1-i*i*DX*DX;
					a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
					a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
					a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
					a6[s][t]=0;

					//Value near LC is average of values to the left and right.
					if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
					{
						a1[s][t]=-0.5;
						a2[s][t]=-0.5;
						a3[s][t]=0;
						a4[s][t]=0;
						a5[s][t]=1;
						a6[s][t]=0;
					}

					//Polynomial coefficients of HHP (constant term taken care of in function "f")
					//c[#] is the coefficient for (Pij)^(#+1)
					const T A_A=1/(RATIO*P_OP*P_OP);
					const T P_RET=RATIO*P_OP;

					c[0][s][t]=A_A*P_OP*P_RET;
					c[1][s][t]=-A_A*(P_OP+P_RET);
					c[2][s][t]=A_A;
				}
			}

			//Left Boundary P=0
			for(int t=1;t<=N_MAX_Y-4;t++)
			{
				int s=0;

				bL1[t]=1;
				bL2[t]=0;
				bL3[t]=0;
				bL4[t]=0;
				bL5[t]=0;
				bL6[t]=0;
			}
```

```
    //Right Boundary RPr+ZPz=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=N_MAX_X-3;

            bR1[t]=-i;
            bR2[t]=i;
            bR3[t]=-j;
            bR4[t]=j;
            bR5[t]=0;
            bR6[t]=0;
    }

    //Bottom Boundary
    //Outside star, inside light cylinder Pz=0
    for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
    {
            int t=0;

            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=-1;
            bB6[s]=0;
    }

    //Outside light cylinder P=P_OP which is specified from the
       start.
    for(int s=N_LC-1;s<=N_MAX_X-4;s++)
    {
            int t=0;

            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
    }

    //Top Boundary RPr+ZPz=0
    for(int s=1;s<=N_MAX_X-4;s++)
    {
            int t=N_MAX_Y-3;

            bT1[s]=-i;
            bT2[s]=i;
            bT3[s]=-j;
            bT4[s]=j;
            bT5[s]=0;
```

```
                bT6[s]=0;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
```

```
        bR5[t]=0;
        bR6[t]=j*P_OP;

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=1;
        bB4[s]=0;
        bB5[s]=0;
        bB6[s]=P_OP;
    }

    //Top Right Corner
    {
        int s=N_MAX_X-3;
        int t=N_MAX_Y-3;

        bR1[t]=-1;
        bR2[t]=1;
        bR3[t]=0;
        bR4[t]=0;
        bR5[t]=0;
        bR6[t]=0;

        bT1[s]=0;
        bT2[s]=0;
        bT3[s]=-1;
        bT4[s]=1;
        bT5[s]=0;
        bT6[s]=0;
    }
}

//This function inserts the star.
void star(void)
{
    for(int s=0;s<N_S_X;s++)
    {
        const T R=DX*i;
        for(int t=0;t<N_S_Y;t++)
        {
            const T Z=DY*j;
            a1[s][t]=0;
            a2[s][t]=0;
            a3[s][t]=0;
            a4[s][t]=0;
            a5[s][t]=1;
            a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
        }
    }
}

```

```
//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        if (!toggle)
        {
                printf("Takamori\n");

                for(int x=0;x<N_MAX_X-2;x++)
                {
                        for(int y=0;y<N_MAX_Y-2;y++)
                        {
                                if(v0[RN(x,y)]>=P_OP)
                                {
                                        HHP[x][y]=0;
                                        HHP_PRIME[x][y]=0;
                                }
                                else
                                {
                                        HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]*v0[RN(x,y)];
                                        HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)];
                                }
                        }
                }
        }
        else
        {
                printf("CKF\n");

                T HHP_L[N_MAX_Y-2];
                for(int KP=0;KP<N_MAX_Y-2;KP++)
                {
                        HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
                }

                //at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
                //for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
                for(int jj=0;jj<=N_S_X-1;jj++)
                {
```

```
for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
{
    int KP=0;
    {
        const T PT=v0[RN(jj,kk)];

        if(PT<v0[RN(N_LC-2,0)])
        {
            if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
            {
                HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
            }
            else
            {
                for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                const T Q2=PT-v0[RN(N_LC-1,KP)];
                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
            }
        }
        else
        {
            HHP[jj][kk]=0;
        }
    }
    if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
}
```

```
		}

		//for NR_S<r<rLC
		for(int jj=N_S_X;jj<=N_LC-2;jj++)
		{
			for(int kk=0;kk<=N_MAX_Y-3;kk++)
			{
				int KP=0;
				{
					const T PT=v0[RN(jj,kk)];

					if(PT<v0[RN(N_LC-2,0)])
					{
						if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
						{
							HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
						}
						else
						{
							for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

							const T Q1=PT-v0[RN(N_LC-1,KP+1)];
							const T Q2=PT-v0[RN(N_LC-1,KP)];
							HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
						}
					}
					else
					{
						HHP[jj][kk]=0;
```

```
            }
        }
        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
    }
}

for(int jj=N_LC-1;jj==N_LC-1;jj++)
{
    for(int kk=0;kk<=N_MAX_Y-3;kk++)
    {
        HHP[N_LC-1][kk]=HHP_L[kk];
    }
}

for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
{
    for(int kk=0;kk<=N_MAX_Y-3;kk++)
    {
        int KP=0;
        const T PT=v0[RN(jj,kk)];

        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
        const T Q2=PT-v0[RN(N_LC-1,KP)];
        //if(fabs(Q1-Q2)>0.00001)
        {
            HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
        }
        if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
    }
}

//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
for(int x=0;x<N_MAX_X-2;x++)
{
    for(int y=0;y<N_MAX_Y-2;y++)
    {
        T part1,part2;
        if(x==N_MAX_X-3)
        {
            part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
        }
        else
        {
```

```
                                        part1=(HHP[x+1][y]-HHP[x][y])
                                            *(v0[RN(x+1,y)]-v0[RN(x,y)
                                            ])/(DX*DX);
                                }
                                if(y==N_MAX_Y-3)
                                {
                                        part2=(HHP[x][y]-HHP[x][y-1])
                                            *(v0[RN(x,y)]-v0[RN(x,y-1)
                                            ])/(DY*DY);
                                }
                                else
                                {
                                        part2=(HHP[x][y+1]-HHP[x][y])
                                            *(v0[RN(x,y+1)]-v0[RN(x,y)
                                            ])/(DY*DY);
                                }
                                HHP_PRIME[x][y]=part1+part2;
                        }
                }
        }
}

//This represents the part of HHP that is a function of "r" and "z" (
    NOT "Phi"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
        return 0;
}

#undef i
#undef j
```

F.2.16 tak_ckf_jets.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE JETS
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
                    a3[s][t]=1-i*i*DX*DX;
                    a4[s][t]=1-i*i*DX*DX;
                    a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                    a6[s][t]=0;
                    */

                    //1st derivative is backward difference, DX and DY independent
                    a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                    a2[s][t]=1-i*i*DX*DX;
                    a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                    a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                    a6[s][t]=0;

                    //Value near LC is average of values to the left and right.
                    if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                    {
                            a1[s][t]=-0.5;
                            a2[s][t]=-0.5;
                            a3[s][t]=0;
                            a4[s][t]=0;
                            a5[s][t]=1;
                            a6[s][t]=0;
                    }

                    //Polynomial coefficients of HHP (constant term taken care of in function "f")
                    //c[#] is the coefficient for (Pij)^(#+1)
                    const T A_A=1/(RATIO*P_OP*P_OP);
                    const T P_RET=RATIO*P_OP;

                    c[0][s][t]=A_A*P_OP*P_RET;
                    c[1][s][t]=-A_A*(P_OP+P_RET);
                    c[2][s][t]=A_A;
            }
    }

    //Left Boundary P=0
    for(int t=1;t<=N_MAX_Y-4;t++)
    {
            int s=0;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;
    }
```

```
        //Right Boundary RPr+ZPz=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=0;
        }

        //Bottom Boundary
        //Outside star, inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=-1;
                bB6[s]=0;
        }

        //Outside light cylinder P=P_OP which is specified from the
            start.
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=-i;
                bT2[s]=i;
                bT3[s]=-j;
                bT4[s]=j;
                bT5[s]=0;
```

```
		bT6[s]=0;
	}

	//Bottom Left Corner
	{
		int s=0;
		int t=0;

		bL1[t]=1;
		bL2[t]=0;
		bL3[t]=0;
		bL4[t]=0;
		bL5[t]=0;
		bL6[t]=0;

		bB1[s]=0;
		bB2[s]=0;
		bB3[s]=1;
		bB4[s]=0;
		bB5[s]=0;
		bB6[s]=P_OP;
	}

	//Top Left Corner
	{
		int s=0;
		int t=N_MAX_Y-3;

		bL1[t]=1;
		bL2[t]=0;
		bL3[t]=0;
		bL4[t]=0;
		bL5[t]=0;
		bL6[t]=0;

		bT1[s]=0;
		bT2[s]=i;
		bT3[s]=-j;
		bT4[s]=j;
		bT5[s]=0;
		bT6[s]=0;
	}

	//Bottom Right Corner
	{
		int s=N_MAX_X-3;
		int t=0;

		bR1[t]=-i;
		bR2[t]=i;
		bR3[t]=0;
		bR4[t]=j;
```

```
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
                        a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
                }
        }
}
```

```
//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
        T KH=6.0;
        T BETA=2;

        if (!toggle)
        {
                printf("Takamori\n");

                for(int x=0;x<N_MAX_X-2;x++)
                {
                        for(int y=0;y<N_MAX_Y-2;y++)
                        {
                                if(v0[RN(x,y)]>=P_OP)
                                {
                                        HHP[x][y]=0;
                                        HHP_PRIME[x][y]=0;
                                }
                                else
                                {
                                        HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA*v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x,y)]/P_OP-2);
                                }
                        }
                }
        }
        else
        {
                printf("CKF\n");

                T HHP_L[N_MAX_Y-2];
                for(int KP=0;KP<N_MAX_Y-2;KP++)
                {
                        HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
                }

                //at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
                //for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
                for(int jj=0;jj<=N_S_X-1;jj++)
                {
                        for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
                        {
```

```
int KP=0;
{
	const T PT=v0[RN(jj,kk)];

	if(PT<v0[RN(N_LC-2,0)])
	{
		if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
		{
			HHP[jj][kk]=KH*KH*0.5*v0[RN(jj,kk)]*(BETA*v0[RN(jj,kk)]/P_OP-1)*(BETA*v0[RN(jj,kk)]/P_OP-2);
		}
		else
		{
			for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

			const T Q1=PT-v0[RN(N_LC-1,KP+1)];
			const T Q2=PT-v0[RN(N_LC-1,KP)];
			HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
		}
	}
	else
	{
		HHP[jj][kk]=0;
	}
}
if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
```

```
            }
        }

        //for NR_S<r<rLC
        for(int jj=N_S_X;jj<=N_LC-2;jj++)
        {
            for(int kk=0;kk<=N_MAX_Y-3;kk++)
            {
                int KP=0;
                {
                    const T PT=v0[RN(jj,kk)];

                    if(PT<v0[RN(N_LC-2,0)])
                    {
                        if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                        {
                            HHP[jj][kk]=KH*KH*0.5*v0[RN(jj,kk)]*(BETA*v0[RN(jj,kk)]/P_OP-1)*(BETA*v0[RN(jj,kk)]/P_OP-2);
                        }
                        else
                        {
                            for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                            const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                            const T Q2=PT-v0[RN(N_LC-1,KP)];
                            HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                        }
```

```
                    }
                    else
                    {
                        HHP[jj][kk]=0;
                    }
                }
                if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
            }
        }

        for(int jj=N_LC-1;jj==N_LC-1;jj++)
        {
            for(int kk=0;kk<=N_MAX_Y-3;kk++)
            {
                HHP[N_LC-1][kk]=HHP_L[kk];
            }
        }

        for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
        {
            for(int kk=0;kk<=N_MAX_Y-3;kk++)
            {
                int KP=0;
                const T PT=v0[RN(jj,kk)];

                for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                const T Q2=PT-v0[RN(N_LC-1,KP)];
                //if(fabs(Q1-Q2)>0.00001)
                {
                    HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                }
                if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
            }
        }

        //TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
        for(int x=0;x<N_MAX_X-2;x++)
        {
            for(int y=0;y<N_MAX_Y-2;y++)
            {
                T part1,part2;
                if(x==N_MAX_X-3)
                {
                    part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)
```

```
                                        ])/(DX*DX);
                                }
                                else
                                {
                                        part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
                                }
                                if(y==N_MAX_Y-3)
                                {
                                        part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(x,y)]-v0[RN(x,y-1)])/(DY*DY);
                                }
                                else
                                {
                                        part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(x,y+1)]-v0[RN(x,y)])/(DY*DY);
                                }
                                HHP_PRIME[x][y]=part1+part2;
                        }
                }
        }
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
        return 0;
}

#undef i
#undef j
```

F.2.17 tak_ckf_null.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE NULLSHEET
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
        //Equation coefficients
        for(int s=0;s<=N_MAX_X-3;s++)
        {
                for(int t=0;t<=N_MAX_Y-3;t++)
                {
                        /*
                        //These equations represent an alternate finite difference choice for the first derivative.
                        //The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
                        //1st derivative is central difference
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX;
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is central difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
                        a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
                        a6[s][t]=0;
                        */

                        /*
                        //1st derivative is backward difference
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
```

```
			a3[s][t]=1-i*i*DX*DX;
			a4[s][t]=1-i*i*DX*DX;
			a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
			a6[s][t]=0;
			*/

			//1st derivative is backward difference, DX and DY independent
			a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
			a2[s][t]=1-i*i*DX*DX;
			a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
			a6[s][t]=0;

			//Value near LC is average of values to the left and right.
			if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
			{
				a1[s][t]=-0.5;
				a2[s][t]=-0.5;
				a3[s][t]=0;
				a4[s][t]=0;
				a5[s][t]=1;
				a6[s][t]=0;
			}

			/*
			//Unused
			//Polynomial coefficients of HHP (constant term taken care of in function "f")
			//c[#] is the coefficient for (Pij)^(#+1)
			const T A_A=1/(RATIO*P_OP*P_OP);
			const T P_RET=RATIO*P_OP;

			c[0][s][t]=A_A*P_OP*P_RET;
			c[1][s][t]=-A_A*(P_OP+P_RET);
			c[2][s][t]=A_A;
			*/
		}
	}

	//The bottom right coner and bottom edge past the light cylinder are changed elsewhere, so the choices for these are largely unimportant.
	//Left Boundary P=0
	for(int t=1;t<=N_MAX_Y-4;t++)
	{
		int s=0;

		bL1[t]=1;
```

```
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;
        }

        //Right Boundary RPr+ZPz=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=-j;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=0;
        }

        //Bottom Boundary
        //Outside star, inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=-1;
                bB6[s]=0;
        }

        //Outside light cylinder H^2=(R^2-1)*(Pz)^2
        //To begin the simulation, just use P=P_OP
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary RPr+ZPz=0
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;
```

```
            bT1[s]=-i;
            bT2[s]=i;
            bT3[s]=-j;
            bT4[s]=j;
            bT5[s]=0;
            bT6[s]=0;
    }

    //Bottom Left Corner
    {
            int s=0;
            int t=0;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;

            bB1[s]=0;
            bB2[s]=0;
            bB3[s]=1;
            bB4[s]=0;
            bB5[s]=0;
            bB6[s]=P_OP;
    }

    //Top Left Corner
    {
            int s=0;
            int t=N_MAX_Y-3;

            bL1[t]=1;
            bL2[t]=0;
            bL3[t]=0;
            bL4[t]=0;
            bL5[t]=0;
            bL6[t]=0;

            bT1[s]=0;
            bT2[s]=i;
            bT3[s]=-j;
            bT4[s]=j;
            bT5[s]=0;
            bT6[s]=0;
    }

    //Bottom Right Corner
    {
            int s=N_MAX_X-3;
```

```
                int t=0;

                bR1[t]=-i;
                bR2[t]=i;
                bR3[t]=0;
                bR4[t]=j;
                bR5[t]=0;
                bR6[t]=j*P_OP;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Right Corner
        {
                int s=N_MAX_X-3;
                int t=N_MAX_Y-3;

                bR1[t]=-1;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=-1;
                bT4[s]=1;
                bT5[s]=0;
                bT6[s]=0;
        }
}

//This function inserts the star.
void star(void)
{
        for(int s=0;s<N_S_X;s++)
        {
                const T R=DX*i;
                for(int t=0;t<N_S_Y;t++)
                {
                        const T Z=DY*j;
                        a1[s][t]=0;
                        a2[s][t]=0;
                        a3[s][t]=0;
                        a4[s][t]=0;
                        a5[s][t]=1;
```

```
			a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
		}
	}
}

//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	if (!toggle)
	{
		printf("Takamori\n");

		for(int x=0;x<N_MAX_X-2;x++)
		{
			for(int y=0;y<N_MAX_Y-2;y++)
			{
				if(v0[RN(x,y)]>=1)
				{
					HHP[x][y]=0;
				}
				else
				{
					HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[RN(x,y)])*pow(1-v0[RN(x,y)],0.4))*(0.428*(5+6*v0[RN(x,y)]*(v0[RN(x,y)]-2))/(pow(1-v0[RN(x,y)],0.6)));
				}
			}
		}
		resetTak();
	}
	else
	{
		printf("CKF\n");

		T HHP_L[N_MAX_Y-2];
		for(int KP=0;KP<N_MAX_Y-2;KP++)
		{
			HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
		}

		//at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
		//for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
```

```
for(int jj=0;jj<=N_S_X-1;jj++)
{
    for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
    {
        int KP=0;
        {
            const T PT=v0[RN(jj,kk)];

            if(PT<v0[RN(N_LC-2,0)])
            {
                if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                {
                    HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
                }
                else
                {
                    for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                    const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                    const T Q2=PT-v0[RN(N_LC-1,KP)];
                    HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                }
            }
            else
            {
                HHP[jj][kk]=0;
            }
        }
```

```
			if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
		}
	}

	//for NR_S<r<rLC
	for(int jj=N_S_X;jj<=N_LC-2;jj++)
	{
		for(int kk=0;kk<=N_MAX_Y-3;kk++)
		{
			int KP=0;
			{
				const T PT=v0[RN(jj,kk)];

				if(PT<v0[RN(N_LC-2,0)])
				{
					if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
					{
						HHP[jj][kk]=HHP_L[N_MAX_Y-3]*PT/v0[RN(N_LC-2,N_MAX_Y-3)];
					}
					else
					{
						for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

						const T Q1=PT-v0[RN(N_LC-1,KP+1)];
						const T Q2=PT-v0[RN(N_LC-1,KP)];
						HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
					}
				}
```

```
			else
			{
				HHP[jj][kk]=0;
			}
		}
		if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
	}
}

for(int jj=N_LC-1;jj==N_LC-1;jj++)
{
	for(int kk=0;kk<=N_MAX_Y-3;kk++)
	{
		HHP[N_LC-1][kk]=HHP_L[kk];
	}
}

for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
{
	for(int kk=0;kk<=N_MAX_Y-3;kk++)
	{
		int KP=0;
		const T PT=v0[RN(jj,kk)];

		for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

		const T Q1=PT-v0[RN(N_LC-1,KP+1)];
		const T Q2=PT-v0[RN(N_LC-1,KP)];
		//if(fabs(Q1-Q2)>0.00001)
		{
			HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
		}
		if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
	}
}

//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
for(int x=0;x<N_MAX_X-2;x++)
{
	for(int y=0;y<N_MAX_Y-2;y++)
	{
		T part1,part2;
		if(x==N_MAX_X-3)
		{
			part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
```

```
				}
				else
				{
					part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
				}
				if(y==N_MAX_Y-3)
				{
					part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(x,y)]-v0[RN(x,y-1)])/(DY*DY);
				}
				else
				{
					part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(x,y+1)]-v0[RN(x,y)])/(DY*DY);
				}
				HHP_PRIME[x][y]=part1+part2;
			}
		}
		resetCKF();
	}
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
	return 0;
}

#undef i
#undef j
```

F.2.18 tak_theory_jets_ckf.c

```
/*
Refers to the type of simulation.
If it is a double simulation, this must match whatever the end graph will be of.
If there is more than one type of simulation mixed together, this must match whatever the bottom boundary is of.
*/
#define TYPE JETS
#define i ((T)(s+1))
#define j ((T)(t+1))

void initialize(void)
{
	//Equation coefficients
	for(int s=0;s<=N_MAX_X-3;s++)
	{
		for(int t=0;t<=N_MAX_Y-3;t++)
		{
			/*
			//These equations represent an alternate finite difference choice for the first derivative.
			//The upper right corner is impossible to solve for using this choice, so I don't recommend this option.
			//1st derivative is central difference
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=1-i*i*DX*DX;
			a4[s][t]=1-i*i*DX*DX;
			a5[s][t]=-4+4*i*i*DX*DX;
			a6[s][t]=0;
			*/

			/*
			//1st derivative is central difference, DX and DY independent
			a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
			a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
			a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
			a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX);
			a6[s][t]=0;
			*/

			/*
			//1st derivative is backward difference
			a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
			a2[s][t]=1-i*i*DX*DX;
```

```
                        a3[s][t]=1-i*i*DX*DX;
                        a4[s][t]=1-i*i*DX*DX;
                        a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
                        a6[s][t]=0;
                        */

                        //1st derivative is backward difference, DX and DY independent
                        a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
                        a2[s][t]=1-i*i*DX*DX;
                        a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
                        a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)-1/i-i*DX*DX;
                        a6[s][t]=0;

                        //Value near LC is average of values to the left and right.
                        if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
                        {
                                a1[s][t]=-0.5;
                                a2[s][t]=-0.5;
                                a3[s][t]=0;
                                a4[s][t]=0;
                                a5[s][t]=1;
                                a6[s][t]=0;
                        }


                        //Polynomial coefficients of HHP (constant term taken care of in function "f")
                        //c[#] is the coefficient for (Pij)^(#+1)
                        const T A_A=1/(RATIO*P_OP*P_OP);
                        const T P_RET=RATIO*P_OP;

                        c[0][s][t]=A_A*P_OP*P_RET;
                        c[1][s][t]=-A_A*(P_OP+P_RET);
                        c[2][s][t]=A_A;
                }
        }

        //Left Boundary P=0
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;
```

```
        }

        //Right Boundary P=P_OP
        for(int t=1;t<=N_MAX_Y-4;t++)
        {
                int s=N_MAX_X-3;

                bR1[t]=0;
                bR2[t]=1;
                bR3[t]=0;
                bR4[t]=0;
                bR5[t]=0;
                bR6[t]=P_OP;
        }

        //Bottom Boundary P=P_OP
        for(int s=1;s<=N_MAX_X-4;s++)
        {
                int t=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Boundary
        //Inside light cylinder Pz=0
        for(int s=1;((s<N_LC-1)&&(s<=N_MAX_X-4));s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=0;
                bT4[s]=1;
                bT5[s]=-1;
                bT6[s]=0;
        }

        //Outside light cylinder P=P_OP which is specified from the
           start.
        for(int s=N_LC-1;s<=N_MAX_X-4;s++)
        {
                int t=N_MAX_Y-3;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=0;
                bT4[s]=1;
```

```
                bT5[s]=0;
                bT6[s]=P_OP;
        }

        //Bottom Left Corner
        {
                int s=0;
                int t=0;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bB1[s]=0;
                bB2[s]=0;
                bB3[s]=1;
                bB4[s]=0;
                bB5[s]=0;
                bB6[s]=P_OP;
        }

        //Top Left Corner
        {
                int s=0;
                int t=N_MAX_Y-3;

                bL1[t]=1;
                bL2[t]=0;
                bL3[t]=0;
                bL4[t]=0;
                bL5[t]=0;
                bL6[t]=0;

                bT1[s]=0;
                bT2[s]=0;
                bT3[s]=0;
                bT4[s]=1;
                bT5[s]=-1;
                bT6[s]=0;
        }

        //Bottom Right Corner
        {
                int s=N_MAX_X-3;
                int t=0;

                bR1[t]=0;
                bR2[t]=1;
                bR3[t]=0;
```

```
        bR4[t]=0;
        bR5[t]=0;
        bR6[t]=P_OP;

        bB1[s]=0;
        bB2[s]=0;
        bB3[s]=1;
        bB4[s]=0;
        bB5[s]=0;
        bB6[s]=P_OP;
    }

    //Top Right Corner
    {
        int s=N_MAX_X-3;
        int t=N_MAX_Y-3;

        bR1[t]=0;
        bR2[t]=1;
        bR3[t]=0;
        bR4[t]=0;
        bR5[t]=0;
        bR6[t]=P_OP;

        bT1[s]=0;
        bT2[s]=0;
        bT3[s]=0;
        bT4[s]=1;
        bT5[s]=0;
        bT6[s]=P_OP;
    }
}

//This function inserts the star.
void star(void)
{
    for(int s=0;s<N_S_X;s++)
    {
        const T R=DX*i;
        for(int t=0;t<N_S_Y;t++)
        {
            const T Z=DY*j;
            a1[s][t]=0;
            a2[s][t]=0;
            a3[s][t]=0;
            a4[s][t]=0;
            a5[s][t]=1;
            a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
        }
    }
}
```

```
//This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI" and may or may not include both linear and nonlinear terms.
__forceinline void hhpSet(T** HHP,T** HHP_PRIME)
{
	T KH=6.6;
	if (!toggle)
	{
		printf("Takamori\n");
		for(int x=0;x<N_MAX_X-2;x++)
		{
			for(int y=0;y<N_MAX_Y-2;y++)
			{
				if(v0[RN(x,y)]>=P_OP)
				{
					HHP[x][y]=0;
					HHP_PRIME[x][y]=0;
				}
				else if (v0[RN(x,y)]<=0.0){HHP[x][y]=0;v0[RN(x,y)]=0.0;}
				else
				{
					T BETA=1+(KH-2)*log(fabs(v0[RN(x,y)]/P_OP));
					HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA*v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x,y)]/P_OP-2);
				}
			}
		}
	}
	else
	{
		printf("CKF\n");
		T HHP_L[N_MAX_Y-2];
		for(int KP=0;KP<N_MAX_Y-2;KP++)
		{
			HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
		}

		//at r<rL, there are many possible cases (closed field lines, open field lines that do or do not cross the light cylinder)
		//for 0<r<=NR_S, we stay OUTSIDE star, because the star value is known and does not need to be altered.
		for(int jj=0;jj<=N_S_X-1;jj++)
		{
			for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
			{
```

```
T BETA=1+(KH-2)*log(fabs(v0[RN(jj,kk)]/P_OP));
int KP=0;
{
        const T PT=v0[RN(jj,kk)];

        if(PT<v0[RN(N_LC-2,0)])
        {
                if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
                {
                        HHP[jj][kk]=KH*KH*0.5*v0[RN(jj,kk)]*(BETA*v0[RN(jj,kk)]/P_OP-1)*(BETA*v0[RN(jj,kk)]/P_OP-2);
                }
                else
                {
                        for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                        const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                        const T Q2=PT-v0[RN(N_LC-1,KP)];
                        HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                }
        }
        else
        {
                HHP[jj][kk]=0;
        }
```

```
				}
				if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
			}
		}

		//for NR_S<r<rLC
		for(int jj=N_S_X;jj<=N_LC-2;jj++)
		{
			for(int kk=0;kk<=N_MAX_Y-3;kk++)
			{
				T BETA=1+(KH-2)*log(fabs(v0[RN(jj,kk)]/P_OP));
				int KP=0;
				{
					const T PT=v0[RN(jj,kk)];

					if(PT<v0[RN(N_LC-2,0)])
					{
						if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
						{
							HHP[jj][kk]=KH*KH*0.5*v0[RN(jj,kk)]*(BETA*v0[RN(jj,kk)]/P_OP-1)*(BETA*v0[RN(jj,kk)]/P_OP-2);
						}
						else
						{
							for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

							const T Q1=PT-v0[RN(N_LC-1,KP+1)];
							const T Q2=PT-v0[RN(N_LC-1,KP)];
							HHP[jj][kk]=(Q1*HHP_L[
```

```
                                        KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                                }
                        }
                        else
                        {
                                HHP[jj][kk]=0;
                        }
                }
                if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
}

for(int jj=N_LC-1;jj==N_LC-1;jj++)
{
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
                HHP[N_LC-1][kk]=HHP_L[kk];
        }
}

for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
{
        for(int kk=0;kk<=N_MAX_Y-3;kk++)
        {
                int KP=0;
                const T PT=v0[RN(jj,kk)];

                for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++);

                const T Q1=PT-v0[RN(N_LC-1,KP+1)];
                const T Q2=PT-v0[RN(N_LC-1,KP)];
                //if(fabs(Q1-Q2)>0.00001)
                {
                        HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP+1])/(Q1-Q2);
                }
                if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
        }
}

//TODO - HHP_PRIME is largely unnecessary, and this calculation may be wrong.
for(int x=0;x<N_MAX_X-2;x++)
{
        for(int y=0;y<N_MAX_Y-2;y++)
        {
```

```
                T part1,part2;
                if(x==N_MAX_X-3)
                {
                    part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(x,y)]-v0[RN(x-1,y)])/(DX*DX);
                }
                else
                {
                    part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(x+1,y)]-v0[RN(x,y)])/(DX*DX);
                }
                if(y==N_MAX_Y-3)
                {
                    part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(x,y)]-v0[RN(x,y-1)])/(DY*DY);
                }
                else
                {
                    part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(x,y+1)]-v0[RN(x,y)])/(DY*DY);
                }
                HHP_PRIME[x][y]=part1+part2;
            }
        }
    }
}

//This represents the part of HHP that is a function of "R" and "Z" (NOT "PSI"). This includes any possible constant term.
__forceinline T f(int m, int n)
{
    return 0;
}

#undef i
#undef j
```

BIBLIOGRAPHY

[1] Scharlemann, E. T., Wagoner, R. V. 1973, ApJ, 182, 951

[2] Contopoulos, I., Kazanas, D., Fendt, C. 1999, ApJ, 511, 351

[3] Takamori, Y., Okawa, H., Takamoto, M., Suwa, Y. 2014, PASJ, 66, 25

[4] Lovelace, R. V. E., Turner, L., Romanova, M. M. 2006, ApJ, 652, 1494

[5] Michel, F. C. 1973, ApJ, 180, L133

[6] Ogura J., Kojima Y. 2003, Prog. Theor. Phys., 109, 619

[7] Contopoulos, I., Kalapotharakos, C., Kazanas, D. 2014, ApJ, 781, 46

[8] Petrova, S. A. 2013, ApJ, 764, 129