Non-Dipolar Magnetic Field Models and Patterns of Radio Emission: Uranus and Neptune Compared

Dr. D. R. Evans
Radiophysics, Inc.
5475 Western Ave.
Boulder CO 80301

A condensed version of this report is to be submitted to Geophysical Research Letters
ABSTRACT

The magnetic field geometries of Uranus and Neptune are superficially similar, and are similarly unlike those of other planets: the field strengths are similar, and they contain extraordinarily large non-dipolar components. As a corollary, the best dipolar field models of each of the two planets comprises a dipole that is considerably offset from the planetary centre and tilted away from the rotational axis.

However, in other respects the best field models of the two planets are quite different. Uranus has a quadrupole model in which all the terms are well determined and in which none of the higher order terms is determined. To represent the magnetometer data acquired during Voyager's Neptune encounter requires a model of order 8 (instead of Uranus' order 2), yet many of the coefficients are poorly determined. A second model, an octupole model comprising the terms up to order three of the order 8 model, has been suggested by the magnetometer team as being useful; its use, however, is limited only to the region outside of about 2 Rₚ, whereas planetary radio emissions have their sources well inside this surface.

Computer code has been written that permits an analysis of the detailed motion of low energy charged particles moving in general planetary magnetic fields. At Uranus, this code reveals the existence of an isolated region of the inner magnetosphere above the day side in which particles may be trapped, separate from the more general magnetospheric trapping. An examination of the so-call ordinary mode uranian radio emissions leads us to believe that these emissions are in fact extraordinary mode emissions coming from particles trapped in this isolated region.

A similar attempt to discover trapping regions at Neptune has proved, unfortunately, to be impossible. This arises from three factors: a) the computation needed to track particles in an eighth order field is more than an order of magnitude greater than that needed to perform a similar calculation in a quadrupole field, and is beyond the capacity of workstation-class computers; b) the octupole field model is known to be in error by too large an amount for it, or any similarly truncated version of the eighth order model, to produce trustworthy results; c) the eighth order model can, in effect, be infinitely varied without affecting the field strength along the spacecraft trajectory.
1 -- MOTION OF CHARGED PARTICLES

Decametric (and longer wavelength) radio emissions from the vicinity of a planet are caused by the acceleration of non-relativistic charged particles in the planetary magnetosphere. The exact mechanism by which most such emission is created is still a subject of some debate, although most planetary radio astronomers currently believe that some variant of the cyclotron maser mechanism is responsible. A good review of many of the competing theories is given in chapter 9 of Dessler (1983).

One common feature of the mechanisms is that they are all coherent. This requirement is demanded by virtue of the high intensities observed in planetary radio emissions. Because of this coherence, and also the remarkable degree of repeatability of most such emissions, it appears likely that the great majority of emissions come from coherently coupled groups of particles trapped inside planetary magnetospheres.

1.1 -- MOTION IN A PLANETARY MAGNETIC FIELD

A charged particle emits radiation whenever it undergoes an acceleration; that is, whenever it is subject to a force. In general, both electrical and magnetic forces may act on a charged particle (we ignore collisions here and throughout this discussion except where we specifically state otherwise). The equation of motion of a charged particle subject to an electrical field \( E \) and a magnetic field \( B \) is given by the deceptively simple equation:

\[
F = qE + q(v \times B)
\]

where:

- \( F \) is the force experienced by the particle;
- \( q \) is the particle's charge;
- \( v \) is the particle's velocity.

For the remainder of our discussion, we will assume that the electrical field is zero, and hence the force on the particle is simply:

\[
F = q(v \times B)
\]

The ramifications of this simple equation are discussed at length in a number of elementary textbooks (such as Roederer, 1970) and will only briefly be reviewed here.

A particle moving in a general magnetic field can experience three levels of periodicity in its motion, if only average particle positions are considered. (That is, if one is not interested in the very small scale, precise position of the particle.) At the lowest, most rapid, level, is the particle's cyclotron motion, the periodicity of the motion perpendicular to the local magnetic field. At an intermediate level is the possibility of a bounce motion, along a line of force. At the highest, slowest level, is the particle's drift motion, along a closed surface of field lines and instantaneously perpendicular to those field lines. We say that a particle is trapped if it exhibits all three kinds of motion. To put it another way, once a particle finds itself in a state in which it exhibits all three periodic kinds of motion, it will continue to traverse the same magnetic field surface unless something external affects the particle's motion. (Such as, for example, a collision with another particle, or energy loss through radiation.)

---

1 See, for example, Wu and Lee (1979).
Not all magnetic field topologies permit the trapping of particles, and, of those that do, not all particles moving in the field will be trapped. A dipole magnetic field, which is, on the largest scale, the appearance of a typical planetary field, does permit such trapping.

1.2 -- DIPOLE MAGNETIC FIELDS

It is easy to show that for a particle moving in a conservative magnetic field, the magnetic moment of the particle,

$$\mu = (\sin^2 \alpha / |B|)$$

(where $\alpha$, the angle between the particle's velocity and the local magnetic field, is called the pitch angle) is a constant of the motion.

In a dipole field, a particle initially starting with a pitch angle different from 90° will travel along a field line, its pitch angle changing continuously, until it reaches a point (providing that the planet's body, a satellite, or an ionosphere does not intervene) where the strength of the local field constrains its pitch angle to be zero. At this point there is a Lorentz force parallel to the field line in such a direction that the particle moves back in the direction it has come. Thus a bouncing motion is set up, symmetrical about the magnetic equator.

As the particle bounces along a field line, the curvature of the magnetic field also induces a slower drift in a direction perpendicular to the field line. In the case of a purely dipolar field, a charged particle simultaneously performs rapid gyro movement, a slower bounce up and down field lines, always mirroring at the same latitude (north and south) and slowly drifts around the field. The locus of this motion traces out a symmetric surface in space. This surface is known as an L shell. It has an associated number, given by the ratio of the distance of the particle from the planet's centre at the point that it crosses the magnetic equator to the planetary radius. (That is, L shells always have values greater than unity.)

1.3 -- NON-DIPOLAR MAGNETIC FIELDS

The simple situation described above can become considerably more complex when charged particles move in planetary fields that are non dipolar. One basic condition still holds: if a particle exhibits motion that returns it to its starting point, it is trapped. In the dipole case, all trapping regions are essentially similar: they are symmetrical about the equator and rotationally symmetric about the axis of the dipole. Neither of these is generally true when the field is non-dipolar.

Non-dipolar magnetic fields are usually specified by the terms of a spherical harmonic expansion of the form:

$$V = a \sum_{m=1}^{\infty} \left\{ \left( \frac{z}{a} \right)^n T_n^r + (\frac{z}{a})^{n+1} T_n^t \right\}$$

where $V$ is the scalar potential from which $B$ can be derived ($B = -\nabla V$);

$a$ is the equatorial radius of the planet;

and $T_n = \sum_{m=0}^{n} \{ P^m_n (\cos \theta [g^m_n \cos (m\phi) + h^m_n \sin (m\phi)] \}$;
The $T_n^m$ represent external currents and so can be ignored. The $P_n^m \cos \theta$ are Schmidt normalised associated Legendre functions of degree $n$ and order $m$, and the $g_n^m$ and the $h_n^m$ are the Schmidt coefficients. (For a considerably more detailed discussion, see the earlier monthly reports).

There is (in general) no way in which a mere inspection of the terms of such an expansion can provide insight into the existence and topography of trapping regions within that field. The only guaranteed way in which to show the existence of such regions is to search exhaustively for them by means of a computer program.

The writing of such a program is nontrivial. Many of the assumptions and simplifications that are valid in the dipole case are not so for non-dipolar fields (although it is often not obvious why this is so). Consequently, particle motion, except for the very smallest level of motion, the gyro motion, needs to be calculated and followed to determine whether, eventually, the particle returns to its initial starting point.

The author has written such a program, and it is provided in Listing I. Ancillary programs are provided in Listings II and III.

The basic functioning of the program can be quite simply described. Input comprises the latitude, longitude and pitch angle of a particle (assumed to be a 1 eV electron). The program considers a point on the surface of the planet (making allowance for oblateness) described by the latitude and longitude. The field line that intersects the planetary surface at that point is then traced until a point is reached (the first such point) where the field strength is at a local minimum. The particle is moved to that point, provided with the given pitch angle and 1 eV of energy, and then released. The program follows the particle's subsequent motion.

The motion may be quite complex. In particular, two facts lead to the likelihood of increased complexity over the simple dipolar case: the non monotonic decrease of field strength along a radius and the presence of multiple mirror equators. We discuss these two items separately.

1.3.1 -- THE DECREASE OF FIELD STRENGTH WITH DISTANCE

It is tempting to extrapolate the dipole case, where moving out along a radius leads to a monotonically decreasing local field, to the non-dipolar case; but to do so would be an error. Indeed, it is possible to construct entirely feasible planetary field models at which the field strength some distance above the surface of the planet goes to zero.

How this can come about can be visualised quite simply. Consider a field model with only two components, one dipolar and one of considerably higher order. Physically, every coefficient in the spherical harmonic expansion corresponds to one or more current loops beneath the planet's surface. The higher the order represented by the coefficient, the closer to the surface its equivalent current loops lie. One can easily imagine a pair of current loops of opposite sign, with the smaller valued current lying closer to the surface. Now, since one loop is located at the centre of the planet and one loop some distance towards the surface, it is quite clear that the field strength at and above the surface near the second current loop will be lower than at other points on the
surface. If the second loop is sufficiently close to the surface, it is even possible that the magnetic field will reverse in direction at the surface, being dominated by the field from the second, closer loop. Yet, far from the planet along the same radius, the field from the larger loop at the planet's centre will dominate. Ergo, there must be some point in between at which the two fields exactly cancel out.

This is not simply a theoretical exercise. Figure 1 shows a slice through the neptunian I8E1 44ev model of Connerney et al., 1991. The slice begins at latitude 30°, longitude 210° (measured in the positive δ direction) and ends at latitude -60°, longitude 330°. The strength of the model field from the surface to a distance of 1 radius above the surface is shown. Two distinct regions of minimal field strength are obvious. The lowest of them drops very close (perhaps all the way) to zero.

Figure 1

Such figures make it clear why simplistic notions such as L shells and monotonically decreasing field intensity with distance are invalid in general, non-dipolar, planetary magnetic fields.

1.3.2 -- MULTIPLE MIRROR EQUATORS

We define the term "mirror equator" as follows: a mirror equator is the locus of points of local minimum of field strength along adjacent field lines.
In a dipole field, there are an infinite number of mirror equators, all confined to (and defining) a sheet perpendicular to the dipole axis and passing through the equator. Any given field line intersects exactly one mirror equator, at the dipole's equator, and the field line is always perpendicular to the mirror equator.

In a non-dipolar field, none of these is necessarily true. Because of the relatively convoluted shape of non-dipolar fields, it is frequently the case that a single field line exhibits more than one local minimum along its length; the mirror equator corresponding to that minimum may be a considerable distance from any plausible definition of the field's equator; and the field line may intersect the equator non-perpendicularly.

In a dipolar field, all mirror equators are necessarily closed lines; in fact, they are constrained to be circles about the dipole axis. Neither of these constraints exists for non-dipolar fields. One can easily imagine a case in which a particular field line exhibits a minimum along its length at a particular point, but an adjacent field line shows no such minimum. Thus, mirror equators in non-dipolar fields may have ends that simply float in space. They may also intersect the planet's body, leading to ends of a different kind.

Mirror equators are fundamental to the problem of trapping, for a trapped particle must necessarily pass through a mirror equator on every bounce. It is at that point that its pitch angle is a maximum. We note that that maximum pitch angle may vary as it slowly drifts across adjacent field lines, and that because of the conservation of magnetic moment, a particle's pitch angle may determine whether a particular mirror equator appears as a closed loop to that particle. Bifurcations in mirror equators are also possible, and the precise path taken by a charged particle at such points is dependent on its energy, pitch angle and even the direction in which it is travelling along the field line. Such minutiae are unmodellable with any degree of rigour and, in any case, are well beyond the confidence with which we should hold non-terrestrial planetary magnetic field models. We therefore simply regard particles that travel close to the ends or bifurcations of mirror equators as untrapped.

For a particle to be trapped within a particular region of the magnetosphere, there must exist a closed mirror equator within that region. If such an equator exists, then a low energy particle with a pitch angle at the mirror equator sufficiently close to 90° will be trapped.

A particle may pass though a mirror equator caused by a local minimum in field strength but have a pitch angle sufficiently different from 90° that it can pass into a different region of the magnetosphere than one with a pitch angle closer to 90°. Therefore, in non-dipolar fields, loss cones come about not merely because of the planet's surface, but also because of the detailed topography of the magnetic field.

---

1 In general, the notion of an "equator", as indeed any notion of magnetic latitude or longitude, is invalid in a non-dipolar field; unfortunately, this rather obvious fact is frequently ignored in the literature.
Thus, in searching for possible trapping regions (which really becomes a search for closed mirror equators), it behooves one to specify an initial pitch angle, at a mirror equator, close to 90°, as particles with pitch angles much different may not be trapped at all. For our computer simulations, then, we typically chose a pitch angle of 89°. Of course, unless the pitch angle of real trapped particles can range to quite different values, over say several tens of degrees, the number of particles trapped is unlikely to be sufficient to support detectable emission. However, after a closed mirror equator has been detected in a model field, one can then vary the pitch angle accordingly to determine the robustness of the mirror equator.

2 -- URANUS

The best uranian magnetic field model is the \(Q_3\) model of Connerney et al. (1987). This is a quadrupole (i.e. order 2) magnetic field model, and, as such, displays relatively little complexity. We note that the actual field at Uranus may exhibit substantial higher order terms. The \(Q_3\) model is simply the most exiguous model that fits the Voyager magnetometer data to within the instrumental errors. A closer flyby (or a more accurate instrument) could well have resolved higher order moments.

The \(Q_3\) model is essentially dipolar, especially in the nightside hemisphere. The relatively high values of the quadrupole coefficients come about because of the large offset and tilt of the essentially dipolar field. (The first uranian magnetic field model (of Ness et al., 1986) was an offset, tilted dipole model in which the dipole was offset by approximately 0.3 \(R_U\) from the planetary centre, and tilted some 60° from the planet's axis of rotation; the axis of this dipole intersected the planet at +15.2°, 47.7°W and -44.2°, 227.5°W.)

In the dayside hemisphere, the \(Q_3\) model has an anomalous, secondary surface dip equator bounding the area roughly from 290° to 60° longitude and 35° to 85° latitude (see Figure 7 of Connerney et al., 1987). Figure 2 is a view of the night side of Uranus, with its essentially dipolar field. Figure 3 is a similar view of the day side, and the fact that the dayside field is not particularly dipolar is quite obvious. The driver computer programs used to produce these Figures is provided in Listing IV.

In reference to their Figure 7, Connerney et al. (1987) refer to the secondary dip equator and correctly state (in different words) that this divides the inner magnetosphere into two distinct, separate regions, in the smaller of which "field lines are trapped close to the planet's surface and are essentially isolated from the rest of the magnetosphere". They fail, however, to draw the corollory: that this permits the possibility of a second magnetic equator, and thus a separate and distinct region of the magnetosphere in which particles might be trapped.

(Note that the presence of a second dip equator does not imply a second trapping region; it merely permits such a region.)

1 For the sake of brevity, the ancillary programs, which run to approximately two thousand lines of code, are not included in this report.
In order to determine whether a second trapping region actually exists, an exhaustive search of the space surrounding the planet must be undertaken. Such a search involves constructing a fine
meshed grid and permitting low energy particles with pitch angles near 90° to move according to the laws of motion described above.

A search of this nature requires a prodigious quantity of computation. The search can be rendered somewhat more tractable by limiting oneself to a curved two dimensional grid (the surface of the planet) rather than the entire three dimensional grid in which the planet is enmeshed. Such a simplification does not materially affect the fineness of the search, because in the three dimensional case many of the points lie on or near field lines on which other points on the mesh lie. Using the surface of the planet eliminates this quasi-redundancy.

The procedure followed, then, was to cover the planet with a fine grid of points, and then, for each such point to "walk" down the associated field line until a minimum (not necessarily a global minimum) was reached. At that point a particle was released with an 89° pitch angle, and its motion was followed. The minimum of the field line is on a closed mirror equator if the particle returns to its original position.

This process consumed a large amount of computer time. Most grid points led to the dipole mirror equator, of course. However, a second mirror equator, associated with the anomalous dayside region, was also located. The surface boundary of this equator is an elongated oval, approximately bounded in latitude by 39° and 42° and in longitude by 60° to 130°. This equator, in the $Q_3$ model, is barely a closed loop for particles with a pitch angle of 89°. The three dimensional area in which particles with smaller pitch angles are trapped is limited to less than 10° in longitude at the surface. However, the mirror equator barely drops into the ionosphere of the planet, and only a very small change in the model (corresponding to a small internal current loop near the surface) would cause the equator to be closed over a considerably wider range of pitch angles. Such a change would be undetectable at the distance at which Voyager flew past the planet and would have no effect on particle motion in the remainder of the magnetosphere.

Figure 4, which is an overhead view looking down at Uranus from latitude 40° and longitude 60°, shows this region. The two dark areas are trapped regions for particles with pitch angles of 89°. The curved hook represents the paths of typical particles with an equatorial pitch angle of 70° emanating from the heart of the trapped region. Particles with pitch angles less than this collide with the planet and are not trapped.
Prior to the Uranus encounter, low frequency (< 100 kHz) radio emissions were detected by the Voyager Planetary Radio Astronomy (PRA) instrument. These were quite different from the much stronger nightside emissions detected at and subsequent to closest approach (at 1800 hours on day 24 of 1986) which corresponded to particles moving in the larger cavity of the magnetosphere. Desch and Kaiser (1987), relying on the fact that this emission was observed to be left hand polarized, have posited that it is ordinary mode emission "via a direct generation process" of which no details are given. Desch and Kaiser used an offset, tilted dipole model (in which, of course, there is no possibility of a secondary mirror equator) and they state that the consequences of the $Q_3$ model "have not been investigated" but nevertheless opine that their conclusion "seems firm".

However, propagating ordinary mode emissions have never been observed coming from a giant planet, and there are good theoretical reasons for believing that such emissions are, at best, extremely difficult to produce in nature. Additionally, their conclusion rests solely on purely dipolar notions of trapping and mirroring, in the very region in which the uranian magnetosphere is not dipolar. Given the results of our search for a secondary mirror equator, a much more likely scenario presents itself: the dayside emissions detected by PRA prior to the uranian encounter are the result of particles trapped in a small region on the dayside. The polarity of the local field is opposite to that of the main field in the surrounding regions, and so left hand polarised emissions coming from such a region are extraordinary mode (that is, in the somewhat confusing terminology of magneto-ionic theory, they are the usual mode of excitations of planetary radio emissions).

Kaiser and Desch go to considerable length to separate their putative ordinary mode emission from the more general, lower frequency so-called "smooth narrow band" emissions¹. Figure 5 shows the PRA data from midnight to 1600 hours on day 24. According to Kaiser and Desch, the emission in channel 194 from approximately 0300 to approximately 0800 is not the same as that in channel 195, a mere 19 kHz (or 15% in frequency) away. Figure 5 indicates that there is no

¹ A name which is misleading, since they are certainly not narrow band.
particular reason to make this distinction. Further, they lend great weight to the observed time of commencement (but not cessation, which does not fit their model particularly well) of the emissions. However, the field line geometry predicted by the $Q_3$ model is complex, and the relative geometry of the spacecraft and the trapping region is changing rapidly throughout this period, so it is eminently reasonable to expect different aspects of the same emission to make themselves visible over these hours.

However, even if one (unnecessarily) concedes the notion that the higher frequency emission is intrinsically different from the lower frequency, the weight of evidence suggests that it is extraordinary mode emission coming from a small, isolated region of the inner magnetosphere around 40° latitude, 60° longitude, rather than some exotic ordinary mode emission associated with a dipole model that is known to be incorrect in the very region in question.

Their strongest "evidence" comes from a measurement of the ellipticity of the emissions seen below channel 195 as compared to that seen above that channel, however they make no attempt to examine the instrumental response at different frequencies to signals with known ellipticity.

Figure 5
3 -- NEPTUNE

The situation at Neptune is vastly more complicated than is that at Uranus. Much of this added complexity is a result merely of the greatly different encounter geometry, and is (probably) not intrinsic.

At Uranus, the encounter geometry was such that all the dipole and quadrupole coefficients of the spherical harmonic expansion could be extracted from the observational dataset with relative ease. Further, the spacecraft's closest approach to Uranus was sufficiently distant that no coefficients of higher order could be reliably determined. Consequently, the Voyager magnetometer team presented the community with a relatively "clean" magnetic field model of order two, in which higher order coefficients were undetermined. The situation at Neptune was quite different.

At Neptune, the encounter geometry was such that the spacecraft entered and exited the system from the southern hemisphere; but, during the actual encounter period, the spacecraft flew extremely close to the northern pole of the planet (closest approach was a mere 1.18 R_N, as opposed to a closest approach distance of 4.2 R_U at Uranus. This resulted in great difficulties for the magnetometer team in producing the best field model (and, moreso, great conceptual difficulties for those trying to use the model provided by Connerney et al.; the literature contains several papers that show that neither their authors nor the papers' referees gave Connerney et al.'s paper the careful reading it deserves and requires).

Because of the peculiar nature of the neptunian encounter geometry, the paper presenting the magnetic field models is lengthy and must be read very carefully. The paper is impossible to précis here, and a discussion has already been given in a prior monthly report, so we shall here merely summarize the points that are most important for our purposes.

Connerney et al. took the magnetometer data and then systematically inverted the data to extract the coefficients of a spherical harmonic expansion, limiting the order to some number N. They then compared the results of the ensuing Nth order model to the observed data. They repeated this process, gradually increasing N, until they reached a point at which the model field matched the observed field to within the known accuracy of the magnetometer. This required a much higher order model than for any of the other giant planets (because the close flyby permitted higher order moments to be detected). However, many coefficients (corresponding to fields in the southern hemisphere) were poorly constrained (if they were constrained at all). Thus the model that they obtained and published, the I8E1 44ev model, while sufficient to reproduce the field along the spacecraft's trajectory, could be (and almost certainly is) in considerable error at distances far from the trajectory. In addition, the I8E1 44ev model suffers from the fact that because it is an 8th order model, it requires a large number of relatively complicated calculations to be made in order to calculate the field strength at a single point. In this regard it is more than ten times as complex as the uranian quadrupole field.

---

1 See the program MAGFIELD.CC and its associated header file, MAGFIELD.H, in Listing 11.
Because of these difficulties, Connerney et al. proposed a second model, the \( O_8 \) model, which is a pure octupole model. However, this model has severe restrictions, which are described in the paper but have generally been ignored by the community. In the first place, the \( O_8 \) model is not in any sense the best octupole model of the neptunian field. It is not formed by inverting the data and constraining all coefficients higher than order three to be zero (which is the procedure used to minimise the difference between model predictions and the observed data). Rather, it is simply the I8E1 44ev model truncated at order 3, which is quite a different matter. Additionally, Connerney et al. show clearly that the \( O_8 \) model, while predicting field strength outside \( 2R_N \) quite well, is not to be relied on inside that distance. Low energy trapped radiation, of course, is almost guaranteed to come from well inside \( 2R_N \).

It is tempting to do as other authors have done and to simply use the \( O_8 \) model for calculations. However, a simple comparison of the two models show that this would be inappropriate.

On a global scale, the two models appear little different. Figures 6 and 7 show "wire model" views looking at the northern and southern axis of the neptunian dipole model using the \( O_8 \) model; Figures 8 and 9 show the same views, but with the I8E1 44ev model.

\footnote{This dipole is offset from the planetary centre by 0.55 \( R_N \) and has a tilt of 46.8°. For more details, see Ness et al., 1989.}
Figure 7

Figure 8
There is almost no discernible difference between these pictures except in the details in the inner magnetosphere. However, one must remember that up to and including order three, the models are identical. Therefore it should come as no surprise that global views such as these, which are dominated by the low order terms, appear almost the same. A better understanding of the difference between the models is provided by Figures 10 through 15. These show the ratios of the two models at a series of distances above the planet's surface. As is to be expected, in general (with one exception), the further one rises in altitude, the more similar the two models become. This is simply a reflection of the fact that in a spherical harmonic expansion, higher order terms can be thought of as resulting from smaller current loops closer to the planet's surface. Thus, higher order terms tend to have a more local effect than lower order terms. At altitudes below about 1.6 $R_N$, the difference between the two models is generally several tens of percent, and sometimes even more.

Figure 16 shows even more clearly the necessity of using the most accurate field model for purposes of searching the inner magnetosphere for possible trapping regions. It shows the same cross section as does Figure 1, except for the $O_8$ model instead of Figure 1's I8E1 44ev model. The contours close to the surface are completely different in the two cases. In particular, the above-ground decrease in magnetic field intensity that was so much a feature of Figure 1 is completely absent in Figure 16. Thus, regions in which trapping might appear in the I8E1 44ev model will be completely absent of such trapping in the $O_8$ model.
Figure 12

Figure 13
Figure 14

Figure 15
Clearly, then, using the $O_8$ model to model the field is not a promising prospect, no matter how attractive.

One could arbitrarily cut off the I8E1 44ev model at some order higher than three but lower than eight, thus rendering the complexity of the ensuing model somewhere between the two extremes. However, there seems no rational way in which to decide where to place such a cut-off; further, given the obvious constraints in even the I8E1 44ev model, any lessening of the number of coefficients will simply lead to an even less useful model.

For several reasons, then, the procedure that worked so well at Uranus cannot be contemplated at Neptune: the field is so complicated that calculations with workstation class computers simply take too long to follow particle motions any distance; the complexity of the field leads to a large number of anomolous regions in the planetary magnetosphere; the discrepancy between the relatively accurate model in the northern hemisphere and (almost certainly) inaccurate model in the southern hemisphere means that trapping regions far from the path would likely by artifacts.

Yet another difficulty exists. Some thirteen eigenvectors are omitted from the I8E1 44ev model, on the grounds that they are not necessary to fit the observed data to within the instrumental accuracy. However, linear combinations of these, subject to constraints detailed by Connerney et al. (1991), may be added to the I8E1 44ev model without affecting the model field along the spacecraft trajectory. They note that poorly resolved and unresolved components of the model may
be substantially altered by such combinations. Obviously, changing the model so dramatically, will greatly change the possible trapping regions.

Despite these difficulties, an attempt has been made to uncover trapping regions near to the spacecraft path. However, with the computing facilities available, a relatively course surface grid had to be adopted, with a typical grid spacing of some 20°, and no such trapping regions have been discovered.

We therefore are forced to conclude that, for the present, a detailed examination of the neptunian magnetosphere for secondary trapping regions is not possible, although it would become so with increased computing power available.
Program to calculate the motion of a low energy electron in a general planetary magnetic field

// C++ code Copyright (C) David R. Evans G4AMJ/NQOI

// program to trace motion of charged particles in planetary magnetic
// fields

#include <defines.h>
#include <magfield.h>
#include <vector.h>
#include <math.h>
#include <iostream.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>

// here define either URANUS or NEPTUNE
#undef URANUS
#define NEPTUNE

// if ORDER3 is defined, then uses 08 instead of full I8E1 44ev
#define ORDER3

#ifdef URANUS
const float ellipticity = 0.0229;
#else
const float ellipticity = 0.017;
#endif
const int LIMIT = 4096;
magfield* field;

// define matherr to abort
int matherr(struct exception *x)
{
    cout << "MATH ERROR\n";
    exit(1);
}

// I calculate the following function: (latitude in degrees)
// from ellipticity = (re - rp) / rp.

float radius(const float co_latitude, const float equatorial_radius = 1)
{
    const float lat_in_radians = PI / 180;
    const float lat_in_radians = PI / 2 - co_latitude,
    C = cos(lat_in_radians),
    S = sin(lat_in_radians);
47     return sqrt(SQR(equatorial_radius) /
48             (C * C + S * S * SQR(1 + ellipticity))) ;
49 }
50
51
52 extern const cartesian uranus_q3(const spherical&);
53
54 const double  twopi = 2 * PI,
55 field_factor = 1.1,
56 circle_size = 0.01,
57 max_intermediate_points = 10,
58 min_distance_from_previous = 0.01,
59 default_horiz_epsilon = 0.04,
60 e = 1.602e-19,
61 m = 9.109558e-31,
62 c = 2.997925e8,
63 v_cg_factor = 256.0,
64 cycles_per_time_step = 1024.0,
65 min_distance_from_previous_output = 0.1;
66
67 const int  skip = 100,
68 up = 1,
69 down = -up;
70
71 double q = 1.602e-19,  // ATT does not permit "q = e"
72     total_time = 0, rotations_so_far = 0;
73
74 float target_field, target_frequency;
75
76 double pitch_angle, alpha, energy;
77
78 // move a distance along a field line
79 cartesian along_field_line(const cartesian& p, const int direction,
80     const double distance)
81 { const cartesian p1 = p + direction * distance * unit(field->field(p));
82     const cartesian pt = p1 - direction * distance * unit(field->field(p1));
83     const cartesian delta_p = pt - p1;
84     return p1 - 0.5 * delta_p;
85 }
86
87 double radius(const cartesian& p)
88 { const double scale = 0.001;
89     const cartesian p1 = along_field_line(p, up, scale);
90     const cartesian p2 = along_field_line(p, down, scale);
91     const double t1 = length(p1 - p2);
92     const double t2 = angle(p1 - p, p - p2);
93     // const double temp = length(p1 - p2) / (2 * angle(p1 - p, p - p2));
94     return length(p1 - p2) / angle(p1 - p, p - p2);
95 }
96
97 // find a mirror equator point
98 cartesian mirror_equator(const cartesian& init)
99 { const double max_distance = 1000, max_error = 0.001;
100     double delta = 0.01;
```
int direction = 1;
cartesian x_old = init, x_new;
cartesian old_field = field->field(init);
x_new = along_field_line(x_old, direction, delta);
cartesian new_field = field->field(x_new);

// start off in the correct direction -- i.e. lower B
if (length(new_field) > length(old_field)) then direction = -1;
while (delta > max_error) {
    // cout << spherical(x_old) << field->field(x_old) << "\n";
    x_new = along_field_line(x_old, direction, delta);
    new_field = field->field(x_new);
    // too far from planet?
    if (length(x_new) > max_distance) then return cartesian(O, O, 0);
    if (length(new_field) > length(old_field)) then
        if (delta > max_error) then
            direction = -direction;
            delta /= 2;
    }
    x_old = x_new;
    old_field = new_field;
}
return x_old;

cartesian correct_mirror_equator(const cartesian& x_init) {
    const double max_frac_error = 0.2,
               max_field_error = 0.001;
    cartesian xO = x_init;
    boolean close_enough;
    do
        xO = mirror_equator(xO);
        cartesian fieldO = field->field(xO);
        if (fieldO == O) then return cartesian(O, O, 0);
        // walk to a region in which the B is closer to the target
        double fractional_field_error = (length(fieldO) - target_field) /
                                        length(fieldO);
        if (fabs(fractional_field_error) > max_frac_error) then
            fractional_field_error = SIGN(fractional_field_error) * max_frac_error;
    
    // are we there yet?
    close_enough = (fabs(fractional_field_error) <=
                    max_field_error);
    if (!close_enough) then
        { double new_radius = 1 + fractional_field_error / 10;
          cartesian temp = xO;
          xO *= new_radius;
```
// check for area in which field decreases inwards or increases outwards
if (((SIGN(new_radius - 1)) !=
(SIGN(length(field0) / length(field->field(x0)) - 1))) then
    { new_radius = 1 / new_radius;
    new_radius *= new_radius;
    x0 *= new_radius;
    if (((SIGN(new_radius - 1)) ==
(SIGN(length(field0) / length(field->field(x0)) - 1))) then
        { cout << "Field shape problem\n";
    x0 = temp * (1 + (SIGN(fractional_field_error) * 0.5));
    }
    }
} while (!close_enough);

return x0;

// calculate parallel and perpendicular components of a field
void calculate_field_gradient(const cartesian& posn, cartesian& par, cartesian& per)
{
    const double grid_spacing = 0.001;
    cartesian b[6];
    b[0] = field->field(cartesian(x(posn) - grid_spacing / 2, y(posn),
    z(posn)));
    b[1] = field->field(cartesian(x(posn) + grid_spacing / 2, y(posn),
    z(posn)));
    b[2] = field->field(cartesian(x(posn), y(posn) - grid_spacing / 2,
    z(posn)));
    b[3] = field->field(cartesian(x(posn), y(posn) + grid_spacing / 2,
    z(posn)));
    b[4] = field->field(cartesian(x(posn), y(posn),
    z(posn) - grid_spacing / 2));
    b[5] = field->field(cartesian(x(posn), y(posn),
    z(posn) + grid_spacing / 2));
    cartesian del = cartesian(length(b[1]) - length(b[0]),
    length(b[3]) - length(b[2]),
    length(b[5]) - length(b[4]));
    del /= grid_spacing;
    const cartesian& b_field = unit(field->field(posn));
    par = _dot(b_field, del) * b_field;
    per = del - par;
}

int do_main(const double in_theta, const double in_phi)
{
    spherical s(1.0, in_theta, in_phi);  // the starting position
    cartesian output_posn;
    int n_out = 0, n_calculated = 0;

    // type_of_run is used to determine how often to produce output.
    // 'P' means a position-based decision; 'T' indicates a time-based one.
    const char type_of_run = 'T';

    // convert
energy = 1000;  // a 1 eV particle
alpha = pitch_angle *= PI / 180;  // radians

// we actually start from the mirror equator for the starting field line
cartesian posn = mirror_equator(cartesian(s));
target_field = length(field->field(posn));
target_frequency = target_field * 2.8e10;

const float seconds_per_step = cycles_per_time_step / target_frequency;
const int original_theta = (int)(t(spherical(posn)) * 180 / PI);
const int end_run_phi = (int)(p(spherical(posn)) * 180 / PI + 1) % 360;
const float magnetic_moment = length(field->field(posn)) / SQR(sin(alpha));
const float speed = sqrt(2 * e * energy / m) / (25000.0 * 1000);

cartesian posn_field = field->field(posn);
cartesian unit_field = unit(posn_field);
boolean finished;

// now let the electron go
do {
    cartesian grad_par, grad_per;
    calculate_field_gradient(posn, grad_par, grad_per);

    // calculate v_par -- assume conservative field
cartesian v_par = unit_field * speed * cos(alpha);

    // calculate v_cg
cartesian v_cg = unit_field * grad_per;
    double factor = m * SQR(speed) *
        (1 + SQR(cos(alpha))) / (2 * e * SQR(length(posn_field)));
    v_cg *= (factor * v_cg_factor);

    // f_par
    factor = - (SQR(speed * sin(alpha))) / (2 * length(posn_field));
cartesian f_par = factor * grad_par;

    // move the electron under constant acceleration
cartesian old_posn = posn;

    // guess at new position
cartesian guess_posn = posn + ((v_cg + v_par) * seconds_per_step +
        0.5 * (f_par) * SQR(seconds_per_step));
cartesian guess_field = field->field(guess_posn);
cartesian guess_unit_field = unit(guess_field);

    // note that grad_par and grad_per will reference the GUESSED position
    calculate_field_gradient(guess_posn, grad_par, grad_per);
    double guess_alpha = (length(guess_field) >= magnetic_moment ? PI / 2 :
        asin(sqrt(length(guess_field) / magnetic_moment)));
    int v_dirn = SIGN(_dot((v_par + f_par * seconds_per_step), guess_field));
    if (v_dirn < 0) then
        guess_alpha = PI - guess_alpha;
    double guess_factor = - (SQR(speed * sin(guess_alpha))) /
        (2 * length(guess_field));
cartesian guess_f_par = guess_factor * grad_par;

// take the mean
f_par = (f_par + guess_f_par) / 2;

// place v_par not quite in the parallel direction (the "field line" bends!!)
cartesian temp_v_par = unit(unit_field + guess_unit_field) * length(v_par);
v_par = ((dot(unit_field, v_par) < 0) ? -temp_v_par : temp_v_par);

// now do it for the drift as well
cartesian guess_v_cg = guess_unit_field * grad_per;
guess_factor = m * SQR(speed) *
               (1 + SQR(cos(guess_alpha))) /
               (2 * e * SQR(length(guess_field)));
guess_v_cg *= (guess_factor * v_cg_factor);

// take the mean
v_cg = (v_cg + guess_v_cg) / 2;

posn += ((v_cg + v_par) * seconds_per_step +
          0.5 * (f_par) * SQR(seconds_per_step));
posn_field = field->field(posn);
unit_field = unit(posn_field);
if (length(posn - old_posn) > 0.1) then
   {cout << "*** BIG MOVEMENT\n" << old_posn << posn << (v_cg + v_par) <<
    (v_cg + v_par) * seconds_per_step <<
    old_posn + (v_cg + v_par) * seconds_per_step << "\n";
}

// add fpar to give direction wrt B
alpha = (length(posn_field) >= magnetic_moment ? PI / 2 :
         asin(sqrt(length(posn_field) / magnetic_moment)));
v_dirn = SIGN(dot(v_par + f_par * seconds_per_step, posn_field));
if (v_dirn < 0) then
   alpha = PI - alpha;

total_time += seconds_per_step;
double distance_from_previous_output = length(posn - output_posn);
n_calculated++;

// stop if we hit the planet
if (r(spherical(posn)) < radius(t(spherical(posn)))) then
   return (-1);

// stop if we go too far from the planet
if (r(spherical(posn)) > 3.0) then
   return (-2);

// possibly print out a description of where we are
switch (type_of_run)
   { case 'P' : case 'p' :
     if (distance_from_previous_output >= min_distance_from_previous_output) then
        {cout << alpha * 180 / PI << " " << r(spherical(posn)) << " " <<
}
315 \( t(\text{spherical(posn)}) \times \frac{180}{\pi} \) << \\
\( p(\text{spherical(posn)}) \times \frac{180}{\pi} \) << \\
output_posn = posn; \\
n_out++; 
\}
break;
case 'T' : case 't' :
if (!((n_calculated % 100)) then 
    if (!((n_calculated % 1000)) then 
        cout << \( \alpha \times \frac{180}{\pi} \) << " " << \\
        \( r(\text{spherical(posn)}) \) << " " << \\
        \( t(\text{spherical(posn)}) \times \frac{180}{\pi} \) << " " << \\
        \( p(\text{spherical(posn)}) \times \frac{180}{\pi} \) << "\n";
        n_out++;
    
break;

finished = (n_out > 1000);
if ((type_of_run == 'T') || type_of_run == 't') then 
    finished = (n_out > LIMIT); 
    spherical_s_posn(posn); 
    finished = finished || ((n_out > LIMIT) && ((int)(p(s_posn) * 180 / \pi) == \\
    end_run_phi)); 
    finished = finished || ((n_out > LIMIT) && ((int)(t(s_posn) * 180 / \pi) == \\
    original_theta)); 
} while (!finished);
// spherical_s_posn(posn);
cout << "Finished, n_out = " << n_out << "\n\n";
// \( \text{nPhi} = \frac{180}{\pi} \times \text{end phi} = \frac{\text{end theta}}{\pi} \) 
// end_run_phi << "\n\Theta = \frac{180}{\pi} \times \text{end theta} = \frac{\text{original_theta}}{\pi} \) 
return 1;

main()
{ cout << "LIMIT = " << LIMIT << " cycles_per_time_step = " << \\
cycles_per_time_step << "\n\n";

// build the correct field model
#ifdef URANUS
const int order = 2, 
rank = 2;
// how many "g"s and "h"s are there? 
int n_coeffs = 0; 
for (int n = 1; n <= order; n_coeffs += ++n); 
int32 * g, * h;
heap_check(g = new int32 [n_coeffs]); 
heap_check(h = new int32 [n_coeffs]); 
// specify the model coefficients 
h[0] = 0; h[2] = 0;
g[0] = 11893;
g[1] = 11579;
h[1] = -15684;
g[2] = -6030;
g[3] = -12587;
g[4] = 196;

heap_check(field = new magfield(order, g, h));
#endif

#ifdef NEPTUNE
const int order = 8,
rank = 8;

// how many "g"s and "h"s are there?
int n_coeffs = 0;
for (int n = 1; n <= order; n_coeffs += ++n) {
  int32 * g, * h;
  heap_check(g = new int32 [n_coeffs]);
  heap_check(h = new int32 [n_coeffs]);
}

// specify the model coefficients
h[0] = 0; h[2] = 0; h[5] = 0; h[9] = 0;
h[14] = 0; h[20] = 0; h[27] = 0; h[35] = 0;

// n = 1
h[1] = -9889;
g[0] = 9732;
g[1] = 3220;

// n = 2
g[2] = 7448;
g[3] = 664;
h[3] = 11230;
g[4] = 4499;
h[4] = -70;

// n = 3
g[5] = -6592;
g[6] = 4098;
h[6] = -3669;
g[7] = -3581;
h[7] = 1791;
g[8] = 484;
h[8] = -770;

// n = 4
g[9] = 2243;
g[10] = 557;
h[10] = -1889;
g[11] = 3099;
g[12] = -1287;
h[12] = 1204;
g[13] = -5073;
h[13] = -456;

// n = 5
421  g[14] = -202;
422  g[15] = -229;
423  g[16] = 526;
424  g[17] = -2846;
425  g[18] = -1425;
426  g[19] = -2835;
427
428  // n = 6
429  g[20] = -2175;
430  g[21] = -466;
431  g[22] = -1269;
432  g[23] = -2233;
433  g[24] = -887;
434  g[25] = -496;
435  g[26] = 755;
436  // n = 7
437  g[27] = 1671;
438  g[28] = 1678;
439  g[29] = 1625;
440  g[30] = 2157;
441  g[31] = -483;
442  g[32] = 1873;
443  g[33] = 584;
444  g[34] = 664;
445  // n = 8
446  g[35] = -689;
447  g[36] = 238;
448  g[37] = -90;
449  g[38] = -1304;
450  g[39] = 311;
451  g[40] = -367;
452  g[41] = -249;
453  g[42] = 1333;
454  g[43] = -1239;
455  heap_check(field = new magfield(order, g, h));
456
457  // truncate if 08
458  // ifdef ORDER3
459  // endif
460  // endif
461
462  do { int dtheta, dphi;
463    cin >> dtheta >> dphi >> pitch_angle;
464    if (!cin.rdstate())
465      cout << "\n" << dtheta << " " << dphi << " " << pitch_angle << "\n";
466    cout << do_main(dtheta • PI / 180, dphi • PI / 180) << "\n\n";
467  } while (!cin.rdstate());
cout << "target frequency = " << target_frequency << "\n\n";
}
LISTING II
a) MAGFIELD.H

Header file of class for the calculation of magnetic fields up to order 8

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2 // a general purpose magnetic field class.
3 // the real trick here is to make it both general and *fast*.
4 // only const magfield should exist
5
6 #include <defines.h>
7 #include <vector.h>
8 #include <math.h>
9 #define real float
10
11 class magfield
12 { int _order, _max_order;
13   int _n_coeffs;
14
15   // the following is messy and ungeneral, but is necessary for speed
16   // (which is the most important requirement, as the field will need
17   // to be calculated _many_ times.)
18
19   int32 g10, g11,
20     g20, g21, g22,
21     g30, g31, g32, g33,
22     g40, g41, g42, g43, g44,
23     g50, g51, g52, g53, g54, g55,
24     g60, g61, g62, g63, g64, g65, g66,
25     g70, g71, g72, g73, g74, g75, g76, g77,
26     g80, g81, g82, g83, g84, g85, g86, g87, g88;
27
28   int32 h10, h11,
29     h20, h21, h22,
30     h30, h31, h32, h33,
31     h40, h41, h42, h43, h44,
32     h50, h51, h52, h53, h54, h55,
33     h60, h61, h62, h63, h64, h65, h66,
34     h70, h71, h72, h73, h74, h75, h76, h77,
35     h80, h81, h82, h83, h84, h85, h86, h87, h88;
36
37   real * _br, * _bt, * _bp;
38
39   // there are a whole slew of constants that we want to make static
40
41 public:
42   static /* const */ real sqrt3,
public:

// a magfield is constructed by one of the following mechanisms:

// 1. magfield(order, all the "g"s, then all the "h"s, which should be
    // int32s
    magfield(const int order, const int32, ...);

// 2. magfield(order, array of all the "g"s, array of all the "h"s
    magfield(const int order, const int32 const g,
             const int32 const h);

// 3. magfield(order, array of "g"s followed by "h"s)
    magfield(const int order, const int32 const gh);

// destructor
    ~magfield(void);

cartesian field(const spherical&) const;

// change the order to which the field is calculated
    inline int order(void)
    { return _order; }
    void order(const int n);
LISTING II
b) MAGFIELD.CC

Class for the calculation of planetary magnetic fields up to order 8

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2 // a general purpose magnetic field class
3 #include <magfield.h>
4 #include <stdarg.h>
5 // constructors
6 magfield::magfield(const int order, const int32,...) : _order(order)
7 { _max_order = _order;
8     va_list ap;
9     va_start(ap, order);
10    // how many "g"s and "h"s, are there?
11    _n_coeffs = 0;
12    for (int n = 1; n <= _order; _n_coeffs += ++n);
13    // allocate space for the components
14    _br = new real [_order + 1];
15    _bt = new real [_order + 1];
16    _bp = new real [_order + 1];
17    int32 * _g, * _h;
18    // allocate the space for the coefficients
19    heap_check(_g = new int32 [_n_coeffs]);
20    heap_check(_h = new int32 [_n_coeffs]);
21    // copy the coefficients from the parameter list to the internal arrays
22    for (n = 0; n < _n_coeffs; n++)
23       _g[n] = va_arg(ap, const int32);
24    for (n = 0; n < _n_coeffs; n++)
25       _h[n] = va_arg(ap, const int32);
26    va_end(ap);
27    // now build an easier way of referring to the arrays; we do this solely
28    // for speed later
29    int32 ** gp, ** hp;
30    heap_check(gp = new int32* [_order + 1]); // so the "gp"s go wrt 1
31    heap_check(hp = new int32* [_order + 1]);
32    int index = 0;
33    for (n = 1; n <= _order; n++)
34       { gp[n] = &_g[index];
35          index = &index;
hp[n] = &(_h[index]);
index += (n + 1);

// the following is incredibly unsubtle, but I can't think of a more elegant way to do it

if (_order > 0) then
{
    g10 = gp[1][0];
g11 = gp[1][1];
h10 = hp[1][0];
h11 = hp[1][1];
}

if (_order > 1) then
{
    g20 = gp[2][0];
g21 = gp[2][1];
g22 = gp[2][2];
h20 = hp[2][0];
h21 = hp[2][1];
h22 = hp[2][2];
}

if (_order > 2) then
{
    g30 = gp[3][0];
g31 = gp[3][1];
g32 = gp[3][2];
g33 = gp[3][3];
h30 = hp[3][0];
h31 = hp[3][1];
h32 = hp[3][2];
h33 = hp[3][3];
}

if (_order > 3) then
{
    g40 = gp[4][0];
g41 = gp[4][1];
g42 = gp[4][2];
g43 = gp[4][3];
g44 = gp[4][4];
h40 = hp[4][0];
h41 = hp[4][1];
h42 = hp[4][2];
h43 = hp[4][3];
h44 = hp[4][4];
}

if (_order > 4) then
{
    g50 = gp[5][0];
g51 = gp[5][1];
g52 = gp[5][2];
g53 = gp[5][3];
g54 = gp[5][4];
g55 = gp[5][5];
}
h50 = hp[5][0];
h51 = hp[5][1];
h52 = hp[5][2];
h53 = hp[5][3];
h54 = hp[5][4];
h55 = hp[5][5];

if (_order > 5) then
{ g60 = gp[6][0];
g61 = gp[6][1];
g62 = gp[6][2];
g63 = gp[6][3];
g64 = gp[6][4];
g65 = gp[6][5];
g66 = gp[6][6];
h60 = hp[6][0];
h61 = hp[6][1];
h62 = hp[6][2];
h63 = hp[6][3];
h64 = hp[6][4];
h65 = hp[6][5];
h66 = hp[6][6];
}

if (_order > 6) then
{ g70 = gp[7][0];
g71 = gp[7][1];
g72 = gp[7][2];
g73 = gp[7][3];
g74 = gp[7][4];
g75 = gp[7][5];
g76 = gp[7][6];
g77 = gp[7][7];
h70 = hp[7][0];
h71 = hp[7][1];
h72 = hp[7][2];
h73 = hp[7][3];
h74 = hp[7][4];
h75 = hp[7][5];
h76 = hp[7][6];
h77 = hp[7][7];
}

if (_order > 7) then
{ g80 = gp[8][0];
g81 = gp[8][1];
g82 = gp[8][2];
g83 = gp[8][3];
g84 = gp[8][4];
g85 = gp[8][5];
g86 = gp[8][6];
g87 = gp[8][7];
g88 = gp[8][8];
157  h80 = hp[8][0];
158  h81 = hp[8][1];
159  h82 = hp[8][2];
160  h83 = hp[8][3];
161  h84 = hp[8][4];
162  h85 = hp[8][5];
163  h86 = hp[8][6];
164  h87 = hp[8][7];
165  h88 = hp[8][8];
166  }
167  destroy_array(gp);
168  destroy_array(hp);
169  destroy_array(g);
170  destroy_array(h);
171  }
172
173  magfield::magfield(const int order, const int32 * const g,
174                      const int32 * const h)
175  { _max_order = order;
176    // how many "g"s and "h"s, are there?
177    _n_coeffs = 0;
178    for (int n = 1; n <= _order; _n_coeffs += ++n);
179    // allocate space for the components
180    _br = new real[_order + 1];
181    _bt = new real[_order + 1];
182    _bp = new real[_order + 1];
183    // now build an easier way of referring to the arrays; we do this solely
184    // for speed later
185    int32 ** gp, ** hp;
186    heap_check(gp = new int32*[_order + 1]); // so the "gp"s go wrt 1
187    heap_check(hp = new int32*[_order + 1]);
188    int index = 0;
189    for (n = 1; n <= _order; n++)
190      { gp[n] = &g[index];
191       hp[n] = &h[index];
192       index += (n + 1);
193      }
194
195  // the following is incredibly unsubtle, but I can't think of a more
196  // elegant way to do it
197  if (_order > 0) then
198    { g10 = gp[1][0];
199     g11 = gp[1][1];
200     h10 = hp[1][0];
201     h11 = hp[1][1];
202    }
203
if (_order > 1) then
    { g20 = gp[2][0];
      g21 = gp[2][1];
      g22 = gp[2][2];
      h20 = hp[2][0];
      h21 = hp[2][1];
      h22 = hp[2][2];
    }

if (_order > 2) then
    { g30 = gp[3][0];
      g31 = gp[3][1];
      g32 = gp[3][2];
      g33 = gp[3][3];
      h30 = hp[3][0];
      h31 = hp[3][1];
      h32 = hp[3][2];
      h33 = hp[3][3];
    }

if (_order > 3) then
    { g40 = gp[4][0];
      g41 = gp[4][1];
      g42 = gp[4][2];
      g43 = gp[4][3];
      g44 = gp[4][4];
      h40 = hp[4][0];
      h41 = hp[4][1];
      h42 = hp[4][2];
      h43 = hp[4][3];
      h44 = hp[4][4];
    }

if (_order > 4) then
    { g50 = gp[5][0];
      g51 = gp[5][1];
      g52 = gp[5][2];
      g53 = gp[5][3];
      g54 = gp[5][4];
      g55 = gp[5][5];
      h50 = hp[5][0];
      h51 = hp[5][1];
      h52 = hp[5][2];
      h53 = hp[5][3];
      h54 = hp[5][4];
      h55 = hp[5][5];
    }

if (_order > 5) then
    { g60 = gp[6][0];
      g61 = gp[6][1];
      g62 = gp[6][2];
      g63 = gp[6][3];
      g64 = gp[6][4];
265 \[ g_{65} = g[6][5]; \]
266 \[ g_{66} = g[6][6]; \]
267 \[ h_{60} = h[6][0]; \]
268 \[ h_{61} = h[6][1]; \]
269 \[ h_{62} = h[6][2]; \]
270 \[ h_{63} = h[6][3]; \]
271 \[ h_{64} = h[6][4]; \]
272 \[ h_{65} = h[6][5]; \]
273 \[ h_{66} = h[6][6]; \]
274 \}
275
276 \text{if} \ (_{\text{order}} > 6) \text{then}
277 \{ \ g_{70} = g[7][0]; \}
278 \ g_{71} = g[7][1];
279 \ g_{72} = g[7][2];
280 \ g_{73} = g[7][3];
281 \ g_{74} = g[7][4];
282 \ g_{75} = g[7][5];
283 \ g_{76} = g[7][6];
284 \ g_{77} = g[7][7];
285 \ h_{70} = h[7][0];
286 \ h_{71} = h[7][1];
287 \ h_{72} = h[7][2];
288 \ h_{73} = h[7][3];
289 \ h_{74} = h[7][4];
290 \ h_{75} = h[7][5];
291 \ h_{76} = h[7][6];
292 \ h_{77} = h[7][7];
293 \}
294
295 \text{if} \ (_{\text{order}} > 7) \text{then}
296 \{ \ g_{80} = g[8][0]; \}
297 \ g_{81} = g[8][1];
298 \ g_{82} = g[8][2];
299 \ g_{83} = g[8][3];
300 \ g_{84} = g[8][4];
301 \ g_{85} = g[8][5];
302 \ g_{86} = g[8][6];
303 \ g_{87} = g[8][7];
304 \ g_{88} = g[8][8];
305 \ h_{80} = h[8][0];
306 \ h_{81} = h[8][1];
307 \ h_{82} = h[8][2];
308 \ h_{83} = h[8][3];
309 \ h_{84} = h[8][4];
310 \ h_{85} = h[8][5];
311 \ h_{86} = h[8][6];
312 \ h_{87} = h[8][7];
313 \ h_{88} = h[8][8];
314 \}
315
316 \text{destroy_array}(gp);
317 \text{destroy_array}(hp);
// destructor
magfield::~magfield(void)
{
    destroy_array(_br);
    destroy_array(_bt);
    destroy_array(_bp);
}

// change the order to which the field is calculated
void magfield::order(const int n)
{
    _order = MIN(n, _max_order);
}

// return the value of the field
cartesian magfield::field(const spherical& pos) const
{
    const real r = pos.r(),
    t = pos.t(),
    p = pos.p();

    // define some handy macros (very un-C++, this)
    #define GC_PLUS_HS(x, y) (g##x##y * cnp##y + h##x##y * snp##y)
    #define GS_MINUS_HC(x, y) (g##x##y * snp##y - h##x##y * cnp##y)

    // n = 1
    if (_order >= 1) then
    {
        const real cp = cos(p),
        sp = sin(p),
        cnpl = cp,
        snpl = sp,
        r3 = r * r * r,
        ct = cos(t),
        st = sin(t);
        const real gc_plus_hs_11 = GC_PLUS_HS(1,1);
        _br[1] = (2 / r3) * (ct * g10 + st * gc_plus_hs_11);
        _bt[1] = (g10 * st - ct * gc_plus_hs_11) / r3;
        _bp[1] = GS_MINUS_HC(1,1) / r3;
    }

    // n = 2
    if (_order >= 2) then
    {
        const real cnp2 = cos(2 * p),
        snp2 = sin(2 * p),
        cnt2 = cos(2 * t),
        snt2 = sin(2 * t),
        stn2 = st * st,
        r4 = r3 * r;
        const real gc_plus_hs_22 = GC_PLUS_HS(2,2),
        gc_plus_hs_21 = GC_PLUS_HS(2,1);
    }
\_br[2] = 3 * (1 + 3 * cnt2) * g20 / (4 * r4) + sqrt27 * (gc_plus_hs_22) * 
\_stn2 / (2 * r4) + sqrt27 * ct * (gc_plus_hs_21) * st / r4;

// bug required changing sign of _bt[2] and removal of factor of st 
// in the (2,1) term 
\_bt[2] = - ((sqrt3 * cnt2 * (1 / st) * (cp * g21 + h21 * sp) * st) / r3 - 3 
* g20 * 
snt2 / (2 * r3) + sqrt3 * (cnp2 * g22 + h22 * snp2) * snt2 / (2 * r3)) / r);

\_bp[2] = - ((1 / st) * (sqrt3 * (cnp2 * h22 - g22 * snp2) * st) / r3 + 
sqrt3 * ct 
* (cp * h21 - g21 * sp) * st / r3) / r);

if (_order >= 3) then 
{ const real r5 = r4 * r, 
cnp3 = cos(3 * p), 
snp3 = sin(3 * p), 
cnt3 = cos(3 * t), 
ctn2 = ct * ct, 
ctn3 = ctn2 * ct, 
\_sgtn3 = stn2 * st; 
const real V\_s4r5 = 4 * r5, 
Vsqrt8r5 = sqrt8 * r5; 
const real gc_plus_hs_33 = GC\_PLUS\_HS(3,3), 
gc_plus_hs_32 = GC\_PLUS\_HS(3,2), 
gc_plus_hs_31 = GC\_PLUS\_HS(3,1); 
\_br[3] = (3 * ct + 5 * cnt3) * g30 / (2 * r5) + 2 * sqrt15 * ct * (cnp2 * 
g32 + h32 
* snp2) * stn2 / r5 - sqrt6 * (1 - 5 * ctn2) * (cp * g31 + h31 * sp) * st 
+ sqrt15 * ct * (cnp3 * g33 + h33 * snp3) * stn3 / r5; 
}

// Simplify[b2[3]] 
_b\_t[3] = (3 * (3 + 5 * cnt2) * g30 / (4 * r4) - sqrt15 * (1 + 3 * cnt2) * 
\_sgnp2 * stn2 / r5 - sqrt6 * (1 - 5 * ctn2) * (cp * g31 + h31 * sp) * st 
+ sqrt10 * (cnp3 * g33 + h33 * snp3) * stn3 / r5; 
}

// BP[3] 
\_b\_p[3] = - ((1 / st) * (sqrt15 * ct * (cnp2 * h32 - g32 * snp2) * stn2 / r4 + 
sqrt3 * sqrt32 * (3 + 5 * cnt2) * (cp * h31 - g31 * sp) * st / r4 + 3 * 
sqrt5 / sqrt8 * (cnp3 * h33 - g33 * snp3) * stn3 / r4) / r);

// n = 4 
if (_order >= 4) then 
{ const real r6 = r5 * r, 
cnp4 = cos(4 * p),
\[ snp_4 = \sin(4 \cdot p), \]
\[ cnt_4 = \cos(4 \cdot t), \]
\[ ctn_3 = ctn_2 \cdot ct, \]
\[ ctn_4 = ctn_3 \cdot ct, \]
\[ stn_4 = stn_3 \cdot st; \]
\[ \]
\[ \text{const real } V_{\text{sqrt8r6}} = \sqrt{8} \cdot r_6, \]
\[ V_{4r6} = 4 \cdot r_6, \]
\[ V_{8r6} = 8 \cdot r_6, \]
\[ V_{st6} = st \cdot r_6; \]
\[ \]
\[ \text{const real } gc_{\text{plus hs}}_{44} = GC_{\text{PLUS HS}}(4,4), \]
\[ gc_{\text{plus hs}}_{43} = GC_{\text{PLUS HS}}(4,3), \]
\[ gc_{\text{plus hs}}_{42} = GC_{\text{PLUS HS}}(4,2), \]
\[ gc_{\text{plus hs}}_{41} = GC_{\text{PLUS HS}}(4,1); \]
\[ \]
\[ _{br}[4] = 5 \cdot (3 - 30 \cdot ctn_2 + 35 \cdot ctn_4) \cdot g_{40} / (8 \cdot r_6) + \sqrt{125} \cdot (5 + 7 \cdot \]
\[ ctn_2) \cdot (cn_{p2} \cdot g_{42} + h_{42} \cdot snp_2) \cdot stn_2 / (8 \cdot r_6) + \sqrt{875} \cdot (cn_{p4} \cdot \]
\[ g_{44} + h_{44} \cdot snp_4) \cdot stn_4 / (8 \cdot r_6) - \sqrt{125} / \sqrt{8} * (3 + ct - 7 \cdot \]
\[ ctn_3) \cdot (cp \cdot g_{41} + h_{41} \cdot sp) \cdot st / r_6 + \sqrt{875} / \sqrt{8} * ct + (cn_{p3} \cdot \]
\[ g_{43} + h_{43} \cdot snp_3) \cdot stn_3 / r_6; \]
\[ \]
\[ _{bt}[4] = (5 \cdot (9 \cdot ct + 7 \cdot ctn_3) \cdot g_{40} \cdot st / (8 \cdot r_5) - \sqrt{5} \cdot (5 + 7 \cdot \]
\[ ctn_3 \cdot (cn_{p2} \cdot g_{42} + h_{42} \cdot snp_2) \cdot st / (4 \cdot r_5) - \sqrt{35} \cdot ct \cdot (cn_{p4} \cdot \]
\[ g_{44} + h_{44} \cdot snp_4) \cdot stn_3 / (2 \cdot r_5) - \sqrt{35} / \sqrt{8} * ct + (cn_{p3} \cdot g_{43} + h_{43} \cdot snp_3) \cdot stn_3 / r_5 / r; \]
\[ \]
\[ _{bp}[4] = (1 / st) \cdot (cp \cdot g_{41} + h_{41} \cdot sp) \cdot st / r_5 - \sqrt{35} / \sqrt{8} * ct + (cn_{p3} \cdot g_{43} + h_{43} \cdot snp_3) \cdot \]
\[ stn_3 / r_5) / r; \]
\[ \]
\[ // n = 5 \]
\[ \]
\[ if (_\text{order} >= 5) \text{ then} \]
\[ \{ \text{const real } r_7 = r_6 \cdot r, \]
\[ ctn_5 = ctn_4 \cdot ct, \]
\[ stn_5 = stn_4 \cdot st, \]
\[ ctn_5 = \cos(5 \cdot t), \]
\[ ctn_5 = \cos(5 \cdot p), \]
\[ snp_5 = \sin(5 \cdot p); \]
\[ \]
\[ \text{const real } gc_{\text{plus hs}}_{55} = GC_{\text{PLUS HS}}(5,5), \]
\[ gc_{\text{plus hs}}_{54} = GC_{\text{PLUS HS}}(5,4), \]
\[ gc_{\text{plus hs}}_{53} = GC_{\text{PLUS HS}}(5,3), \]
\[ gc_{\text{plus hs}}_{52} = GC_{\text{PLUS HS}}(5,2), \]
\[ gc_{\text{plus hs}}_{51} = GC_{\text{PLUS HS}}(5,1); \]
\[ \]
\[ \text{const real } V_{2r7} = 2 \cdot r_7, \]
\[ V_{4r7} = 4 \cdot r_7, \]
\[ \text{Vsqrt32r7} = \sqrt{32} \cdot r7; \]
\[ \text{_br[5]} = 3 \cdot (15 \cdot ct - 70 \cdot \text{ctn3} + 63 \cdot \text{ctn5}) \cdot g50 / (4 \cdot r7) + \sqrt{945} \cdot (5 \cdot ct + 3 \cdot \text{ctn3}) \cdot (\text{cnp2} \cdot g52 + h52 \cdot \text{snp2}) \cdot \text{stn2} / (8 \cdot r7) + 9 \cdot \sqrt{35} \cdot \text{ct} \cdot (\text{cnp4} \cdot g54 + h54 \cdot \text{snp4}) \cdot \text{stn4} / (4 \cdot r7) - \sqrt{135} \cdot (1 + 14 \cdot \text{ctn2}) \cdot (\text{cnp3} \cdot g53 + h53 \cdot \text{snp3}) \cdot \text{stn2} \cdot \text{st} / r7 + 9 \cdot \sqrt{7} \cdot r7 \]
\[ \text{const real Vstr7} = \text{st} \cdot r7, \]
\[ \text{Vsqr128r7} = \sqrt{128} \cdot r7, \]
\[ V8r7 = 8 \cdot r7; \]
\[ \text{_bt[5]} = (15 \cdot (15 + 28 \cdot \text{cnt2} + 21 \cdot \text{ctn4}) \cdot g50 \cdot \text{st} / (64 \cdot r6) - \sqrt{105} \cdot (5 + 12 \cdot \text{cnt2} + 15 \cdot \text{ctn4}) \cdot (\text{cnp2} \cdot g52 + h52 \cdot \text{snp2}) \cdot \text{st} / (32 \cdot r6) - 3 \cdot \sqrt{35} \cdot \text{ct} \cdot (\text{cnp4} \cdot g54 + h54 \cdot \text{snp4}) \cdot \text{stn3} / (16 \cdot r6) - \sqrt{15} \cdot (1 + 9 \cdot \text{ctn2}) \cdot (\text{cnp3} \cdot g53 + h53 \cdot \text{snp3}) \cdot \text{stn5} / r6 - 3 \cdot \sqrt{35} \cdot \text{ct} \cdot (\text{cnp4} \cdot g54 + h54 \cdot \text{snp4}) \cdot \text{stn3} / (16 \cdot r6) - 3 \cdot \sqrt{35} \cdot (1 + 9 \cdot \text{ctn2}) \cdot (\text{cnp3} \cdot g53 + h53 \cdot \text{snp3}) \cdot \text{stn5} / r6 - 3 \cdot \sqrt{35} \cdot (1 + 9 \cdot \text{ctn2}) \cdot (\text{cnp3} \cdot g53 + h53 \cdot \text{snp3}) \cdot \text{stn5} / r6 - 3 \cdot \sqrt{35} \cdot (1 + 9 \cdot \text{ctn2}) \cdot (\text{cnp3} \cdot g53 + h53 \cdot \text{snp3}) \cdot \text{stn5} / r6) / r; \]
\[ \text{const real Vsqrt128r8} = \sqrt{128} \cdot r8, \]
\[ Vsqrt512r8 = \sqrt{512} \cdot r8, \]
\[ V8r8 = 8 \cdot r8, \]
\[ V16r8 = 16 \cdot r8, \]
\[ V32r8 = 32 \cdot r8, \]
\[ V_{128r8} = 128 \times r8, \]
\[ V_{str8} = st \times r8; \]
\[ \_br[6] = 7 \times (-5 + 105 \times ctn2 - 315 \times ctn4 + 231 \times ctn6) \times g60 \div (16 \times r8) + \]
\[ \sqrt{5145} \div \sqrt{512} \times (5 - 5 \times ct + 30 \times ctn3 - 33 \times ctn5) \times (cp \times g61 + \]
\[ h61 \times sp) \times st \div (8 \times r8) + \sqrt{5145} \times g62 \times (3 \times ct + 11 \times ctn3) \times \]
\[ (cnp3 \times g63 + h63 \times snp3) \times stn2 \times r8 + 3 \times \sqrt{3777} \div \sqrt{128} \times ct \times \]
\[ (cnp5 \times g65 + h65 \times snp5) \times stn5 \div r8; \]
\[ \_bt[6] = - (21 \times (50 \times ct + 45 \times ctn3 + 33 \times ctn5) \times g60 \div (128 \times \]
\[ r7) + \]
\[ \sqrt{105} \div \sqrt{128} \times (5 - 19 \times ctn2 + 51 \times ctn4 - 33 \times ctn6) \times (cnp2 \times g62 + \]
\[ h62 \times snp2) \times st \div (32 \times r7) + \sqrt{2079} \div \sqrt{128} \times ct \times (cnp6 \times g66 + \]
\[ h66 \times snp6) \times stn5 \div (8 \times r7) + \sqrt{2079} \div \sqrt{128} \times ct \times \]
\[ (cnp3 \times g63 + h63 \times snp3) \times stn4 \times (st + r7 + 3 \times \]
\[ \sqrt{2079} \div \sqrt{128} \times ct \times (cnp5 \times g65 + h65 \times snp5) \times stn5 \div r7) \times r); \]
\[ // n = 7 \]
\[ if (_order >= 7) then \]
\[ {\text{const real r9 = r8 \times r}, \]
\[ ctn7 = ctn6 \times ct, \]
\[ stn7 = stn6 \times st, \]
\[ cnt7 = \cos(7 \times t), \]
\[ cnp7 = \cos(7 \times p), \]
\[ snp7 = \sin(7 \times p); \]
\[ const real gc_plus_hs_77 = GC_PLUS_HS(7,7), \]
\[ gc_plus_hs_76 = GC_PLUS_HS(7,6), \]
\[ gc_plus_hs_75 = GC_PLUS_HS(7,5), \]
\[ gc_plus_hs_74 = GC_PLUS_HS(7,4), \]
\[ gc_plus_hs_73 = GC_PLUS_HS(7,3), \]
\[ gc_plus_hs_72 = GC_PLUS_HS(7,2), \]
\[ gc_plus_hs_71 = GC_PLUS_HS(7,1); \]
\[ const real V2r9 = 2 \times r9, \]
\[ V4r9 = 4 \times r9, \]
\[ V32r9 = 32 \times r9, \]
\[ V_{sqtr8r9} = sqrt8 \times r9, \]
\[ Vstr9 = st \times r9, \]
\[ \sqrt{512} \text{V} \sqrt{512} \text{V} \text{str9} = \sqrt{512} * \text{V} \text{str9}; \]

\[ \text{br}[7] = \left( -\frac{35 * \text{ct} + 315 * \text{ctn3} - 693 * \text{ctn5} + 429 * \text{ctn7}}{2 * \text{r9}} + \frac{\sqrt{21}}{\sqrt{512}} \right) \]

\[ \text{sn} \text{n6} / \text{r9} - \text{sqrt}7 \left( \frac{5 - 135 * \text{ctn2} + 495 * \text{ctn4} - 429 * \text{ctn6}}{2 * \text{r9}} \right) - \frac{\sqrt{21}}{\sqrt{512}} \left( \frac{3 - 66 * \text{ctn2} + 143 * \text{ctn4}}{2 * \text{r9}} \right) - \frac{\sqrt{231}}{\sqrt{8}} \left( \frac{13 * \text{ctn2} - (\text{ctn5} * \text{h75} + \text{h75} * \text{sn} \text{n5})}{\text{stn4}} \right) / (4 * \text{r9}) \]

\[ \text{bt}[7] = \left( -\frac{\sqrt{512} * \text{ct} * (\text{ctn5} * \text{g77} + \text{h77} * \text{sn} \text{n7})}{\text{stn7}} \right) / (4 * \text{r9}); \]

\[ \text{bp}[7] = -\left( (1 / \text{st}) * \left( -\left( \frac{\text{sqrt7} * \text{st} * (5 - 135 * \text{ctn2} + 495 * \text{ctn4} - 429 * \text{ctn6})}{2 * \text{r9}} \right) - \frac{\sqrt{231}}{\sqrt{8}} \left( \frac{13 * \text{ctn2} - (\text{ctn5} * \text{h75} + \text{h75} * \text{sn} \text{n5})}{\text{stn4}} \right) / (32 * \text{r8}) \right) \]

if \((\_\text{order} >= 8)\) then

\[ \text{n} = 8; \]

\[ \text{cn} \text{t9} = \text{ctn8} * \text{ct}, \]

\[ \text{stn8} = \text{stn7} * \text{st}, \]

\[ \text{ctn8} = \cos(6 * \text{t}), \]

\[ \text{cp} = \cos(6 * \text{p}), \]

\[ \text{sn} \text{p8} = \sin(6 * \text{p}); \]

46
const real V4rl0 = 4 • r10,
V32rl0 = 32 • r10,
Vsqrt2048rl0 = sqrt2048 • r10,
Vstrl0 = r10 • st,
V4strl0 = 4 • Vstrl0,
V8strl0 = 2 • V4strl0,
V16strl0 = 2 • V8strl0,
V32strl0 = 2 • V16strl0,
Vsqrt32strl0 = sqrt32 • Vstrl0,
Vsqrt128strl0 = sqrt128 • Vstrl0;

const real Vcpg81 = cp • g81,
Vcpn3g83 = cpn3 • g83,
Vcpn4g84 = cpn4 • g84,
Vcpn5g85 = cpn5 • g85,
Vcpn6g86 = cpn6 • g86,
Vcpn7g87 = cpn7 • g87,
Vcpn8g88 = cpn8 • g88,
Vaph81 = sp • h81,
Vapn2h82 = snp2 • h82,
Vapn3h83 = snp3 • h83,
Vapn4h84 = snp4 • h84,
Vapn5h85 = snp5 • h85,
Vapn6h86 = snp6 • h86,
Vapn7h87 = snp7 • h87,
Vapn8h88 = snp8 • h88;

const real V3sqrt?15 = 3 • sqrt?15;

gc_plus_hs_88 = GC_PLUS_HS(8,8),
gc_plus_hs_87 = GC_PLUS_HS(8,7),
gc_plus_hs_86 = GC_PLUS_HS(8,6),
gc_plus_hs_85 = GC_PLUS_HS(8,5),
gc_plus_hs_84 = GC_PLUS_HS(8,4),
gc_plus_hs_83 = GC_PLUS_HS(8,3),
gc_plus_hs_82 = GC_PLUS_HS(8,2),
gc_plus_hs_81 = GC_PLUS_HS(8,1);
683  _bt[8] = (9 * (1225 * ct + 1155 * cnt3 + 1001 * cnt5 + 715 * cnt7) * g80 * st ) /

684  (1024 * r9) - 3 * sqrt35 / sqrt131072 + (105 * ct + 121 * cnt3 + 143 * 
685  cnt5 + 143 * cnt7) * (cnp2 * g82 + h82 * snp2) * st / r9 - 3 * sqrt77 *  
686  (138 * ct + 117 * cnt3 + 65 * cnt5) * (cnp4 * g84 + h84 * snp4) * stn3 /  
687  (128 * r9) - sqrt3861 / sqrt512 * (9 * ct + 5 * cnt3) * (cnp6 * g86 + h86  
688  * snp6) * stn5 / r9 - 3 * sqrt715 * ct * (cnp8 * g88 + h88 * snp8) * stn7  
689  / (16 * r9) - 3 * (35 * cnt2 + 154 * cnt4 + 429 * cnt6 + 1430 * cnt8) * (1  
690  / st) * (cp * g81 + h81 * sp) / st / (1024 * r9) - sqrt10395 * (21 + 42 *  
691  cnt2 + 39 * cnt4 + 26 * cnt6) * (cnp3 * g83 + h83 * snp3) * st / st / (256  
692  * r9) - 3 * sqrt1001 * (11 + 19 * cnt2 + 10 * cnt4) * (cnp5 * g85 + h85 *  
693  cnt5) * stn3 * st / (64 * r9) - 3 * sqrt715 * (3 + 4 * cnt2) * (1 / st) *  
694  (cnp7 * g87 + h87 * snp7) * stn7 / (32 * r9) / r;  
695  _bp[8] = - ((1 / st) * (3 * sqrt35 / sqrt512 * (35 / 128 + cnt2 / 2 + 11 *  
696  cnt4) / 
697  32 - 143 * cnt8 / 128) * (cnp2 * h82 - g82 * snp2) / r9 + 3 * sqrt77 * (99  
698  + 156 * cnt2 + 65 * cnt4) * (cnp4 * h84 - g84 * snp4) * stn4 / (128 * r9)  
699  + sqrt3861 / sqrt2048 * (13 + 15 * cnt2) * (cnp6 * h86 - g86 * snp6) *  
700  stn6 / r9 + 3 * sqrt715 * (cnp8 * h88 - g88 * snp8) * stn8 / (16 * r9) - 3  
701  * (35 * ct - 385 * cnt3 + 1001 * cnt5 - 715 * cnt7) * (cp * h81 - g81 *  
702  sp) * st / (32 * r9) + sqrt10395 * (3 * ct - 26 * cnt3 + 39 * cnt5) *  
703  (cnp3 * h83 - g83 * snp3) * stn2 * st / (32 * r9) - 15 * sqrt1001 * (ct -  
704  5 * cnt3) * (cnp5 * h85 - g85 * snp5) * stn4 * st / (32 * r9) + 21 *  
705  sqrt715 * ct * (cnp7 * h87 - g87 * snp7) * stn7 / (32 * r9)) / r);  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  

48
real magfield::sqrt3 = sqrt(3);
real magfield::sqrt5 = sqrt(5);
real magfield::sqrt6 = sqrt(6);
real magfield::sqrt7 = sqrt(7);
real magfield::sqrt8 = sqrt(8);
real magfield::sqrt10 = sqrt(10);
real magfield::sqrt15 = sqrt(15);
real magfield::sqrt21 = sqrt(21);
real magfield::sqrt27 = sqrt(27);
real magfield::sqrt32 = sqrt(32);
real magfield::sqrt35 = sqrt(35);
real magfield::sqrt55 = sqrt(55);
real magfield::sqrt77 = sqrt(77);
real magfield::sqrt105 = sqrt(105);
real magfield::sqrt125 = sqrt(125);
real magfield::sqrt128 = sqrt(128);
real magfield::sqrt135 = sqrt(135);
real magfield::sqrt189 = sqrt(189);
real magfield::sqrt231 = sqrt(231);
real magfield::sqrt280 = sqrt(280);
real magfield::sqrt343 = sqrt(343);
real magfield::sqrt429 = sqrt(429);
real magfield::sqrt512 = sqrt(512);
real magfield::sqrt715 = sqrt(715);
real magfield::sqrt840 = sqrt(840);
real magfield::sqrt875 = sqrt(875);
real magfield::sqrt945 = sqrt(945);
real magfield::sqrt1001 = sqrt(1001);
real magfield::sqrt1029 = sqrt(1029);
real magfield::sqrt1120 = sqrt(1120);
real magfield::sqrt1155 = sqrt(1155);
real magfield::sqrt1512 = sqrt(1512);
real magfield::sqrt2048 = sqrt(2048);
real magfield::sqrt2079 = sqrt(2079);
real magfield::sqrt3003 = sqrt(3003);
real magfield::sqrt3773 = sqrt(3773);
real magfield::sqrt3861 = sqrt(3861);
real magfield::sqrt5145 = sqrt(5145);
real magfield::sqrt8192 = sqrt(8192);
real magfield::sqrt9317 = sqrt(9317);
real magfield::sqrt9856 = sqrt(9856);
real magfield::sqrt10395 = sqrt(10395);
real magfield::sqrt11319 = sqrt(11319);
real magfield::sqrt18711 = sqrt(18711);
real magfield::sqrt30240 = sqrt(30240);
real magfield::sqrt32768 = sqrt(32768);
real magfield::sqrt34749 = sqrt(34749);
real magfield::sqrt86515 = sqrt(86515);
real magfield::sqrt93555 = sqrt(93555);
real magfield::sqrt131072 = sqrt(131072);
real magfield::sqrt446875 = sqrt(446875);
real magfield::sqrt4084101 = sqrt(4084101);
real magfield::sqrt4204629 = sqrt(4204629.);
real magfield::sqrt5872581 = sqrt(5872581.);
real magfield::sqrt124540416 = sqrt(124540416.).
LISTING III
a) VECTOR.H

Header file for class to perform vector calculations

```c++
// C++ code Copyright (C) David R. Evans G4AMJ/NQOI

#ifndef VECTORH
#define VECTORH

#include <iostream.h>
#include <math.h>
#include <defines.h>

class cartesian;
class spherical;

enum AXES { X_AXIS = 0, Y_AXIS, Z_AXIS };

// -------------------------- cartesian class --------------------------
class cartesian
{
    double _len, _x, _y, _z;
    boolean length_guaranteed;

    public:
        cartesian(void);
        cartesian(const double, const double, const double);
        cartesian(const spherical&);
        cartesian(cartesian&);
        void operator=(cartesian);
        void operator=(spherical);
        int operator==(int n) { return (length(*this) == n); }
        int operator!=(int n) { return (length(*this) != n); }
        void operator*=(double);
        void operator/=(double);
        void operator+=(cartesian c) { *this = *this + c; }

        friend double length(cartesian &);
        friend double length(const cartesian&);

        inline void x(double d) { _x = d; length_guaranteed = false; }
        inline void y(double d) { _y = d; length_guaranteed = false; }
        inline void z(double d) { _z = d; length_guaranteed = false; }

        inline double x(void) const { return _x; }
        inline double y(void) const { return _y; }
        inline double z(void) const { return _z; }

        cartesian operator*(const cartesian&) const;
};
```

51
// cartesian + cartesian
inline cartesian operator+(const cartesian& rhs) const
{ return cartesian(_x + rhs._x, _y + rhs._y, _z + rhs._z); }

// cartesian - cartesian
inline cartesian operator-(const cartesian& rhs) const
{ return cartesian(_x - rhs._x, _y - rhs._y, _z - rhs._z); }

// -cartesian
inline cartesian operator-(void) const
{ return cartesian(-_x, -_y, -_z); }

// cartesian / double
inline cartesian operator/(const double f) const
{ return cartesian(_x / f, _y / f, _z / f); }

friend ostream& operator<<(ostream& os, const cartesian&); }

// -------------------------- common functions --------------------------
inline cartesian operator*(const double d, const cartesian& c)
{ return cartesian(d * c.x(), d * c.y(), d * c.z()); }
inline cartesian operator*(const cartesian& c, const double d)
{ return d * c; }
inline double x(const cartesian& c)
{ return c.x(); }
inline double y(const cartesian& c)
{ return c.y(); }
inline double z(const cartesian& c)
{ return c.z(); }

// _dot(cartesian, cartesian) -- the underline is necessary to
// avoid namespace collision with the dot class
inline double _dot(const cartesian& vl, const cartesian& v2)
{ return vl.x() * v2.x() + vl.y() * v2.y() + vl.z() * v2.z(); }

// cross(cartesian, cartesian)
inline cartesian cross(const cartesian& vl, const cartesian& v2)
{ return vl.x() * v2.y() - vl.y() * v2.x(); }

// unit(cartesian)
inline cartesian unit(const cartesian& c)
{ return (c / length(c)); }

// orthogonal(cartesian)
inline cartesian orthogonal(const cartesian& c)
{ return cartesian(y(c) - z(c), z(c) - x(c), x(c) - y(c)); }
cartesian rotate(const cartesian& c, const double deg, AXES = Z_AXIS);

// ------------------------ spherical class ------------------------

class spherical
{
    double _r, _theta, _phi;
	public:
        spherical(void);
        spherical(const double, const double, const double);
        spherical(const cartesian&);
        spherical(spherical&);
        void operator=(spherical);
        void operator=(cartesian);

        // spherical += spherical
        inline void operator+=(const spherical& s)
        { *this = *this + s; }

        // spherical *= double
        inline void operator*=(double d) { _r *= d; }

        // spherical /= double
        inline void operator/=(double d) { _r /= d; }

        inline double latitude(void) const
        { return (90 - t() * 180 / PI); }

        inline double longitude(void) const
        { return p() * 180 / PI; }

        inline double r(void) const { return _r; }
        inline double t(void) const { return _theta; }
        inline double p(void) const { return _phi; }

        inline spherical operator*(const spherical& s) const
        { return cross(*this, s); }

        inline spherical operator+(const spherical& s) const
        { return (const cartesian)(*this) + (const cartesian)s; }

        inline spherical operator-(const spherical& s) const
        { return (const cartesian)(*this) - (const cartesian)s; }

        inline spherical operator-(void)
        { return spherical(_r, PI - _theta, PI + _phi); }

        friend ostream& operator<<(ostream&, const spherical&);
// -------------------------- common functions --------------------------

inline double r(const spherical& s)
{ return s.r(); }

inline double t(const spherical& s)
{ return s.t(); }

inline double p(const spherical& s)
{ return s.p(); }

// unit(spherical)
inline spherical unit(const spherical& s)
{ return spherical(1.0, t(s), p(s)); }

// -------------------------- external functions --------------------------

extern istream& operator>>(istream&, spherical&);
extern istream& operator>>(istream&, cartesian&);
extern double angle(const cartesian&, const cartesian&);

#endif

#endif
LISTING III
b) VECTOR.CC
Class to perform vector calculations

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3 #include <vector.h>
4
5 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ cartesian
6
7 // generic constructor
8 cartesian::cartesian(void)
9 { _len = _x = _y = _z = 0;
10     length_guaranteed = true;
11 }
12
13 // constructor with (double, double, double)
14 cartesian::cartesian(const double tx, const double ty,
15                  const double tz)
16 { _x = tx;
17     _y = ty;
18     _z = tz;
19     _len = 0;
20     length_guaranteed = false;
21 }
22
23 // constructor with spherical
24 cartesian::cartesian(const spherical& s)
25 { _x = r(s) * sin(t(s)) * cos(p(s));
26     _y = r(s) * sin(t(s)) * sin(p(s));
27     _z = r(s) * cos(t(s));
28     _len = r(s);
29     length_guaranteed = true;
30 }
31
32 cartesian::cartesian(cartesian& c)
33 { _len = c._len;
34     _x = c._x;
35     _y = c._y;
36     _z = c._z;
37     length_guaranteed = c.length_guaranteed;
38 }
39
40 void cartesian::operator=(cartesian c)
41 { _len = c._len;
42     _x = c._x;
43     _y = c._y;
44     _z = c._z;
45     length_guaranteed = c.length_guaranteed;
46 }
47
48 // cartesian = spherical
49 void cartesian::operator=(spherical s)
50 { _x = r(s) * sin(t(s)) * cos(p(s));
51
52
53
54
55
\[ y = r(s) \cdot \sin(t(s)) \cdot \sin(p(s)); \]
\[ z = r(s) \cdot \cos(t(s)); \]
\[ \text{len} = r(s); \]
\[ \text{length}_{\text{guaranteed}} = \text{true}; \]

// cartesian *= double
void cartesian::operator*=(double d)
{ _len *= d;
  _x *= d;
  _y *= d;
  _z *= d;
}

// cartesian /= double
void cartesian::operator/=(double d)
{ _len /= d;
  _x /= d;
  _y /= d;
  _z /= d;
}

// length(const cartesian)
double length(const cartesian& cc)
{ if (cc.length_guaranteed) then
  return cc._len;
else
  return sqrt(cc._x * cc._x + cc._y * cc._y + cc._z * cc._z);
}

// length(cartesian)
double length(cartesian& param)
{ if (!(param.length_guaranteed)) then
  param._len = sqrt(param._x * param._x + param._y * param._y + param._z * param._z);
  param.length_guaranteed = true;
return param._len;
}

// cartesian * cartesian
cartesian cartesian::operator*(const cartesian& rhs) const
{ return cartesian(_y * rhs._z - rhs._y * _z,
  -(x * rhs._z - rhs._x * _z),
  _x * rhs._y - rhs._x * _y); }

56
104 // output the components of a cartesian
105 ostream& operator<<(ostream& st, const cartesian& v)
106 { st << "cartesian : x = " << x(v) << " y = " << y(v) << " z = " << z(v) << " length = " << length(v) << \n";
107    return st;
108 }
109
110 // input the components of a cartesian
111 istream& operator>>(istream& st, cartesian& v)
112 { double tx, ty, tz;
113    st >> tx >> ty >> tz;
114    v = cartesian(tx, ty, tz);
115    return st;
116 }
117
118 // angle(cartesian, cartesian)
119 double angle(const cartesian& c1, const cartesian& c2)
120 { double param = _dot(c1, c2) / (length(c1) * length(c2));
121    param = MIN(1.0, param);
122    param = MAX(-1.0, param);
123    return acos(param);
124 }
125
126 // rotate about an axis
127 cartesian rotate(const cartesian& c, const double deg, AXES axis)
128 { const double rad = deg * PI / 180,
129    cr = cos(rad),
130    sr = sin(rad);
131    switch (axis) { case X_AXIS : return cartesian(c.x(),
132                       c.y() * cr + c.z() * sr,
133                       -c.y() * sr + c.z() * cr);
134    case Y_AXIS : return cartesian(c.x() * cr - c.z() * sr,
135                       c.y(),
136                       c.x() * sr + c.y() * cr);
137    case Z_AXIS : return cartesian(c.x() * cr + c.y() * sr,
138                       -c.x() * sr + c.y() * cr,
139                       c.z());
140    }
141    }
142 }
143
144 // ********************************************** spherical
145
146 // generic spherical constructor
147 spherical::spherical(void) { _r = _theta = _phi = 0; }
148
149 // spherical constructor with (double, double, double) (angles in rad)
150 spherical::spherical(const double r, const double theta, const double phi)
151 { _r = r;
152    _theta = theta;
153    _phi = phi;
154 }
155
156 // IMPORTANT NOTE -- we do not constrain the values of the private
157 // data members while the object is being built. This is because
// we may simply be using spherical objects as a way to do
// spherical arithmetic (e.g. r, theta, phi may be component
// lengths of a vector). If data constraint is required then build
// the spherical object from a cartesian

// spherical constructor (cartesian)
spherical::spherical(const cartesian& c)
{ _r = length(c);
  _phi = (x(c) ? atan(y(c) / x(c)) : (y(c) > 0 ? PI / 2 : -PI / 2));
  if (x(c) < 0) then
    _phi += PI;
  while (_phi < 0) { _phi += 2 * PI; }
  _phi = fmod(_phi, 2 * PI);
  _theta = (z(c) ? atan(sqrt(_r * _r - z(c) * z(c)) / z(c)) : PI / 2);
  while (_theta < 0) { _theta += PI; }
  _theta = fmod(_theta, PI);
}

// ZTC-required copy-initialiser
spherical::spherical(spherical s)
{ _r = s._r;
  _theta = s._theta;
  _phi = s._phi;
}

// explicit equality operator
void spherical::operator=(spherical s)
{ _r = s._r;
  _theta = s._theta;
  _phi = s._phi;
}

// spherical = cartesian
void spherical::operator=(cartesian c)
{ _r = length(c);
  _phi = (x(c) ? atan(y(c) / x(c)) : (y(c) > 0 ? PI / 2 : -PI / 2));
  if (x(c) < 0) then
    _phi += PI;
  while (_phi < 0) { _phi += 2 * PI; }
  _phi = fmod(_phi, 2 * PI);
  _theta = (z(c) ? atan(sqrt(_r * _r - z(c) * z(c)) / z(c)) : PI / 2);
  while (_theta < 0) { _theta += PI; }
  _theta = fmod(_theta, PI);
}

// output the components of a spherical
ostream& operator<<(ostream& st, const spherical& v)
{ st << "spherical : r = " << r(v) << " t = " << t(v) * 180 / PI << " p = "
    << p(v) * 180 / PI << "\n";
  return st;
211 }
212
213 // input the components of a spherical
214 istream& operator>>(istream& st, spherical& v)
215 { double r, t, p;
216   st >> r >> t >> p;
217   t *= PI / 180;
218   p *= PI / 180;
219   v.r(r);
220   v.t(t);
221   v.p(p);
222   return st;
223 }
LISTING IV
WIREMODEL.CC

Program to produce the locus of magnetic field lines

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQ0I
2
3 // produce, on stdout, a wiremodel of magnetic field
4
5 #include <defines.h>
6 #include <magfield.h>
7 #include <vector.h>
8
9 #include <math.h>
10 #include <iostream.h>
11 #include <stdlib.h>
12 #include <sys/time.h>
13 #include <sys/resource.h>
14
15 // here define either URANUS or NEPTUNE
16 #undef URANUS
17 #define NEPTUNE
18
19 // if ORDER3 is defined, then uses 08 instead of full I8E1 44ev
20 #undef ORDER3
21
22 #ifdef URANUS
23 const float ellipticity = 0.0229;
24 #else
25 const float ellipticity = 0.017;
26 #endif
27
28 magfield* field;
29
30 // define matherr to abort
31 int matherr(struct exception *x)
32 { cout << "MATH ERROR\n";
33   exit(1);
34 }
35
36 // move a distance along a field line
37 cartesian along_field_line(const cartesian& p, const int direction,
38                             const double scale)
39 { const cartesian p1 = p + direction * scale * unit(field->field(p));
40   const cartesian pt = p1 - direction * scale * unit(field->field(pt1));
41   const cartesian delta_p = pt - p1;
42   return p1 - 0.5 * delta_p;
43 }
44
45 main(0)
46 {
47
48 // build the correct field model
49 #ifdef URANUS
50    const int order = 2,
51       rank = 2;
52
53 // how many "g"s and "h"s are there?
54    int n_coeffs = 0;
55    for (int n = 1; n <= order; n_coeffs += ++n);
56
57 int32 * g, * h;
58
59  heap_check(g = new int32 [n_coeffs]);
60  heap_check(h = new int32 [n_coeffs]);
61
62 // specify the model coefficients
63    h[0] = 0;  h[2] = 0;
64    g[0] = 11893;
65    g[1] = 11579;
66    h[1] = -15684;
67    g[2] = -56030;
68    g[3] = -12587;
69    g[4] = 196;
70    h[3] = 6116;
71    h[4] = 4759;
72
73  heap_check(field = new magfield(order, g, h));
74 #endif
75
77 #ifdef NEPTUNE
78    const int order = 8,
79       rank = 8;
80
81 // how many "g"s and "h"s are there?
82    int n_coeffs = 0;
83    for (int n = 1; n <= order; n_coeffs += ++n);
84
85 int32 * g, * h;
86
87  heap_check(g = new int32 [n_coeffs]);
88  heap_check(h = new int32 [n_coeffs]);
89
90 // specify the model coefficients
91    h[0] = 0;  h[2] = 0;  h[5] = 0;  h[9] = 0;
92    h[14] = 0;  h[20] = 0;  h[27] = 0;  h[35] = 0;
93
94 // n = 1
95    g[0] = 9732;
96    g[1] = 3220;    h[1] = -9889;
97
98 // n = 2
99    g[2] = 7448;
100   g[3] = 664;    h[3] = 11230;
102
103 // n = 3
104 g[5] = -6592;
107 g[8] = 484;   h[8] = -770;
108
109 // n = 4
110 g[9] = 2243;
111 g[10] = 557;   h[10] = -1889;
113 g[12] = -1287;  h[12] = 1204;
115
116 // n = 5
117 g[14] = -202;
119 g[16] = 526;   h[16] = -1134;
120 g[17] = -2846;  h[17] = 1067;
121
122 // n = 6
123 g[18] = -1425;  h[18] = -1551;
124 g[19] = -2835;  h[19] = -1090;
125
126 // n = 7
127 g[20] = -2175;
128 g[21] = -466;   h[21] = 4432;
129 g[22] = -1269;  h[22] = -1598;
130 g[23] = -2233;  h[23] = 1721;
131 g[24] = -887;   h[24] = 370;
133 g[26] = 755;    h[26] = 1439;
134
135 // n = 8
136 g[27] = 1671;
137 g[28] = 1678;   h[28] = -3159;
138 g[29] = 1625;  h[29] = 1862;
139 g[30] = 2157;  h[30] = -1120;
140 g[31] = -483;  h[31] = 515;
141 g[32] = 1873;  h[32] = 1923;
142 g[33] = 584;   h[33] = -2749;
143 g[34] = 664;   h[34] = 3344;
144
145 // n = 9
146 g[35] = -689;
147 g[36] = 238;    h[36] = 1446;
148 g[37] = -90;    h[37] = -79;
149 g[38] = -1304;  h[38] = 1043;
150 g[39] = 311;   h[39] = -22;
151 g[40] = -367;  h[40] = -465;
152 g[41] = -249;  h[41] = 1043;
153 g[42] = 1333;  h[42] = -2138;
154 g[43] = -1239;  h[43] = 2519;
heap_check(field = new magfield(order, g, h));

// truncate if 08

#ifdef ORDER3
    field->order(3);
#endif
#endif

int modulo;

double dlat = 160;

for (double dlong = 0; dlong < 360; dlong += 4)
{
    spherical pos(1, dlat * PI / 180, dlong * PI / 180);
    // move so that measurements are (approximately) wrt the dipole model
    #ifdef URANUS
        pos = rotate(pos, 45.8, Y_AXIS);  // 105.2 | 45.8 (day | night)
        pos = rotate(pos, 227.5, Z_AXIS);  // 47.7 | 227.5
    #endif
    #ifdef NEPTUNE
        pos = rotate(pos, 138, Y_AXIS);  // 138 | 55
        pos = rotate(pos, 57, Z_AXIS);  // 57 | 278
    #endif

    cerr << dlat << " " << dlong << "\n" << pos;
    while ((r(pos) > 0.999) && (r(pos) < 20))
    {
        cout << "0 " << r(pos) << " " << t(pos) * 180 / PI << " " << p(pos) * 180 / PI << "\n";
        // for uranus, +1 goes with first pair
        pos = along_field_line(pos, +1, 0.01);  // +1 | -1
    }
}
LISTING V
PHOTO3.CC

Program to produce plot wire models

// C++ code Copyright (C) David R. Evans G4AMJ/NQOI

// takes the output of the file wiremodel on stdin and produces a
// picture of the wire model on the laserprinter.

// The output format of wiremodel is the same as quadnn.

#include <DREstring.h>
#include <gf_lw.h>
#include <surface.h>
#include <surf_lw.h>
#include <vector.h>
#include <stdio.h>
#include <math.h>
#include <signal.h>
#include <ctype.h>
#include <iostream.h>

// define matherr to abort
int matherr(struct exception *x)
{
    cout << "MATH ERROR\n";
    exit(1);
    return 1;
}

cartesian obs(100, 100, 100), OX, XO, OI(O, O, I), perp_OXOI;
double zoom = 1.0;
DREstring file_name;
laserwriter* pad;
spherical screen_posn(spherical);
void plot_proc(void),
    posn_proc_l(void);
laserproc(void)
{
    laserwriter lw;
    pad = &lw;
    plot_proc();
    // lw.print("lw");
    lw.detach(cout);
}

void laserproc(void)
{
    laserwriter lw;
    pad = &lw;
    plot_proc();
    // lw.print("lw");
    lw.detach(cout);
}
void plot_proc(void)
{
    pad->clear();
    spherical obs(s);
    pad << text(pad->width() / 10, pad->height() / 10,
            pad->height() / 30,
            DREstring(t(s) * 180 / PI, 3.2) + "", " +
            DREstring(p(s) * 180 / PI, 3.2));

    /* generate the "planet" */
    int planet_radius = (int)(pad->height() / 2 * zoom *
            asin(1 / length(s)) / PI);
    int xc = pad->width() / 2;
    int yc = pad->height() / 2;

    // we force the radius to be 100
    zoom = 100 / (pad->height() / 2 * zoom * asin(1 / length(s)) / PI);
    planet_radius = (int)(pad->height() / 2 * zoom *
            asin(1 / length(s)) / PI); // should be 100

    *pad << Circle(xc, yc, planet_radius);

    // define coordinate system
    // origin = 0 = centre of planet
    // X = observer
    // S = point to be plotted
    // I = (0, 0, 1) [defined to be vertically up on the screen]
    // L = point st angle ILS is the angle between planes OXI and OXS

    OX = obs;
    XO = -OX;
    perp_OXOI = cross(OX, OI);

    for (int latitude = -90; latitude <= 90; latitude += 30)
    { for (int longitude = 0; longitude <= 359; longitude++)
        { if (abs(latitude) == 90) then longitude = 359;
            spherical ll = screen_posn(spherical(1.001, (90 - latitude) * PI / 180,
                                             longitude * PI / 180));
            if (ll.r()) then
              *pad << dot(xc + (int)(r(ll) * cos(t(ll))),
                      yc - (int)(r(ll) * sin(t(ll))));
        }
    }

    spherical source;
    double alpha;
    do
    { cin >> alpha >> source;
        spherical screen = screen_posn(source);
        double distance = length(obs - source) - length(obs);
        int dot_size = (int)((2 - distance) * 2 + 1.5);
        dot_size = MIN(dot_size, 3);
        dot_size = MAX(dot_size, 1);
        if (screen.r()) then
          *pad << dot(xc + (int)(r(screen) * cos(t(screen))),
                      yc - (int)(r(screen) * sin(t(screen))));
    }
while (!cin.eof());

void print_proc(void)
{
    pad->print("lw");
    //pad->print(cout);
}

spherical screen_posn(spherical s)
{
    cartesian OS = s,
    XS = XO + OS;

    // point on screen is at R, T wrt centre of screen
    const double R = pad->height() / 2 * zoom * angle(XO, XS) / PI,
    plane_angle = angle(perp_OXOI, crossOX, OS));

    // which side of the central meridian?
    const int side = SIGN(_dot(OS, perp_OXOI));
    double T = PI / 2 + side * plane_angle;
    while (T < 0) T += 2 * PI;
    T = fmod(T, 2 * PI);

    boolean hidden = (length(OS) <= 1);
    double angDXO = angle(XS, XO);
    double param = lengthDX) * sin(angDXO);
    param = MIN(param, 1.0);
    param = MAX(-1.0, param);
    double angODX = asin(param);
    if (angDXO < PI / 2) then
        angODX = PI - angODX;
    double angDOX = PI - angODX - angDXO;
    double lenXD = sin(angDOX)/sin(angDXO);

    hidden = hidden || ((angle(XO, XS) < asin(1 / lengthDXO)) &&
    ((length(XS) > lenXD)));)
    if (hidden) then
        return spherical(0, 0, 0);
    return spherical(R, T, 0);
}

main(int argc, char** argv)
{
    if (argc != 3) then
        { cerr << "Usage: " << argv[0] << " theta phi (degrees) < filename.
";
            exit(-1);
        }
    float T, P;
    sscanf(argv[1], "%f", &T);
    sscanf(argv[2], "%f", &P);
obs = spherical(100, T * PI / 180, P * PI / 180);
laser_proc();
REFERENCES


**Abstract**

The magnetic field geometries of Uranus and Neptune are examined in detail. Computer code to track low energy charged particles has been generated. Running the code against the uranium A3 magnetic field model indicates that emissions previously thought to be an exotic ordinary mode emission can be more simply explained as extraordinary mode emissions from an isolated region of the inner dayside uranium magnetosphere.

A similar project to examine possible particle trapping in the neptunian magnetosphere founnd because of the peculiar nature of the neptunian magnetosphere.