UNIVERSITY of CALIFORNIA
SANTA CRUZ

RANDOM ANGLE GENERATION: MODELING DIFFUSE LIGHT REFLECTION IN PLANETARY RINGS

A thesis submitted in partial satisfaction of the
requirements for the degree of

BACHELOR OF SCIENCE

in

ASTROPHYSICS

by

Demitri A. Morgan

10 June 2009

The thesis of Demitri A. Morgan is approved by:

Jeffrey N. Cuzzi

Professor Jeffrey N. Cuzzi
Technical Advisor

Professor David P. Belanger
Thesis Advisor

Professor David P. Belanger
Chair, Department of Physics

Abstract

Random angle generation: modeling diffuse light reflection in planetary rings

by

Demitri A. Morgan

This work addresses the topic of generating random angles describing the scattered directions of simulated photon packets within Monte Carlo ray-tracing simulations of planetary rings, such as that of Salo and Karjalainen (2003), using an arbitrary reflectance law as the probability distribution and not merely one describing a Lambert surface. Also, a computational method which accomplishes this task, written in the *Interactive Data Language*™ , is developed and tested, using as the reflectance law an experimental approximation (proposed by Cuzzi, 2008) to those derived in Hapke (1993) describing photometric properties of surfaces having "large-scale roughness". The method generates the random angles with an implementation of the rejection sampling algorithm and uses polynomial approximations of the reflectance law as comparison and squeeze functions, in addition to running performance tests and a searching for approximations that yield higher performance.

Contents

List of Figures

1 Introduction

In early photometric modeling of planetary rings, the rings have been treated as semi-infinite, homogeneous multilayers of particles, with each ring particle having a radius much less than the average distance to its nearest neighbor (negligible or zero volume filling factor). The more recent work of Salo and Karjalainen (2003) has taken the approach of removing the assumption of zero volume filling factor and using direct and indirect ray tracing Monte Carlo method to model the scattering properties of the rings. The method treated a region within a ringlet as a rectangular volume having horizontally periodic boundaries and spherical ring particles much larger than the wavelength of visible light. By introducing simulated quanta of light ("rays") into the region which scatter into random directions at every intersection with the surface of a particle, and recording the direction and "weight"[1] of the escaping rays, the method simulates the directional distribution of radiation reflected by Saturn's rings. At each scattering, the random direction that a ray reflects into is distributed according to Lambert's reflectance law, which is uniform with respect to the azimuthal angle (denoted ϕ in Salo and Karjalainen (2003) and ψ in Hapke (1993)) and utilizes the mathematical simplicity of this model in the process of randomly generating the direction of the reflected ray. This work will address the use of arbitrary reflectance laws in order to generalize the concept of photometric modeling by a Monte Carlo ray tracing method.

2 Geometric description of the problem

The following conditions were imposed in the Monte Carlo method of Salo and Karjalainen (2003) and will serve as the context of this work: denoting the local surface normal, incident ray vector and emitted ray vector as $\hat{n}$, $\hat{e}_0$ and $\hat{e}$, respectively,

- Each ring particle is taken to have a regular and opaque surface; every time a ray intersects the surface of an object, the scattered photon packet must not emerge from the inward side of the local tangent plane, which can be likened to "below the local horizon" of the object. Formally, $\hat{e}_0 \cdot \hat{n} < 0$ and $\hat{e} \cdot \hat{n} > 0$;

- The random direction that a simulated photon packet scatters into has a probability distribution that is proportional to the reflection law.

- Ring particles in a simulated ringlet are similar in their composition, so that the reflection law and surface albedo may be assumed to apply universally throughout the region.

[1] To lessen the intensity of multiply scattered radiation, each simulated photon packet has an initial "weight" that is reduced at each scattering.

- Surfaces of the ring particles do not contain any large-scale anisotropy, i.e. grain. Formally, for any three angles $\pi - i$, e and g (see Section 2.1) that the three vectors $\hat{n}$, $\hat{e}_0$ and $\hat{e}$ make with each other, the reflection law function evaluated at them must be equal to the reflection law function evaluated at the three angles $\pi - i'$, e' and g' that three other vectors $F(\hat{n})$, $F(\hat{e}_0)$, and $F(\hat{e})$ make with each other if $i' = i$, $e' = e$ and $g' = g$ (F is in this case any orientation-preserving isometry of $\Re^3$).

2.1 Notation and terminology

At each interaction of a simulated photon packet ("*ray*") with the surface of a simulated ring particle, referred to as an *instance of reflection* or a *scattering event*, the incident ray initially travels in the direction $\hat{e}_0$ before the intersection. The vector pointing in the direction that the ray came from (denoted $\hat{\imath} \equiv -\hat{e}_0$) makes an angle i with respect to the surface's local normal vector, denoted $\hat{n}$. After computing the value of the parameter $\cos i$ (denoted μ_0), which is the dot product $\hat{\imath} \cdot \hat{n}$, the Monte Carlo method then passes this value to the procedure that randomly generates the vector of the scattered ray's direction, denoted $\hat{e}$. This vector is described by the two returned values, $\cos e$ (denoted μ) and ψ through an orthonormal basis (Section 2.2), and these values are constrained to the intervals $[0, 1]$ and $[0, 2\pi]$ respectively; e is the angle between $\hat{e}$ and $\hat{n}$, so e, like i, must always lie within the interval $[0, \frac{\pi}{2}]$ as a direct result of the assumption that the ring particles are opaque. The variable ψ is the angle between the projections of $\hat{\imath}$ and $\hat{e}$ to the local tangent plane, as shown in Fig. 2.1. The sub-routine (a.k.a. the method) for generating random values of

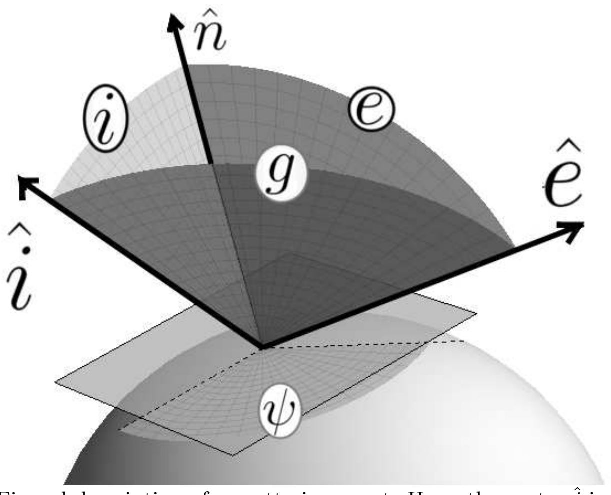


Figure 2.1: Figural description of a scattering event. Here, the vector $\hat{\imath}$ is shown in place of $\hat{e}_0$.

μ and ψ will require a reflectance function, denoted P, to be assigned to it for use as the main reflectance law. The method will assume that P only takes as its arguments angles that describe the directions of the incident and emitted ray, which may include i, e, the angle between $\hat{\imath}$ and $\hat{e}$ (the *phase angle*, denoted g), or the luminance coordinates L and Λ (see Fig. 2.3). In this project, the angles i, e and ψ will be used; they are not dependent on one another and are well-defined for all orientations of $\hat{n}$, $\hat{\imath}$, $\hat{e}$. Methods for transforming between the values of μ_0, μ, ψ and values of g, L, Λ are discussed in Section 2.3, and the reason for sampling μ instead of e is given in Section 2.4.

2.2 Coordinate basis

The task of generating random angles describing the scattered direction is made much simpler if the angles are independent of one another, i.e. e and ψ as opposed to e and g (for example); in the latter case, not only does the value of e affect the value of g and vice versa, but for any given i, e and g there are at least two possible directions that these angles could describe, as shown by Fig. 2.2. Using $\cos e$ and ψ, not only is it easier to fully express all directions, but the acceptable range of e can be directly imposed because the sample space remains rectangular (having constant boundaries and linear independence between all variables). While not all reflectance laws are explicit functions of i, e and ψ, methods for re-expressing such functions in terms of i, e, ψ (see Section 2.3) may be used to accommodate them. Generating a random vector

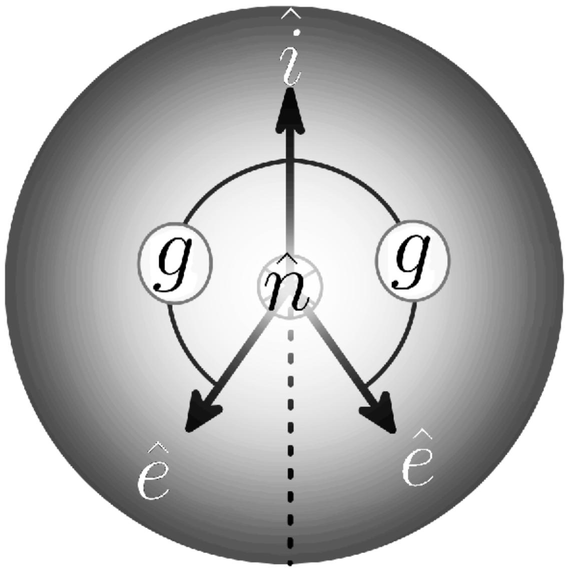


Figure 2.2: For any given i, e, and g, there are at least two distinct possible $\hat{e} \in H$ that the three angles could correspond to, given $\hat{n}$ and $\hat{\imath}$.

representing the scattered ray requires the construction of the vector using the values μ and ψ returned by it and the components of the vectors $\hat{e}_0$ and $\hat{n}$. This can be done by first constructing a local orthonormal vector basis using $\hat{i}$ and $\hat{n}$;

$$\begin{aligned} \hat{x}'_1 &\equiv \frac{\hat{i} - \hat{n}\cos i}{\sin i} && (2.1a) \\ \hat{x}'_2 &\equiv \frac{\hat{n} \times \hat{i}}{\sin i} && (2.1b) \\ \hat{x}'_3 &\equiv \hat{n} && (2.1c) \end{aligned}$$

This basis can then be used to give an adequate description of vectors within the hemispherical set of directions into which rays may reflect, denoted H. The emitted ray's direction in the basis $(\hat{x}'_1, \hat{x}'_2, \hat{x}'_3)$ is then:

$$\hat{e} = (\hat{x}'_1 \cos\psi + \hat{x}'_2 \sin\psi)\sin e + \hat{x}'_3 \cos e \qquad (2.2)$$

Or, using exclusively the returned values:

$$\hat{e} = (\hat{x}'_1 \cos\psi + \hat{x}'_2 \sin\psi)\sqrt{1-\mu^2} + \hat{x}'_3 \mu$$

Using the expression of $\hat{i}$ and $\hat{n}$ in the coordinate basis of the simulation $(\hat{x}_1, \hat{x}_2, \hat{x}_3)$ will then allow Definitions 2.1 and hence also the exit ray to be expressed in the same basis;

$$\begin{aligned} \hat{i} &= -e_1\hat{x}_1 - e_2\hat{x}_2 - e_3\hat{x}_3 \quad (e_i \text{ denotes component } i \text{ of } \hat{e}_0) \\ \hat{n} &= = n_1\hat{x}_1 + n_2\hat{x}_2 + n_3\hat{x}_3 \\ \therefore\ \hat{x}'_1 &= \frac{1}{\sin i}\left[(-e_1 - n_1\cos i)\hat{x}_1 + (-e_2 - n_2\cos i)\hat{x}_2 + (-e_3 - n_3\cos i)\hat{x}_3\right]\ , \\ \hat{x}'_2 &= \frac{1}{\sin i}\left[(n_3e_2 - n_2e_3)\hat{x}_1 + (n_1e_3 - n_3e_1)\hat{x}_2 + (n_2e_1 - n_1e_2)\hat{x}_3\right] \\ \hat{x}'_3 &= n_1\hat{x}_1 + n_2\hat{x}_2 + n_3\hat{x}_3 \end{aligned}$$

Hence, the emitted ray expressed in the basis of the simulation is:

$$\begin{aligned} \hat{e} &= \left(\frac{\sqrt{1-\mu^2}}{\sin i}\left[(n_3e_2 - n_2e_3)\sin\psi - (e_1 + n_1\cos i)\cos\psi\right] + \mu n_1\right)\hat{x}_1 \\ &+ \left(\frac{\sqrt{1-\mu^2}}{\sin i}\left[(n_1e_3 - n_3e_1)\sin\psi - (e_2 + n_2\cos i)\cos\psi\right] + \mu n_2\right)\hat{x}_2 \\ &+ \left(\frac{\sqrt{1-\mu^2}}{\sin i}\left[(n_2e_1 - n_1e_2)\sin\psi - (e_3 + n_3\cos i)\cos\psi\right] + \mu n_3\right)\hat{x}_3 \end{aligned}$$

2.3 The phase angle and luminance coordinates

Some models of diffuse reflection, such as the brightness distribution of Akimov (1979), traditionally describe reflectance on the surface of a spherical body and are functions of the luminance coordinates and/or the phase angle. In order to have the option of using such functions as probability distributions, the phase angle and luminance coordinates must be determined automatically given values of i, e, ψ, as described in Section 2.3.2, so that the value of the reflectance function for any value of i, e, ψ (or μ_0, μ, ψ) can be calculated. While it may be possible to randomly generate the reflection solely in terms of these angles, it is far easier to use i, e (or μ_0, μ) and ψ to do so, as stated in Section 2.2, and hence doing so will not be addressed in this work.

2.3.1 Definitions

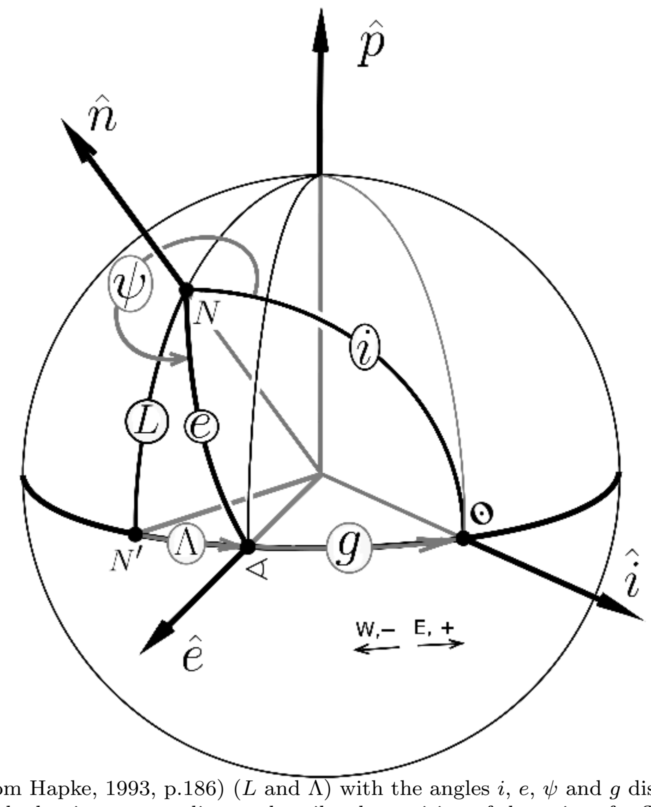


Figure 2.3: (derived from Hapke, 1993, p.186) (L and Λ) with the angles i, e, ψ and g displayed in context. Light reflects off of a spherical body, and the luminance coordinates describe the position of the point of reflection on its surface (N) relative to the plane spanned by $\hat{\imath}$ and $\hat{e}$.

The phase angle, denoted g, and the luminance longitude and latitude, Λ and L (respectively) are as shown in Fig. 2.3. The following formal definitions, based on similar definitions and notation from Hapke (1993), will be used for these angles and the geometric elements on which their definitions are based:

- The *photometric plane* is the plane spanned by $\hat{\imath}$ and $\hat{e}$ whose intersection with the surface of the sphere of solid angle is referred to as the *luminance equator*.

- The *principal plane* is the common plane that $\hat{\imath}$, $\hat{n}$, and $\hat{e}$ lie in if $\psi = 0, 180°$, etc.

- The *prime luminance meridian* is the meridian in the direction of the observer (or, more generally, the emitted ray), which runs through $\sphericalangle$.

- The *phase*, either positive or negative, is determined by which side of the prime luminance meridian the light source (☉) is on, and the *phase angle* is the shortest angle from $\sphericalangle$ *to* ☉ along the luminance equator.

- An angle along the luminance equator is negative if it runs clockwise ("West"), and positive if it runs counterclockwise ("East"), where these directions are shown in the perspectives of Fig. 2.3 and Fig. 2.4.

- The *luminance latitude* (or *photometric latitude*), L, is the angle $\angle NN'$ (the angle between the vector $\hat{n}$ and its projection to the photometric plane).

- The *luminance longitude* (or *photometric longitude*), Λ, is the angle along the luminance equator running from N' to the prime luminance meridian.

2.3.2 Transforming to phase angle and luminance coordinates

The phase angle and luminance coordinates are, as derived in Shkuratov *et al.* (2003) from Hapke (1993, p.187),

$$g(i,e,\psi) = \cos^{-1}\left(\cos\psi\sin i\sin e + \cos i\cos e\right) \tag{2.3a}$$

$$L(i,e,\psi) = \cos^{-1}\left(\sqrt{\frac{\left[\sin\left(i+e\right)\right]^2 - (\cos\psi/2)^2\sin 2e\sin 2i}{\left[\sin\left(i+e\right)\right]^2 - (\cos\psi/2)^2\sin 2e\sin 2i + (\sin e)^2(\sin i)^2(\sin\psi)^2}}\right) \tag{2.3b}$$

$$\Lambda(i,e,\psi) = \cos^{-1}\left(\frac{\cos e}{\cos L(i,e,\psi)}\right) \tag{2.3c}$$

Using these, the reflection law may be expressed entirely in terms of i, e and ϕ; if for example the function P it takes as its arguments g, L, Λ, it may be composed with these three functions as $P(i,e,\psi) = P\left[g(i,e,\psi), L(i,e,\psi), \Lambda(i,e,\psi)\right]$. Although Equations 2.3b and 2.3c may suffice for most values of i, e, ψ, they will result in indeterminate expressions for the anomalous cases $\psi = 0$ and $i = e = 0$. In order to circumvent not-a-number errors, L can be assigned a value of 0 if $\psi = 0$ and/or $i = e = 0$; this is the result of both the limiting case $\psi \to 0$ and $i = e \to 0$. However, in spite of this safeguard, there is an inescapable irregularity in L and Λ that results at zero phase angle regardless of how the luminance coordinates are set up for this case. This is because when $g = 0$, the photometric plane is ill-defined; if two vectors are parallel, then there is no unique plane that is parallel to both of them, and if $g = 0$ then $\hat{e}$ is parallel to $\hat{\imath}$. More specifically, the irregularity is that in the limiting case $i = e \neq 0$ and $\psi \to 0$, L should equal i; the equality of i and e results in a symmetry which causes the meridian running through N to be exactly halfway between $\odot$ and $\triangleleft$. This results in $\Lambda = g/2$ and, using the spherical law of cosines on $\triangle \odot N'N$ gives the expression $\cos i = \cos L \cos \frac{g}{2}$ and thus $L = \cos^{-1}(\cos i / \cos \frac{g}{2})$. This translates to $L = i$ for $\psi \to 0$, and hence the value of L (and by extension, Λ) in the limiting case $g \to 0$ cannot be assigned a unique value. The consequence of this is that a reflection law depending upon the luminance coordinates may possibly also be irregular around zero phase angle.

2.3.3 Orientation

If the quantity $g + \Lambda$ is to be used in any circumstance, it is necessary to include the sign of the angle Λ in the transformation between i, e, ψ and g, L, Λ; the quantity, defined as the angle subtended by the arc running from $\odot$ to N' (as in Hapke, 1993, Eq. 8.5), may not be accurately represented by $|g| + |\Lambda|$. This is because, for any given values of i and L, there are two distinct values of $g + \Lambda$, as illustrated by 4(b) and 4(c): one in which $|g + \Lambda| > |g|$, and another in which $|g + \Lambda| < |g|$. Hence, even if the sign of g may be ignored, the sign of Λ relative to that of g remains important if the value of $g + \Lambda$ is ever needed as a parameter of a reflection law, and so, for the sake of completeness, a closed-form expression for the sign of Λ will be derived here in order to retain the option of automatically determining the sign of Λ for any given i, e and ψ. The task may

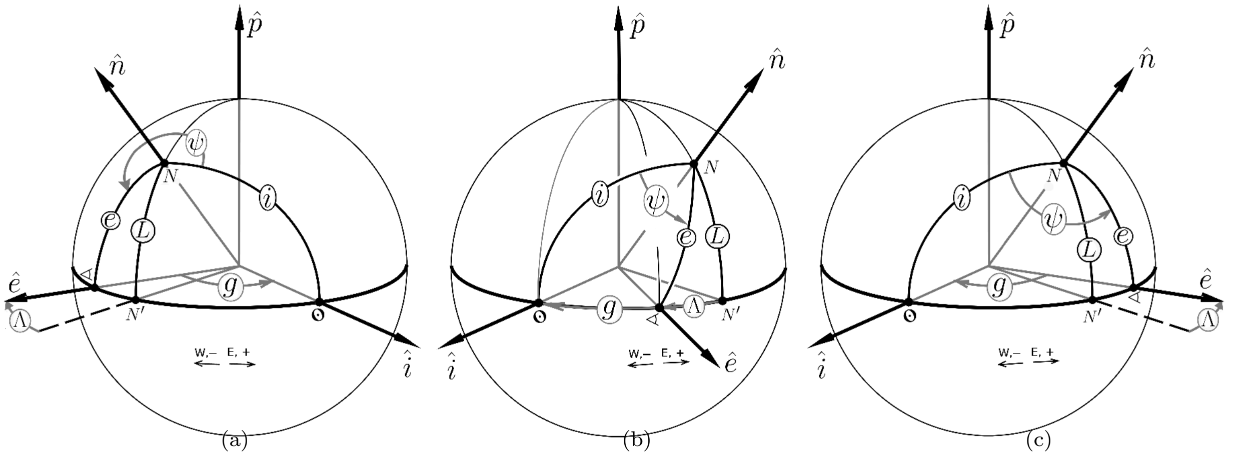


Figure 2.4: Alternate orientations of the vectors $\hat{\imath}$, $\hat{n}$, and $\hat{e}$, and the corresponding signs of these angles; (a) with positive phase and negative longitude, (b) with negative phase and negative longitude (Hapke, 1993, p.186), (c) with negative phase and positive longitude.

be accomplished by using the definitions of positive and negative angles along the luminance equator given in Section 2.3.1. However, by those definitions, the sign of the phase may be either positive or negative, so the sign of Λ *relative* to the phase is of greatest importance. To achieve this first requires determining East and West along the luminance equator, and in this there is need for a third point of reference with which to add perspective. This point must be N; the vectors $\hat{\imath}$ and $\hat{e}$ do not adequately describe the *orientation* of the photometric plane, but only the *luminance axis* (which runs perpendicular to it). The photometric plane's positive normal, $\hat{p}$, may then be defined as follows: let it be parallel (or anti-parallel) to $\hat{e} \times \hat{\imath}$, and let it be such that $\hat{p} \cdot \hat{n} > 0$, or in other terms, let N always be "North" of the luminance equator, and the quantity L always positive. With these assumptions in place, the positive normal $\hat{p}$ (the "North luminance pole") can be explicitly written as follows:

$$\hat{p} = sign\left[\hat{n} \cdot \left(\hat{e} \times \hat{\imath}\right)\right] \frac{\hat{e} \times \hat{\imath}}{|\sin g|} \tag{2.4}$$

To first obtain the sign of g, one can begin by considering how $\lhd$ being East of $\odot$ will result in the vector $\hat{e} \times \hat{\imath}$ pointing "North", and how the opposite arrangement will result in it pointing "South". It then becomes clear that the sign of $\sin g$ (and hence also the sign of g)[2] is related to the orientation of the three vectors $\hat{\imath}$, $\hat{e}$ and $\hat{n}$ as follows:

$$sign[\sin g] = sign\left[\hat{p} \cdot \left(\hat{e} \times \hat{\imath}\right)\right] \tag{2.5}$$

Then, using Equation 2.2 and Equation 2.4 along with definitions 2.1, this value can be examined and simplified by expressing each part of it in terms of $\hat{\imath}$ and $\hat{n}$; first, the quantity $\hat{n} \cdot \left(\hat{e} \times \hat{\imath}\right)$ can be reduced to an expression containing familiar angles;

$$\begin{aligned}
\hat{n} \cdot \left(\hat{e} \times \hat{\imath}\right) &= \hat{n} \cdot \left[\left(\hat{x}_1 \times \hat{\imath} \cos\psi + \hat{x}_2 \times \hat{\imath} \sin\psi\right) \sin e + \hat{x}_3 \times \hat{\imath} \cos e\right] \\
&= \hat{n} \cdot \left[\left(\frac{-\hat{n} \times \hat{\imath}}{\sin i} \cos\psi + \frac{\left(\hat{n} \times \hat{\imath}\right) \times \hat{\imath}}{\sin i} \sin\psi\right) \sin e + \left(\hat{n} \times \hat{\imath}\right) \cos e\right] \\
&= \hat{n} \cdot \left[\left(\hat{n} \times \hat{\imath}\right) \times \hat{\imath}\right] \frac{\sin\psi \sin e}{\sin i} \\
&= \hat{n} \cdot \left(\hat{\imath} \cos i - \hat{n}\right) \frac{\sin\psi \sin e}{\sin i} \\
&= -\sin i \sin e \sin\psi
\end{aligned}$$

Since i and e are positive, this implies

$$sign\left[\hat{n} \cdot \left(\hat{e} \times \hat{\imath}\right)\right] = sign(-\sin\psi)$$

[2]The constraint $0 \leq |g| \leq \pi$ directly follows from the assumption that neither i nor e can be greater than $\frac{\pi}{2}$; g cannot be greater than $i + e$. This implies that the sign of $\sin g$ is the same as the sign of g.

Using this result, Equation 2.5 then reduces to:

$$sign(g) = sign(-\sin\psi) \tag{2.6}$$

Next, to obtain the sign of Λ, one can similarly begin by considering the conditions under which it will be positive and negative, and assigning to Λ the sign of the dot product between $\hat{p}$ and some vector characteristic of it. The vector in question is $\hat{n} \times \hat{e}$; by inspection of Figures 2.4, this vector should point South of the luminance equator whenever Λ is negative, and North of it whenever Λ is positive. Using this principle, the sign of the longitude can be then be written explicitly:

$$sign(\Lambda) = sign\left(\hat{p} \cdot (\hat{n} \times \hat{e})\right)$$

In this case, simplifying this expression will first require expressing $\hat{e} \times \hat{\imath}$ in terms of $\hat{\imath}$ and $\hat{n}$, again using Equation 2.1, 2.2, and 2.4, and then simplifying:

$$\begin{aligned} \hat{e} \times \hat{\imath} &= \left(\left[\left(\frac{-\cos i}{\sin i}\right)\hat{n} \times \hat{\imath}\right]\cos\psi + \left[\frac{\left(\hat{n} \times \hat{\imath}\right) \times \hat{\imath}}{\sin i}\right]\sin\psi\right)\sin e + \hat{n} \times \hat{\imath}\cos e \\ &= \left(\cos e - \cos\psi\cot i\sin e\right)\hat{n} \times \hat{\imath} + (-\sin\psi\csc i\,\sin e)\hat{n} + \left(\sin\psi\cot i\sin e\right)\hat{\imath} \end{aligned}$$

Next, performing the same for $\hat{n} \times \hat{e}$:

$$\begin{aligned} \hat{n} \times \hat{e} &= \sin e\left[\frac{\cos\psi\cos i}{\sin i}\hat{n} \times \hat{\imath} + \frac{\sin\psi}{\sin i}\hat{n} \times \left(\hat{n} \times \hat{\imath}\right)\right] \\ &= \sin e\left[\left(\cos\psi\cot i\right)\hat{n} \times \hat{\imath} + \left(\sin\psi\cot i\right)\hat{n} + \left(-\sin\psi\csc i\right)\hat{\imath}\right] \end{aligned}$$

Finally, the dot product between them may be obtained by considering how $\hat{\imath} \cdot \hat{n} = \cos i$, and $(\hat{n} \times \hat{\imath}) \cdot (\hat{n} \times \hat{\imath}) = \sin^2 i$, and how all other terms will be zero;

$$\hat{p} \cdot (\hat{n} \times \hat{e}) = \frac{sign(-\sin\psi)\sin e\cos i}{|\sin g|}\left[\cos\psi\left(\cos e - \cos\psi\cos i\sin e\right) + \left(\frac{\sin\psi}{\sin i}\right)^2\left(\cos e - 1\right)\right]$$

Hence, eliminating a factor of $sign(g)$ and all the factors of positive quantities from this expression gives the sign of Λ relative to g:

$$sign(\Lambda) = sign\left[\cos\psi\left(\cos e - \cos\psi\cos i\sin e\right) + \left(\frac{\sin\psi}{\sin i}\right)^2\left(\cos e - 1\right)\right] \tag{2.7}$$

2.4 The sample space

As in Salo and Karjalainen (2003), this project will adopt the convention that P is a conditional probability distribution of the values of $(\mu_0; \mu, \psi)$ instead of the values of $(i; e, \psi)$. This avoids the bias in probability density caused by the transformation between points (e_n, ψ_n) in the domain $D^* \equiv [0, \frac{\pi}{2}] \times [0, 2\pi]$ and points $\hat{e}_n$ in H. To illustrate this effect, consider how a uniform reflection law, being constant with respect to e and ψ ($P = \frac{1}{2\pi}$ for example), should generate isotropically distributed ray vectors in H, each associated with e and ψ through the parameterization given by Equation 2.2. Then, N points of $(e_n, \psi_n) \in D^*$ corresponding to N uniform points $\hat{e}_n \in H$ will have a non-uniform density that is proportional to $\sin e$. This is because the (average) number of points contained within an area element $de\ d\psi \subset D^*$ corresponding to points contained in a surface element $dA \subset H$ will be $\frac{N}{2\pi} dA\ \sin e\ de\ d\psi$ (where $A \subset B$ denotes "A is a subset of B"). Conversely, a uniform density of points in (e, ψ) will translate to a non-uniform density of points on the hemisphere; points will be "compressed" at lower values of e due to the lesser range of positions that variation of ψ can result in. This may be remedied by generating values of e with probability density proportional to $\sin e$, and if the transformation law of probabilities (Press *et al.*, 2007, p.363) is used, the resulting method is essentially identical to sampling a value of μ uniformly in the interval $[0, 1]$ and then treating e as $\cos^{-1} \mu$. Furthermore, a similar change of variables may be adopted for the incident angle i

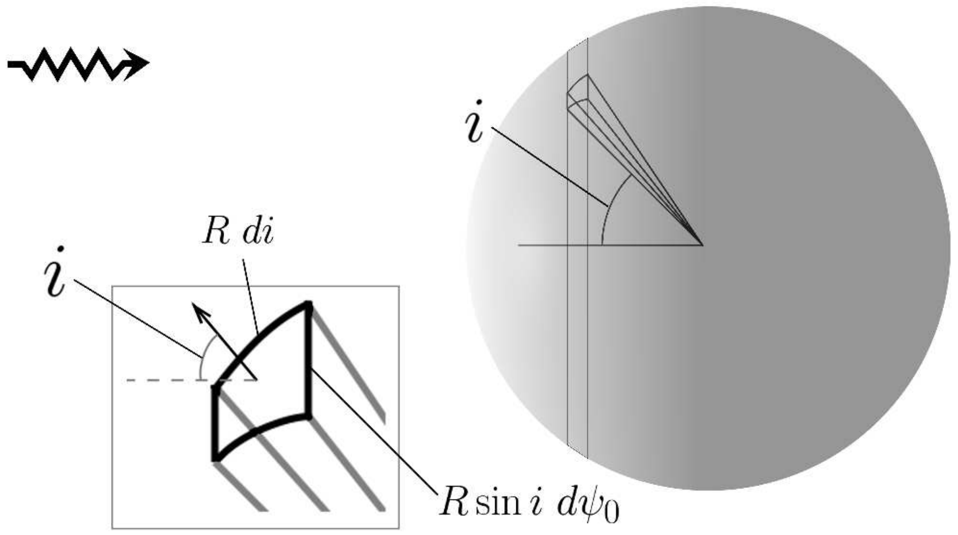


Figure 2.5: Incidence of a photon packet on a ring particle, assumed spherical.

wherever the analysis of P, i.e. the numerical integration performed in Section 3.4, is concerned; greater weight should ideally be given to more likely angles of incidence during performance testing. Although no histogram for i can be practically obtained before running the ray tracing simulation, a moderately safe representation of the distribution of incident angles can be found as follows: assume a spherical ring particle, and a photon packet randomly incident on it, as shown in Figure 2.5. Consider then how an infinitesimal surface element on the particle's surface at incident angle i having an area of $dA = R^2 \sin i \; di \; d\psi_0$ should have a cross-sectional area of $\hat{i} \cdot (dA\hat{n}) = R^2 \sin i \cos i \; di \; d\psi_0$ (from the perspective of the incident direction). The combined cross-sectional area of all such infinitesimal surface elements at incident angle i is:

$$R^2 \sin i \cos i \; di \int_0^{2\pi} d\psi_0 = 2\pi R^2 \sin i \cos i \; di$$

The probability of the ray intersecting the object's surface within a region is proportional to the region's cross-sectional area, and therefore, the probability distribution of i is:

$$p(i) = I \sin i \cos i$$

where I is a normalization constant. Analysis of the function P will hence use this probability distribution in order to give more weight to the more likely values of i that will be encountered during the simulation, and it accomplishes this by similarly relating i to a variable that is uniformly distributed, as e is to μ. Using the transformation method as described in Press *et al.* (2007, p.363), values of i with such a probability distribution can be related to values of a uniformly-distributed variable, denoted ρ, through the transformation law of probabilities:

$$\rho = \int_0^i p(i')di'$$

Computing I and the indefinite integral yields:

$$\rho = \sin^2 i \tag{2.8}$$

To summarize: values (μ, ψ) will be sampled within the domain $[0,1] \times [0, 2\pi]$, abbreviated D, with distribution P. The function $P(i; e, \psi)$ is considered a conditional probability distribution of (μ, ψ) given i. Wherever modeling or analyzing the reflection law is concerned, the change of variables $i = \sin^{-1} \rho$ will be used. To simplify notation, the function P' is defined as follows:

$$P'(\rho; \mu, \psi) \equiv P\left(\sin^{-1}\left(\sqrt{\rho}\right);\ \cos^{-1}(\mu)\,,\ \psi\right) \tag{2.9}$$

The variable ρ is related to μ_0 via $\rho = 1 - \mu_0^2$ (essentially, $1 - \cos^2 i$) and is thus set constant during each scattering event. Values of ρ and μ are constrained to the interval $[0, 1]$, while the value of ψ is constrained to $[0, 2\pi]$. The domain $[0, 1] \times [0, 1] \times [0, 2\pi]$ in which triplets (ρ, μ, ψ) exist is abbreviated D'.

3 General description of the developed method

The method, summarized in this section, takes the same essential form of a *universal generator* as described in Hörman *et al.* (2003); in order to generate random sample points according to an arbitrarily-chosen distribution, it utilizes the rejection sampling algorithm, and automatically self-optimizes to the distribution in use during an initial *setup* procedure which, although computationally expensive, only needs to be executed once for each distinct P'. The initial performance cost (a.k.a. the *setup time*) of doing so will be negligible if the number of reflections in the ray-tracing simulation is considerably large; reducing the *marginal execution time* (the time required to generate one random pair of μ, ψ) is more important than reducing the setup time when the sampling algorithm will be used a large number of times (Hörman *et al.*, 2003, p.5).

3.1 Rejection sampling

The rejection sampling method will generate random pairs of (μ, ψ) with probability density P' by first generating *trial points* of (μ, ψ) within the sample space (D) distributed according to a *comparison function*, denoted f_c (which must always be greater than or equal to P'), and a third uniformly-distributed random deviate (referred to here as the *trial gauge*, and denoted r) in the interval between zero and the value of f_c. However, triplets (μ, ψ, r) such that r does not have a value less than P' (as illustrated by the univariate example depicted in Fig. 3.1) are rejected in favor of trying to generate a new trial point and trial gauge. Points (μ, ψ, r) that are not rejected will in this way be distributed uniformly in the space contained beneath the function P' (a subset of $D \times \Re^+$), and the values of μ and ψ (their "shadows" in D) will be distributed according to P' (Hörman *et al.*, 2003, p.17,18). The general structure of the rejection sampling algorithm (using a "squeeze function") is as follows:

1. Generate random values of μ and ψ using a polynomial comparison function or a constant, in either case denoted f_c.

2. Generate r uniformly in the interval $[0, f_c(\rho; \mu, \psi)]$ (where ρ is a pre-determined value in each instance).

3. Evaluate the *squeeze function* (denoted f_s). The squeeze function must always be less than the distribution P', and is meant to improve efficiency by being less expensive to execute than P' (see "The Squeeze principle" in Hörman *et al.*, 2003, p.20).

4. **If:** $r < f_s(\rho; \mu, \psi)$ close the loop and return the values of the trial point;

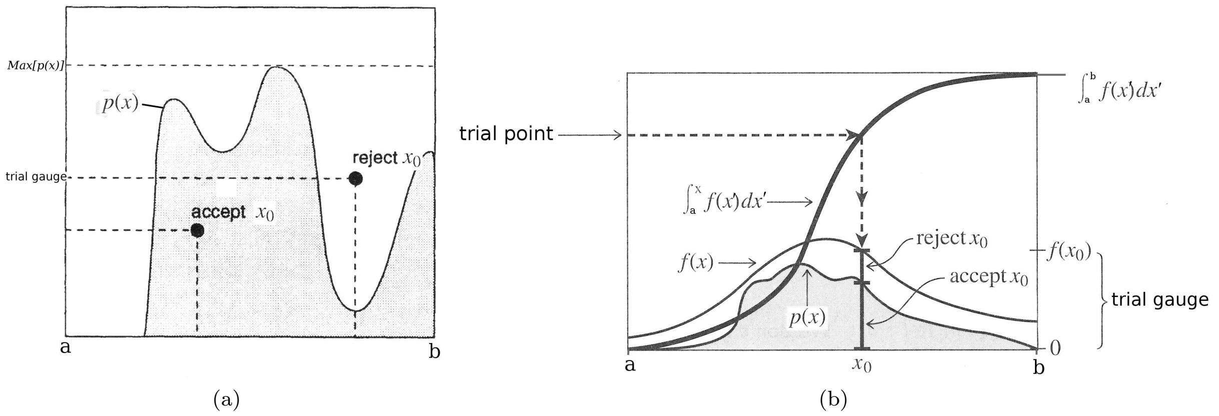


Figure 3.1: (a) (Adapted from Landau and Paéz, 2004, p. 108): Rejection sampling with a constant: generate a random deviate uniformly within the domain, and then another (the "trial gauge") between zero and some value that is greater than the distribution within the domain; reject if it is greater than the distribution evaluated at the first random value, accept otherwise; (b) (Adapted from Press *et al.*, 2007, p. 366): Rejection sampling using a comparison function: generate a random deviate within the domain according to a comparison function, and then generate the trial gauge between zero and the value of the comparison function evaluated at the trial point value, and perform a similar test as with the constant.

Else if: $r < P'(\rho; \mu, \psi)$ close the loop and return the values of the trial point;

Else Start over at step 1.

3.2 Modeling P'

The purpose of the comparison function is to reduce the volume containing points that must be rejected. This yields a gain in performance (Hörman *et al.*, 2003, p.22) depending on how much volume is eliminated by the function and how costly it is to perform trial point generation with it. To create comparison and squeeze functions to loosely mimic the shape of P', the method uses Taylor series expansions, and the means of performing this is described in greater detail in Appendix B.2. Although the method can in theory be generalized to arbitrary order, the approximations will be limited in this project to 3^{rd} order due to the limitations of achieving higher-order numerical derivatives with sufficient accuracy. The orders of approximation will be denoted m_c for the comparison and m_s for the squeeze function.

While the Taylor series may in some cases provide a moderately reliable polynomial approximation to the distribution P', there is never any guarantee that it will always remain greater than (or, for the squeeze function, less than) the distribution function P'. Securing this is essential to having functions f_c and f_s that work properly; as stated in Section 3.1, the comparison function must always be greater than P', and the squeeze function must always be less than P'. To correct for the imperfections of Taylor approximations, the initialization process will run a "safety" test/adjustment on each of the polynomials. This procedure works as follows (with p denoting a generic power series approximation):

Before testing: Define three new objects:

- A function (denoted d) that computes $p(\rho;\mu,\psi) - P'(\rho;\mu,\psi)$
- A function (denoted $-d$) that computes $P'(\rho;\mu,\psi) - p(\rho;\mu,\psi)$
- A 19-component vector of double-precision numbers (denoted Δ_c here)
- Another 19-component vector of doubles, Δ_s

Main testing procedure: For each order m, from 2 to 20, perform the following tasks:

1. Use the IDL function `AMOEBA` to compute the minimum of d within D', and store the returned value in $\Delta_c[m-2]$;
2. Again use `AMOEBA` to compute the minimum of $-d$ within D', and store the returned value in $\Delta_s[m-2]$;

Result: The array $\Delta[l, m-2]$ corresponds to corrections that must be subtracted from the 0^{th}-order power series term of each m^{th} order Taylor approximation, for m from 2 to 20, so that that the comparison functions are always greater than, and the squeeze functions always less than, the function P' within the domain D'.

In light of this method for making the polynomial approximations safe for use as comparison/squeeze functions, an important consideration in assessing the efficacy of using polynomials (especially Taylor series approximations) is how big of an adjustment must be made, because a larger adjustment will adversely affect the performance of the trial point generator; the direct consequence is that a larger volume of points will have to be rejected. As mentioned in Section 2.3.2, irregularities may exist in the distribution around $i = e$ and $\psi = 0$ (zero phase angle), and while this is to be expected (because of the spike in luminosity caused by the opposition effect; see Hapke (1993, pp.216-235)) it will have the possible consequence of making the random number generation process inefficient and problematic because of the difficulty of creating polynomials modeled after the shape of the reflection law. A sharper opposition spike thus may translate to the rejection of a high if not higher volume of trial points than would result from using a constant/uniform comparison function, especially if the necessary adjustments are large. Although multivariate transformed density rejection, as described in Hörman *et al.* (2003), may possibly be able to avoid this problem by constructing a polytope-shaped comparison function out of hyperplanes tangent to the convex density-transformed distribution, the implementation of that method is not covered in this project. Instead, given that it is possible to test whether using a polynomial is more or less efficient than using a constant (by the method described in Section 3.4), the chosen approach to random deviate generation is to test whether or not a simple polynomial comparison function can offer a better performance, and to use one if it does.

3.3 Polynomial plane-sweep for bivariate trial point generation

Trial point generation is in essence the sampling of points uniformly within a region that has one dimension more than the number of variables being generated by the rejection process, as depicted in Fig. 3.1. In this project, the chosen method for accomplishing this will employ the same principle as the sweep-plane technique discussed in Hörman *et al.* (2003), but this principle will be used on the volume contained beneath a polynomial comparison function instead of a polytope. By generating the first of the two random variables according to the marginal distribution[3], the second variable can then be generated according to the conditional distribution that results from fixing the value of the already-generated first variable and the predetermined value, ρ. During each step of this process, the method will utilize the transformation law of probabilities (Press *et al.*, 2007, p.363), and this requires the cumulative distribution, which is the indefinite integral of a distribution, to be determined from the marginal/conditional distributions of μ and ψ. For the first variable, ψ, the transformation law reads:

$$\int_0^1 \int_0^{\psi} f_c(\rho; \mu, \psi')\ d\psi'\ d\mu = R[0,1] \cdot \int_0^1 \int_0^{2\pi} f_c(\rho; \mu, \psi)\ d\psi\ d\mu \tag{3.1}$$

This can be interpreted as sweeping a plane through D in the direction of ψ until reaching an amount of volume under f_c that is uniformly distributed between zero and the total volume under f_c in D (given a value of ρ). After solving this equation for ψ, the problem of generating the next random variable, μ, reduces to the univariate case of obtaining a random value by the transformation method, and is essentially the same as depicted in Figure 1(b);

$$\int_0^{\mu} f_c(\rho, \psi; \mu')\ d\mu' = R[0,1] \cdot \int_0^1 f_c(\rho, \psi; \mu)\ d\mu \tag{3.2}$$

Now, because the comparison function is a finite power series of ρ, μ and ψ, the indefinite integral of such a function with respect to any of the three variables should also be a finite power series. Arranging the polynomial coefficients by the order of the term they correspond to thus optimizes this process and ensures that no redundant operations are performed (see Appendix B.2). Furthermore, in this project, no approximations higher than order 3 are used, which allows the use of the analytical solutions to the quartic, cubic and quadratic equations (as described in Abramowitz and Stegun, 1970, pp.17-18) to solve for values of μ and ψ and thus generate them via the transformation method.

[3]The marginal distribution is the result of integrating the comparison function over all possible values of the second variable (Feller, 1970, p.67).

3.4 Performance testing

The `setup` procedure not only creates comparison functions, but also a test of performance for each order of approximation. To demonstrate how it accomplishes this, consider first how the execution time of a process is essentially the sum of the execution times of all sub-processes, and how if any process is repeated, its contribution to the total execution time is expected to be the product of the time it takes to execute once and the average number of times it must be executed. Next, it may be noted that the average total number of trial points that will be needed is equal to twice the value of the *rejection constant* (Hörman *et al.*, 2003, p.22), and in the case of this project, this value (denoted N) is equal to:

$$N = 2\frac{\int_{D'} f_c d\mathscr{V}}{\int_{D'} P' d\mathscr{V}} \tag{3.3}$$

(in this case, $d\mathscr{V} = d\rho\ d\mu\ d\psi$). The test of efficiency will also require knowing how many times (on average) that P' will have to be evaluated versus how many trial points total will be necessary, since the squeeze function is expected to be less costly than P' to evaluate. Using the formula for this value from Hörman *et al.* (2003, p.22), it should be in this case:

$$N_{P'} = \frac{\int_{D'} f_c\ d\mathscr{V} - \int_{D'} f_s d\mathscr{V}}{\int_{D'} P' d\mathscr{V}} = \frac{N}{2} - \frac{\int_{D'} f_s d\mathscr{V}}{\int_{D'} P' d\mathscr{V}} \tag{3.4}$$

Using these numbers of repetitive executions, the average total amount of computational resources required to find one point by rejection sampling can thus be determined by revisiting and examining the generic algorithm for rejection sampling (in Section 3.1), and assigning to each *operation* an amount of time consumed in its execution, denoted $O(operation)$, and then multiplying by the average number of times that it will be executed. One iteration of the algorithm requires:

- Generation of a *trial* point, with trial gauge: $O(trial)$
- An execution of the squeeze function: $O(f_s)$
- Comparison of two double-precision numbers: $O(test)$
- (If the trial gauge was not less than the squeeze function):
  - An execution of the distribution function: $O(P')$
  - A comparison of two double-precision numbers: $O(test)$

Hence, the generation of one point using a polynomial, denoted as the operation *poly*, requires (on average):

$$O(poly) = N * [O(trial) + O(f_s) + O(test)] + N_{P'} * [O(P') + O(test)] \tag{3.5}$$

Then, using the definitions:

$$\begin{aligned} I_{P'} &\equiv \int_{D'} P' d\mathscr{V} \\ I_c &\equiv \int_{D'} f_c \, d\mathscr{V} \\ I_s &\equiv \int_{D'} f_s \, d\mathscr{V} \end{aligned}$$

Also, with the abbreviated notation:

$$\begin{aligned} O(pre) &\equiv O(trial) + O(f_s) + O(test) \\ O(exe) &\equiv O(P') + O(test) \end{aligned}$$

Equation 3.5 can then be re-constructed for the performance test as follows:

$$O(poly) = \frac{I_c}{2I_{P'}} \cdot O(pre) + \frac{I_c - I_s}{I_{P'}} \cdot O(exe) \tag{3.6}$$

If a constant is being used instead of the polynomial trial point generator, the quantity will be:

$$O(const) = \frac{\pi Max(P')}{I_{P'}} \cdot O(pre) + \frac{2\pi Max(P') - I_s}{I_{P'}} \cdot O(exe)$$

Thus, by performing the integral $I_{P'}$ and the integrals I_c and I_s (for each m_c and m_s) using the IDL function `INT_3D`, and then running the `PROFILER` procedure to determine $O(pre)$ and $O(exe)$, `setup` computes the marginal execution time of the rejection sampling algorithm for each of the trial point generators available. Furthermore, since the method of approximation used to generate comparison functions permits the use of any arbitrary starting point within the domain D', the overall initialization procedure can perform a cursory search over D' for a good approximation by picking several random points, performing the setup and performance testing for each point, and then picking the best comparison function out of all those available at the end of the search.

4 Development and testing of a prototype angle generator

Throughout the development process, various adjustments and accommodations were found necessary in order to circumvent the various difficulties that were encountered:

A search over the sample space; given the arbitrary shape of the function and the limit of approximations to 3rd order, it was at a later time in this project deemed appropriate to make use of the flexibility of the approximation method to search for better approximation starting points.

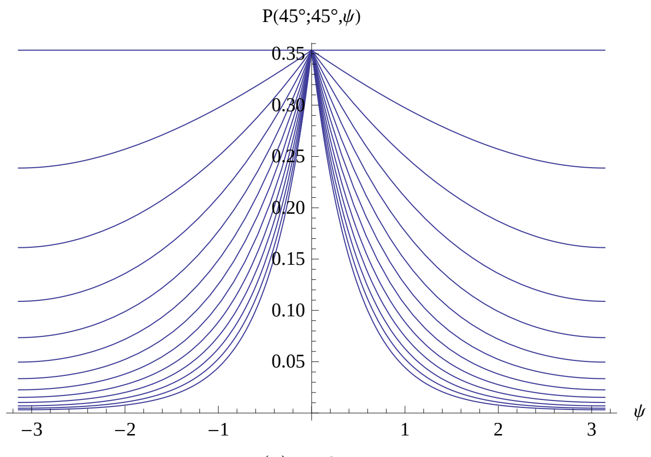
Adaptive precision and maximum iterations; For phase functions having a more pronounced opposition peak, the maximization routine `AMOEBA` often failed to converge to the desired precision due to the irregular shape of the function. The step taken to circumvent this was to set the precision and iterations to initial values, and then reduce the precision/increase the maximum number of iterations until the function converged, and to afterwards make an additional safety correction to the maximum by multiplying the result by $(1 + precision)$, where $precision$ is increased in each iteration but is not allowed to exceed 10^{-1}. The procedure then quits if amoeba still fails to converge.

Use of analytical solutions to polynomials; because of the restriction of the comparison functions to 3rd order and lower, the cumulative distributions were limited to 4th order and lower, which serendipitously permitted use of the analytical solutions to the quartic and cubic equations. The benefit of using them is in their efficiency and safety; the analytical solutions give a much more straightforward path to computing the roots with very high precision in very few operations, as opposed to the polynomial root-finding function `FZ_ROOTS`. The latter function, which uses Laguerre's method to determine the roots, was an option explored in the earlier stages of development, but it was found to be unreliable for use in automatic procedures called upon very frequently, since it would occasionally return null and throw the error that it failed to converge to the desired precision.

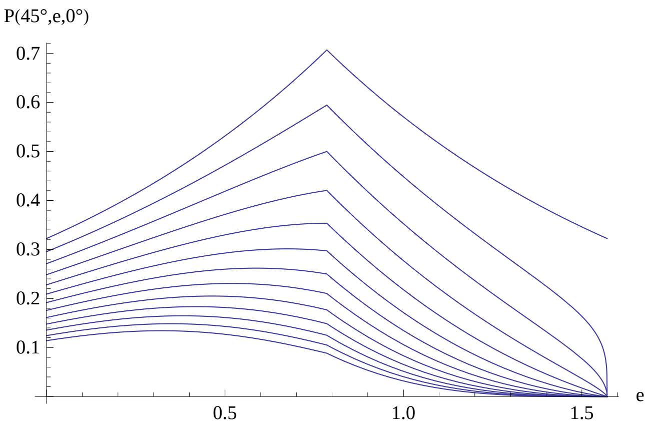
Organized retention of all setup constants: the procedure stores all constants created by the setup at each point within a structure declared within a common block shared between functions associated with trial point generation, so that in any larger ray-tracing simulation, the optimal setup remains available for use.

4.1 Input

The method described throughout section 3, which in this project has been implemented in IDL, used the following function (J. Cuzzi, personal communication, 2008) as its probability distribution during all of the



(a) scale=0.8



(b) scale=0.8

Figure 4.1: A plot of the effect of the Minnaert and steepness parameters on the shape of the reflectance law, for the examples: (a) $\nu = 2$ while $A = 0$ to $A = 2$, $dA = 0.25$, $i = e = \pi/4$, $-\pi \leq \psi \leq \pi$, and (b) $A = 1$ while $\nu = 1$ to $\nu = 4$, $d\nu = 0.25$ and $\psi = 0$, $i = \pi/4$ and $0 \leq e \leq \pi/2$. In both (a) and (b), the highest curve corresponds to the lowest value of the varied parameter, and all curves below them correspond to successively higher values.

tests:

$$P(e, g) = \mathbf{e}^{-Ag}\mu_0^{\nu}\mu^{\nu-1} \tag{4.1}$$

In this case, A is the *steepness* (a measure of the opposition spike's intensity) and ν is the *Minnaert parameter*. After the change of variables described in Sections 2.3.2 and 2.4, this function may be evaluated as:

$$P'(\rho; \mu, \psi) = (1-\rho)^{\nu/2}\,\mu^{\nu-1}\mathbf{e}^{-A\cos^{-1}\left[\sqrt{\rho(1-\mu^2)}\cos\psi + \mu\sqrt{1-\rho}\right]} \tag{4.2}$$

Considering the properties of this function, the following results were expected of the approximations and performance tests:

Unpredictable order-to-efficiency relations; Due to the irregularity of the reflectance law, and the tendency of the approximations to diverge from the distribution for larger displacements from their starting points, it is difficult to predict how useful a given order of approximation will be based on its starting point and the shape of the function. Thus, the most efficient orders of approximation, as well as their starting points, are expected to show variety.

Very small performance benefits of using a squeeze function; the reflectance law is in this case not very costly to evaluate compared to 3-D polynomials of order 3 and lower, so the advantage of using a squeeze function and the differences in performance caused by using different squeeze functions should be diminished; the the term containing dependence of Equation 3.6 on the volume contained by f_s is directly proportional to the cost of evaluating the function.

Slow apparent convergence to the true density; although points sampled using the resulting trial point generator will be properly distributed according to the reflectance law, the density will appear to converge to the true density rather slowly because a very small bin size will be needed to properly gauge the density of points near points where there is a high amount of variation in the density, i.e. near opposition.

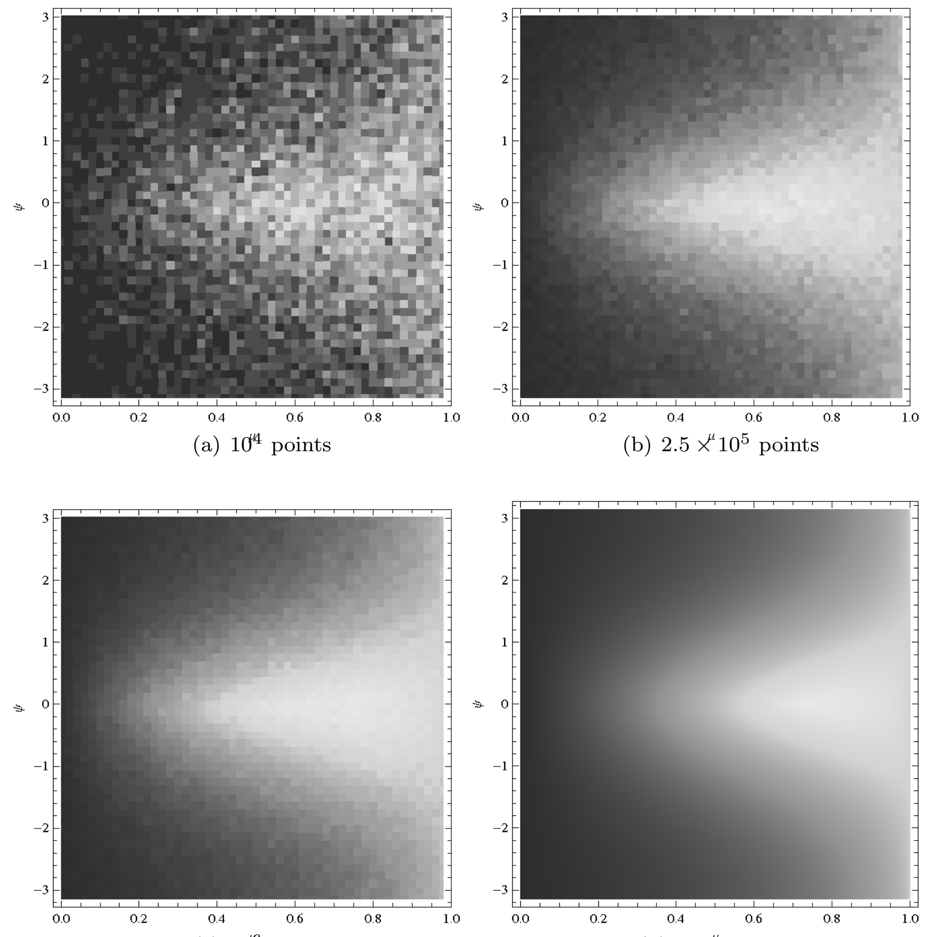
The parameters A and ν were set to 1 and 2, respectively, for this test. As demonstrated in Fig. 4.1, these two values should correspond to a slightly pronounced opposition peak, and greatly diminished intensities at lower emitted angles, while lower values of the steepness and Minnaert parameter should result in the reflection law having a less irregular shape. For that matter, it is reasonable to conclude that the performance of the method for values of A less than the values at which the method is benchmarked should be greater; as explained in Section 3.2, a more highly irregular distribution will require greater safety adjustments to be made, resulting in lower performance. Setting $A = 1$ therefore provides a rough measure of the minimum performance for all values of A lower than 1, although this choice has been made arbitrarily nevertheless.

4.2 Output testing and performance results

To ensure that the algorithm successfully generates random values with proper density, a cursory test of it was performed by generating large quantities of random points for a given incident angle ($\pi/4$) and observing the count of points in an array of bins, shown in Fig. 4.2. Furthermore, at the end of the `SEARCHSETUP` procedure, which performed a survey of approximations taken at 20 uniformly-sampled points ("$\mathbf{x}_0$") in the domain D', four sets of approximations were chosen for display purposes. While the performances of these approximations, displayed in Fig. 4.3 and Fig. A.1, are not all very good, they each contain a best point, and for each of the four sets of approximations chosen, the best approximations in the set correspond to one of the four most efficient trial point generators in the survey. These results are directly in line with expectations; the starting points that yielded the four most efficient trial point generators are not particularly close to one another, and the best order of approximation differs for each point, along with relations between order of approximation and performance that are also different for each point. The setup procedure succeeded in finding a useful approximation; at the starting point $(0.866, 0.412, 6.037)$, the approximation orders $m_c = 3$ and $m_s = 2$ resulted in the performance:

$$\frac{O(poly)}{O(const)} = 0.472707$$

This corresponds to a better trial point generator than a uniform one; the marginal execution time for this polynomial comparison/squeeze function combination is less than half that of the former.



(a) 10^4 points

(b) 2.5×10^5 points

(c) 10^6 points

(d) Equation 4.2

Figure 4.2: Program output as a density plot of points, using 50 bins in each dimension, with $A = 1$, $\nu = 2$ and $i = \frac{\pi}{4}$. To display the opposition spike in the center of each plot, the bins were transposed in their values of the ψ coordinate using the branch cut $-\pi \leq \psi \leq \pi$. Note the slow convergence to the function in Equation 4.2

5 Conclusions

It has been demonstrated that, for a reflection law with steepness less than or equal to unity, it is possible to achieve better performance of the rejection sampling algorithm with polynomial approximations as comparison functions, even with a moderately unreliable method of approximating, than with using a uniform comparison for generating trial points; using a moderately small survey of points, optimal comparison functions can be found. Additional measures not covered in this work that could be taken to find even more efficient comparison functions include searches around each of four optimal approximation points, or an entirely different method of polynomial approximation that is more reliable than the Taylor expansion; insofar as it is trivial to obtain analytical expressions for the cumulative and marginal distributions from a 3-variable polynomial (per the method described in Section B.3), most of the the techniques used in this project may

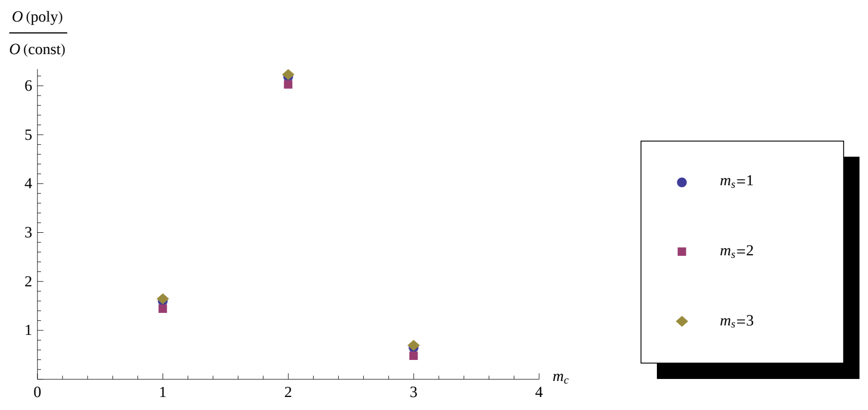


Figure 4.3: Mean marginal execution times of the rejection sampling algorithm, using the trial point generators made at the randomly chosen approximation starting point $\mathbf{x}_0 \approx (0.866, 0.412, 6.037)$, expressed as fractions of the execution time of the uniform trial point generator. Note: errors for these values were significantly low (less than 10^{-2}), and are therefore invisible at the scales shown in these plots. Note also the very small differences in performance between the various orders of the comparison function. The performance measurements for the approximation starting points that resulted in the 2nd, 3rd and 4th most efficient approximations are displayed in Appendix A (Fig. A.1).

be generalized to arbitrary orders of approximation if the approximation's coefficients can be placed into an array with indices corresponding to the powers of the three variables, as accomplished by the method derived in Appendix B.2. While this still poses the difficulty of obtaining a reliable method for solving the polynomial equations derived from the transformation law, and closed-form solutions to polynomials higher than 4th order may not be feasible, numerical root-finding techniques, i.e. Newton-Rhapson, can be made more efficient and reliable by computing derivatives analytically, which is not difficult if the coefficients of the polynomials are organized in this way.

References

Abramowitz, M., and I. A. Stegun 1970. *Handbook of Mathematical Functions.* Dover.

Akimov, L. A. 1979. On the brightness distributions over the lunar and planetary disks. *Soviet Astronomy* **23**, 231–235.

Boas, M. L. 1983. *Mathematical Methods in the Physical Sciences* (2 ed.). John Wiley & Sons.

Bowman, K. P. 2005. *Introduction to Programming with IDL.* Academic Press.

Courant, R., and F. John 1999. *Introduction to Calculus and Analysis*, Volume 1. Springer. Reprint of the 1989 Edition.

Feller, W. 1970. *An Introduction to Probability Theory and Its Applications*, Volume 2. John Wiley & Sons.

Hapke, B. 1993. *Theory of Reflectance and Emittance Spectroscopy.* Cambridge University Press.

Hörman, W., J. Leydold, and G. Derflinger 2003. *Automatic Nonuniform Random Variate Generation.* Springer.

Kreslavsky, M., Y. G. Shkuratov, Y. I. Velikodsky, V. G. Kaydash, and D. G. Stankevich 2000. Photometric properties of the lunar surface derived from clementine observations. *Journal of Geophysical Research* **105**, 20,281–20,295.

Landau, R. H., and M. J. Paéz 2004. *Computational Physics.* Wiley-VCH.

Mardsen, J. E., and A. J. Tromba 2003. *Vector Calculus* (5 ed.). Freeman.

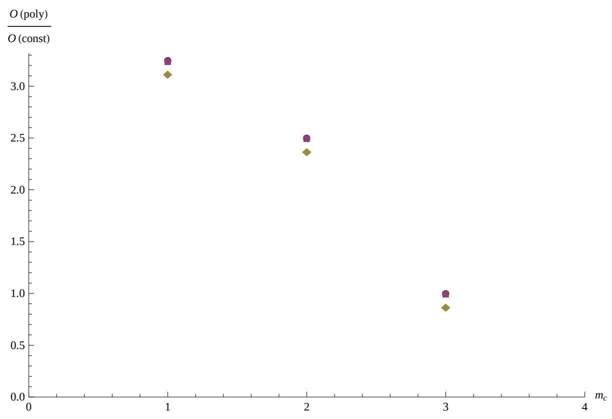
Press, W. H., S. A. Teukolsky, and W. T. Vetterling 2007. *Numerical Recipes: The Art of Scientific Computing* (3 ed.). Cambridge University Press.

Salo, H., and R. Karjalainen 2003. Photometric modeling of saturn's rings: Monte carlo method and the effect of nonzero volume filling factor. *Icarus* **164**, 428–460.

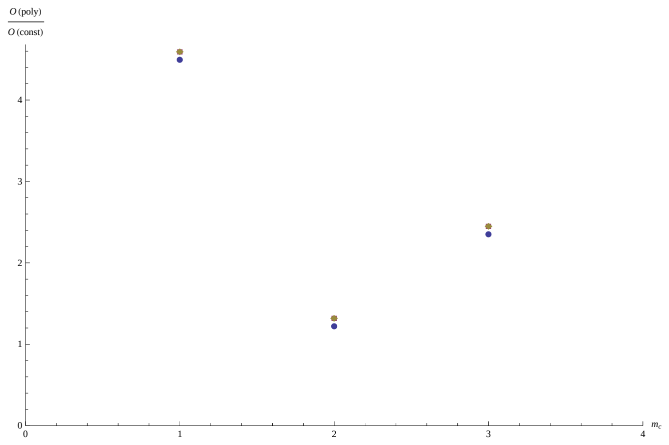
Shkuratov, Y., D. Petrov, and G. Videen 2003. Classical photometry of prefractal surfaces. *Optical Society of America 20*(11), 2081–2092.

Appendix

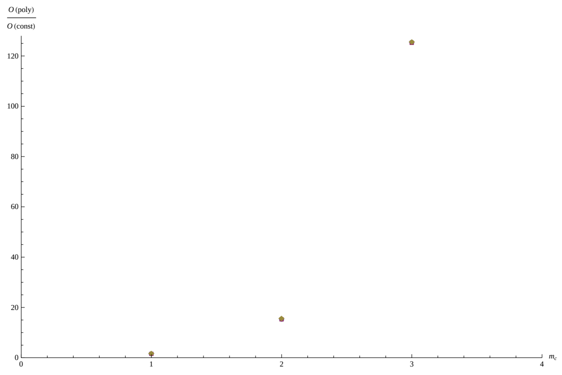
A Additional Figures and Tables



(a) $\mathbf{x}_0 \approx (0.725, 0.586, 3.677)$



(b) $\mathbf{x}_0 \approx (0.313, 0.745, 5.111)$



(c) $\mathbf{x}_0 \approx (0.770, 0.775, 6.200)$

Figure A.1: Performance of approximations ranging in order from 1 to 3, using the same legend as Fig. 4.3. These approximation starting points resulted in the second through fourth most efficient approximations, respectively.

$A \times B$	Cartesian product (if A and B are sets); cross product (if A and B are vectors)
D^*	The set $[0, \frac{\pi}{2}] \times [0, 2\pi]$ containing pairs (e, ψ)
D	the set $[0, 1] \times [0, 2\pi]$ containing pairs (μ, ψ)
D'	the set $[0, 1] \times [0, 1] \times [0, 2\pi]$ containing triplets (ρ, μ, ψ)
e	emitted polar angle
$\in$	"...is an element of..."
$\hat{e}$	unit vector of emitted ray's direction
$\hat{e}_0$	unit vector of incident ray's direction
f_c	comparison function
f_s	squeeze function
g	Phase angle
H	the unit hemisphere above the local tangent plane on the surface of a simulated ring particle
i	incident polar angle
$\hat{\imath}$	source of incident ray; points in opposite direction of $\hat{e}_0$
L	Photometric latitude / luminance latitude
Λ	Photometric longitude / luminance longitude
m_c	order of the comparison function polynomial
m_s	order of the squeeze function polynomial
$\hat{n}$	unit vector normal to the surface of the ring particle at the point of incidence
$O(operation)$	denotes the time consumed in the execution of an *operation*
P	reflection law function of the variables (i, e, ψ)
P'	change of variables in P to (ρ, μ, ψ)
ψ	azimuthal angle between the projections of $\hat{\imath}$ and $\hat{e}$ to the local tangent plane
r	trial gauge value
$\Re^+$	The set of all positive real numbers
ρ	The value $\sin^2 i$
$\subset$	"...is a subset of..."
$\mathbf{x}_0$	point at which an approximation to a function is made

Table 1: Table of symbols

B Numerical techniques

Apart from preexisting IDL functions, certain techniques were found necessary to derive for the purposes of this experiment;

B.1 Partial derivatives

Finding partial derivatives numerically will be of importance in the methods of Section B.2, so a procedure for obtaining them, using midpoint approximation, is developed and discussed here. To begin with, the mixed partial derivative in two dimension is (Press *et al.*, 2007, p.231):

$$\frac{\partial^2 f}{\partial x \partial y} \approx \frac{[f(x+h,y+h) - f(x+h,y-h)] - [f(x-h,y+h) - f(x-h,y-h)]}{4h^2}$$

Re-writing this formula to show how it uses the same method for computing the second partial as for the first partial will provide insight as to how to develop a generic algorithm for computing mixed partials;

$$\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right) \approx \frac{\frac{f(x+h,y+h)-f(x+h,y-h)}{2h} - \frac{f(x-h,y+h)-f(x-h,y-h)}{2h}}{2h}$$

In this expression, both of the two terms in the main numerator are the numerical partial derivative of f with respect to y, evaluated at two different values of x. Taking this concept a step further and using the abbreviated notation $f(+,-,+) \equiv f(x+h, y-h, z+h)$, it may be concluded that:

$$\frac{\partial}{\partial x}\left(\frac{\partial}{\partial y}\left(\frac{\partial f}{\partial z}\right)\right) \approx \frac{\frac{\frac{f(+,+,+)-f(+,+,-)}{2h} - \frac{f(+,-,+)-f(+,-,-)}{2h}}{2h} - \frac{\frac{f(-,+,+)-f(-,+,-)}{2h} - \frac{f(-,-,+)-f(-,-,-)}{2h}}{2h}}{2h}$$

Note how the numerical partial derivative applied “last” (the outermost expression) contains the nested expressions for the numerical partial derivatives applied “earlier”, while the expression for the “first” derivative is present at the innermost level, where it is just evaluated at different values of (x, y, z) in order to compute the higher derivatives. In light of this pattern, a recursive algorithm is clearly the most straightforward way to implement higher-order partial derivatives. Using this abbreviated notation:

$$\partial(i,j,k)(\rho,\mu,\psi) \equiv \frac{\partial^{(i+j+k)} P'}{\partial \rho^i \partial \mu^j \partial \psi^k}|_{(\rho,\mu,\psi)}$$

The algorithm is as follows:

```
$\partial$ : require $i, j, k, \rho, \mu, \psi$
```

1. *If* $i > 0$, then return:

$$\frac{\partial(i-1,j,k)(\rho+\frac{h}{2},\mu,\psi)-\partial(i-1,j,k)(\rho-\frac{h}{2},\mu,\psi)}{h}$$

2. *Else if* $j > 0$, then return:

$$\frac{\partial(0,j-1,k)(\rho,\mu+\frac{h}{2},\psi)-\partial(0,j-1,k)(\rho,\mu-\frac{h}{2},\psi)}{h}$$

3. *Else if* $k > 0$, then return:

$$\frac{\partial(0,0,k-1)(\rho,\mu,\psi+\frac{h}{2})-\partial(0,0,k-1)(\rho,\mu,\psi-\frac{h}{2})}{h}$$

4. *Else* (i.e. if all three of them are zero) then return:

$$P'(\rho;\mu,\psi)$$

While this algorithm works in the generic sense of it being able to correctly generate the expression for higher-order mixed partial derivatives, it is prone to extreme inaccuracies at even lower orders due to the limitations of machine precision. Therefore, at each level of recursion, an intermediary function utilizing Ridders' algorithm as described in Press *et al.* (2007, p.231-232) will be used to extrapolate better approximations for the numerical derivatives. However, even with this safeguard in place, preliminary code testing has shown it to be largely unreliable and costly to evaluate for derivatives higher than order 3, hence the decision to limit the comparison functions to 3rd-order approximations.

B.2 Multi-dimensional Taylor expansions

The method derived here will be used to generate comparison and squeeze functions as Taylor expansions in ρ, μ and ψ, taking advantage of P being a well-defined and (mostly) regular function. It is first worthwhile to introduce the 3rd-order Taylor approximation in n dimensions (Mardsen and Tromba, 2003, p.199):

$$\begin{aligned} f(\mathbf{x}_0+\mathbf{h}) &= f(\mathbf{x}_0)+\sum_{i=1}^{n} h_i\partial_i f|_{\mathbf{x}_0}+\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} h_i h_j \partial_i\partial_j f|_{\mathbf{x}_0} \\ &\quad +\frac{1}{3!}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} h_i h_j h_k \partial_i\partial_j\partial_k f|_{\mathbf{x}_0}+R_3(\mathbf{x}_0,\mathbf{h}) \end{aligned} \tag{B.1}$$

This can be inductively generalized to arbitrary order (Mardsen and Tromba, 2003, p.199). Following the same pattern as the first few terms, the sum of all m$^{\text{th}}$-order terms in the power series is:

$$\frac{1}{m!}\sum_{i_1=1}^{n}\sum_{i_2=1}^{n}\cdots\sum_{i_m=1}^{n}\left(\prod_{p=1}^{m} h_{i_p}\right)\left(\prod_{p=1}^{m}\partial_{i_p}\right) f|_{\mathbf{x}_0} \tag{B.2}$$

Formally, this means that each of the m$^{\text{th}}$-order terms in the Taylor expansion has a set of indices i_p for $1 \leq p \leq m$ corresponding to a product of all h_{i_p}, and a sequential application of partial derivatives ∂_{i_p} to f, resulting in a total product $h_1^{E_1} h_2^{E_2} \cdots h_n^{E_n}$ and mixed partial derivative $\partial_1^{E_1}\partial_2^{E_2}\cdots\partial_n^{E_n} f$, after commuting within the product of all h_{i_p} and applying the Theorem of Mixed Partial Derivatives (Mardsen and Tromba, 2003, p.183). By assuming that this can be done to each term in the sum of terms of any order (i.e. of the form of Equation B.2), the Taylor series can be expressed completely in terms of the components of a variable $\mathbf{x}$ instead of the displacement $\mathbf{h}$ from $\mathbf{x}_0$ by use of an object that is defined to be the coefficient of the power series term in which each component i of $\mathbf{x}$, denoted x_i, is raised to the power e_i. In other words, the variable $\mathbf{h}$ may be phased out in favor of making the function a polynomial of the components of $\mathbf{x} \equiv \mathbf{x}_0 + \mathbf{h}$, by means of computing the coefficients of the polynomial that would result if every instance of $h_i^{E_i}$ were replaced with $(x_i - x_{0,i})^{E_i}$ and expanded using the Binomial Theorem. If the coefficient tensor (denoted $C_{\{e_i\}}$) can be computed, it may then be used to construct the m$^{\text{th}}$-order Taylor series as follows:

$$f(\mathbf{x}) \approx \sum_{e_1=0}^{m}\sum_{e_2=0}^{(m-e_1)}\sum_{e_3=0}^{(m-e_1-e_2)}\cdots\sum_{e_n=0}^{\left(m-\sum_{i=1}^{n-1} e_i\right)}\left[\left(\prod_{i=1}^{n} x_i^{e_i}\right) C_{\{e_i\}}\right] \tag{B.3}$$

In this expression, each combination of powers (denoted as the set $\{e_i\}$) of variables x_i will only need to appear once in the resulting power series of $x_1, x_2 ... x_n$ etc., making various computational tasks more straightforward; coefficients in the series can be selected by what power of any given variable their respective term contains. Derivation of this quantity will require taking into account the following changes that occur:

1. The coefficient of a product of $x_i^{e_i}$ (for i ranging from 1 to n) that resulted from replacing each instance of $h_i^{E_i}$ in a term of the series with $(x_i - x_{0,i})^{E_i}$ and then expanding (where $x_{0,i}$ denotes component i of $\mathbf{x}_0$, the set of all E_i denotes the powers of h_i in the original product(s), and the set of all e_i denotes a particular combination of powers of x_i).

2. Each such coefficient is repeated a number of times equal to the number of terms in the Taylor series containing the particular combination of powers $\{E_i\}$ of variables $\{h_i\}$.

3. The factor coming from Equation B.2: the mixed partial derivative $\partial_1^{E_1}\partial_2^{E_2}\cdots\partial_n^{E_n} f$, divided by the factorial of $\sum_{i=1}^{n} E_i$.

4. For any combination of powers $\{e_i\}$ of variables $\{x_i\}$, there is at least one possible combination of powers $\{E_i\}$ of variables $\{h_i\}$ that produces a term with powers $\{e_i\}$. The total coefficient resulting from the previous three changes for a given set of $\{E_i\}$ adds together with coefficients resulting from each other such set.

B.2.1 Binomial expansion

Based on the Binomial Theorem (Courant and John, 1999, p.59), each term in the expansion of a product of all $(x_i - x_{0,i})^{E_i}$, having a variable x_i raised to power e_i, is of the form:

$$\binom{E_i}{e_i} (-x_{0,i})^{E_i - e_i}\, x_i^{e_i}$$

Hence, the unique term coming from the expansion of $\prod_{i=1}^{n} h_i^{E_i}$ containing the desired powers is simply the product over all of these;

$$\prod_{i=1}^{n} \binom{E_i}{e_i} (-x_{0,i})^{E_i - e_i}\, x_i^{e_i}$$

Neglecting the number of terms in the larger Taylor expansion having similar powers E_i of h_i, and the factor of the partial derivatives of f and $\frac{1}{m!}$ in the original m$^{\text{th}}$-order sum, each term containing a product of all $x_i^{e_i}$ that originated from a term containing a product of $h_i^{E_i}$ (for given sets of e_i and E_i) will have a coefficient equal to:

$$\prod_{i=1}^{n} \binom{E_i}{e_i} (-x_{0,i})^{E_i - e_i} \tag{B.4}$$

B.2.2 Multiplicity of Taylor series terms with a given set of powers

Utilizing the Theorem of Equality of Mixed Partial Derivatives (Mardsen and Tromba, 2003, p.183) and commuting the product of all h_{i_p}, there will be many terms that are equal to each other, i.e. having similar powers $\{E_i\}$ of $\{h_i\}$ and partial derivatives of f. To elaborate, a term in the original Taylor series should be equal to another term if each index (an integer ranging from 1 to n) shows up the same number of times in both terms; the terms only differ by the order in which each index shows up, but not the number of times that they show up. Because of this, the number of terms in the original series that result in each h_i having power E_i must be determined and multiplied into the total coefficient. The process of determining the number of all terms containing a similar set of E_i can be likened to the "ball and box" problem presented in Boas (1983, p.702); the boxes are in this case each index, and each distinguishable ball represents an instance that a particular index is included. The balls must be distinguishable because in this case they represent the placement of each index, from first to m$^{\text{th}}$, in each term of a sum following the form of Equation B.2.

Again using the notation $\binom{N}{M}$ for "N choose M", the number of combinations of ways in which all variables h_i could each show up E_i times in an m$^{\text{th}}$-order term is:

$$\binom{m}{E_1} \cdot \binom{m - E_1}{E_2} \cdot \binom{m - E_1 - E_2}{E_3} \cdots \binom{m - \sum_{i=1}^{n-1} E_i}{E_n}$$

which may also be written:

$$\frac{\prod_{i=1}^{n} \left(m - \sum_{j=1}^{i-1} E_j\right)!}{\prod_{i=1}^{n} \left[(E_i)! \left(m - \sum_{j=1}^{i} E_j\right)!\right]}$$

The product in the numerator and the product in the denominator will telescope down to $i = 2$, leaving:

$$\frac{m!}{\prod_{i=1}^{n} (E_i)!} \tag{B.5}$$

B.2.3 Constructing the sum

The next step will be to constrain the domain of E_i to include only the appropriate values, multiply by the factor of partial derivatives of f and $\frac{1}{m!}$, and sum over the domain. First, it is noted that, in an m$^{\text{th}}$-degree Taylor expansion, the sum of all values E_i must not exceed m. Also, the application of partial derivatives may be easily expressed in terms of sets of E_i by re-arranging them, using again the equality of mixed partial derivatives. Then, considering how the minimum possible value of E_i is e_i (from the definition of e_i given), this allows the sum to be constructed using Equation B.4 and Equation B.5 by replacing m in Equation B.5 with the sum of all E_i, which corresponds to a term of order less than or equal to m;

$$C_{\{e_i\}} = \sum_{E_1=e_1}^{m} \sum_{E_2=e_2}^{(m-E_1)} \cdots \sum_{E_n=e_n}^{\left(m-\sum_{i=1}^{n-1} E_i\right)} \frac{\left(\sum_{i=0}^{n} E_i\right)!}{\prod_{i=1}^{n} (E_i)!} \left[\prod_{i=1}^{n} \binom{E_i}{e_i} \left(-x_{0,i}\right)^{E_i - e_i}\right] \frac{\left(\prod_{i=1}^{n} \partial_i^{E_i}\right) f|_{\mathbf{x}_0}}{\left(\sum_{i=1}^{n} E_i\right)!}$$

This expression can be cleaned up a bit, beyond the immediately obvious canceling of $(\sum_{i=1}^{n} E_i)!$; using the usual definition of $\binom{a}{b}$ as "a choose b", the numerator of the product of all $\binom{E_i}{e_i}$ coming from Equation B.4 should contain a product of all $(E_i)!$ that cancels with the same product in the denominator of the factor coming from Equation B.5, but also a product of all $(e_i)!$ coming from the denominator that will be the same for each term in the multi-sum, and can thus can be factored out. After these operations, the expression for $C_{\{e_i\}}$ can be more simply written as:

$$C_{\{e_i\}} = \prod_{i=1}^{n} \frac{1}{(e_i)!} \cdot \left(\sum_{E_1=e_1}^{m} \sum_{E_2=e_2}^{(m-E_1)} \cdots \sum_{E_n=e_n}^{\left(m-\sum_{i=1}^{n-1} E_i\right)} \left[\prod_{i=1}^{n} \frac{\left(-x_{0,i}\right)^{E_i - e_i}}{(E_i - e_i)!}\right] \cdot \left[\prod_{i=1}^{n} \partial_i^{E_i}\right] f|_{\mathbf{x}_0} \right) \tag{B.6}$$

B.2.4 Implementation

The coefficients of an n-dimensional, m$^{\text{th}}$-order Taylor series can thus be stored in an array of size $m \times m \times m ...$ (etc.; n dimensions). This will inevitably result in wasted memory; slots in $C_{\{e_i\}}$ such that $\sum_{i=1}^{n} e_i > m$ will not be used, because arrays in most programming languages can only be rectangular. Additionally, there is often a limit to the number of dimensions of arrays that can be declared. In the case of IDL, without the use of NetCDF files, seven dimensions is the limit (Bowman, 2005, p.24). For the purposes of this experiment ($n = 3$ and $m \leq 3$) it will not be very wasteful or complicated to implement, and it does not require more dimensions in any array than the limitation of the programming language. For instance, the polynomial functions that will be generated by this method, for order m, take the simple form of three nested "for" loops;

$$f_c(\rho; \mu, \psi) = \sum_{i=0}^{m} \sum_{j=0}^{(m-i)} \sum_{k=0}^{(m-i-j)} \rho^i \mu^j \psi^k C_{ijk} \tag{B.7}$$

The m$^{\text{th}}$-order coefficient array, denoted C_{ijk}, is computed beforehand as follows:

$$C_{ijk} = \frac{1}{i!j!k!} \sum_{I=i}^{m} \sum_{J=j}^{(m-I)} \sum_{K=k}^{(m-I-J)} \frac{(-\rho_0)^{I-i}(-\mu_0)^{J-j}(-\psi_0)^{K-k}}{(I-i)!(J-j)!(K-k)!} \cdot \frac{\partial^{(I+J+K)} P'}{\partial \rho^I \partial \mu^J \partial \psi^K}|_{(\rho_0, \mu_0, \psi_0)} \tag{B.8}$$

The motivation for using this method is to simplify a few necessary computational tasks, discussed in Section B.3, but also to give the option of choosing any arbitrary point in D as the starting point (denoted $\mathbf{x}_0$ in Equation B.1) for the Taylor approximation. Having this as an option allows something other than $(0, 0, 0)$ to be used, which is important; since $\rho = 0$, $\mu = 0$ and $\psi = 0$ corresponds to a point of zero phase angle, there may be potentially anomalous function behavior near those values due to the opposition effect (Hapke, 1993) that would best be avoided, since that could interfere with proper calculation of partial derivatives.

B.3 Bivariate nonuniform sampling with a polynomial comparison function

The generation of trial points using a constant is fairly simple, as illustrated by Figure 1(a); the generator only needs to uniformly sample three random numbers R_1, R_2 and R_3 in the interval $[0, 1]$ and then return R_1 as the value of μ, $2\pi R_2$ as the value of ψ, and $Max(P')R_3$ as the trial gauge (Hörman *et al.*, 2003, p.17). However, for orders 1 and higher, the method described in Section 3.3 will be used. As mentioned, this requires the analytical, closed-form expression of the comparison function's indefinite integral with respect to each variable to be known and well-defined. Hence, using a power series simplifies this task greatly, especially if the coefficients are organized into an array as they are in Equation B.7. The first step of the trial point generation process requires the volume contained under the comparison function to be known. Integrating Equation B.7 with respect to μ and ψ over the domain D and factoring out powers of ρ gives the analytical expression for this volume;

$$\mathscr{V}(\rho) = \sum_{i=0}^{m} \rho^i \sum_{j=0}^{(m-i)} \sum_{k=0}^{(m-i-j)} \frac{C_{ijk}\,(2\pi)^{k+1}}{(j+1)\,(k+1)} \tag{B.9}$$

Declaring this next quantity beforehand will make execution of $\mathscr{V}(\rho)$ more efficient, as it eliminates redundant computation;

$$\overline{\mathscr{V}}_i \equiv \sum_{j=0}^{(m-i)} \sum_{k=0}^{(m-i-j)} \frac{C_{ijk}\,(2\pi)^{k+1}}{(j+1)\,(k+1)}$$

Equation B.9 may then be computed as:

$$\mathscr{V}(\rho) = \sum_{i=0}^{m} \overline{\mathscr{V}}_i \rho^i$$

B.3.1 Generating ψ with the marginal distribution

In order to generate the first of the two deviates (which, for more efficient computation, is chosen to be ψ), the process must use the marginal distribution function:

$$f_{c,\mu}(\rho;\psi) \equiv \int_0^1 f_c(\rho;\mu,\psi)d\mu$$

Performing this integral yields:

$$f_{c,\mu}(\rho;\psi) = \sum_{i=0}^{m} \sum_{j=0}^{(m-i)} \sum_{k=0}^{(m-i-j)} \frac{C_{ijk}\rho^i\psi^k}{(j+1)}$$

Next, to use the transformation law requires the analytical expression of the indefinite integral with respect to ψ; the value of ψ generated will be, according to the transformation law, the solution to Equation 3.1;

$$R_1\mathscr{V}(\rho) = \int_0^{\psi} f_{c,\mu}(\rho;\psi')d\psi'$$

(R_1 is a uniformly-sampled random deviate between 0 and 1). After performing this integral, the equation may be re-written:

$$R_1\mathscr{V}(\rho) - \sum_{i=0}^{m} \sum_{j=0}^{(m-i)} \sum_{k=0}^{(m-i-j)} \frac{C_{ijk}\;\rho^i\psi^{k+1}}{(j+1)(k+1)} = 0 \tag{B.10}$$

Considering how ψ must lie in the interval $[0,2\pi]$, the value of ψ can be computed as the lowest positive real root of this polynomial, and hence, to sample ψ, the generator will need to invoke a root-finding procedure

(`POLYSOLVE`, see Section C.1.3), and then search the array of roots returned by it for the lowest positive real one. To be able to use `POLYSOLVE`, the trial point generator must first compute the coefficients of this polynomial (denoted $C_k^\psi(\rho)$ here). First, however, a constant array that eliminates all redundant computation needs to be declared, similar to $\overline{\mathscr{V}}_i$ for the case of $\mathscr{V}(\rho)$ (Equation B.9). Expressions for both the coefficients and the preceding array of constants can be achieved by first re-arranging the sums and factoring;

$$R_1\mathscr{V}(\rho) - \sum_{k=0}^{m} \psi^{k+1} \sum_{i=0}^{(m-k)} \rho^i \sum_{j=0}^{(m-k-i)} \frac{C_{ijk}}{(k+1)(j+1)} = 0$$

The array of constants can hence be declared:

$$\overline{C}_{ik}^{\psi} \equiv \frac{-1}{k+1} \sum_{j=0}^{(m-i-k)} \frac{C_{ijk}}{j+1}$$

This leaves:

$$R_1\mathscr{V}(\rho) + \sum_{k=0}^{m} \psi^{k+1} \sum_{i=0}^{(m-k)} \overline{C}_{ik}^{\psi} \rho^i = 0$$

Next, the coefficient array can be declared as a function of ρ by displacing all instances of index k by -1 and letting k run from 0 to $m+1$ instead;

$$C_k^\psi(\rho) \equiv \begin{cases} R_1\mathscr{V}(\rho) & \text{if } k = 0; \\ \sum_{i=0}^{(m-k+1)} \overline{C}_{i(k-1)}^{\psi} \rho^i & \text{if } k > 0. \end{cases}$$

Using this formulation, Equation B.10 is:

$$\sum_{k=0}^{m+1} C_k^\psi(\rho) \psi^k = 0$$

Hence, the array returned by the function $C_k^\psi(\rho)$ (`MARDISTCOEFF`, see Section C.1.3) may be tidily passed to `POLYSOLVE` in order to obtain a value for ψ.

B.3.2 Generating μ with the conditional distribution

At this stage, the trial point generator is ready to sample μ. Given the values of ρ and ψ now available and fixed, the conditional distribution of μ, after the pattern of Equation B.7, is:

$$f_c(\rho, \psi; \mu) = \sum_{i=0}^{m} \sum_{j=0}^{(m-i)} \sum_{k=0}^{(m-i-j)} \rho^i \mu^j \psi^k C_{ijk}$$

Then, the same method for generating ψ can be used to generate μ; the value of μ is the solution to Equation 3.2 (using another uniformly-sampled random deviate, R_2):

$$R_2 \int_0^1 f_c(\rho, \psi; \mu) d\mu = \int_0^\mu f_c(\rho, \psi; \mu') d\mu' \tag{B.11}$$

The definite integral on the left-hand side, denoted $\mathscr{A}(\rho, \psi)$, can be computed as:

$$\mathscr{A}(\rho, \psi) = \sum_{i=0}^{m} \sum_{j=0}^{m-i} \sum_{k=0}^{m-i-j} \frac{C_{ijk}\ \rho^i \psi^k}{(j+1)}$$

By re-arranging the sums, the factors that cannot be predetermined may be factored out, making the constant pre-declared array more obvious;

$$\begin{aligned}
\mathscr{A}(\rho, \psi) &= \sum_{i=0}^{m} \sum_{k=0}^{m-i} \rho^i \psi^k \sum_{j=0}^{m-i-k} \frac{C_{ijk}}{(j+1)}, \\
\overline{\mathscr{A}}_{ik} &\equiv \sum_{j=0}^{m-i-k} \frac{C_{ijk}}{(j+1)}, \\
\Rightarrow \mathscr{A}(\rho, \psi) &= \sum_{i=0}^{m} \sum_{k=0}^{m-i} \overline{\mathscr{A}}_{ik} \rho^i \psi^k
\end{aligned}$$

Next, the indefinite integral on the right-hand side of Equation B.11 is given by:

$$\int_0^\mu f_c(\rho, \psi; \mu') d\mu' = \sum_{i=0}^{m} \sum_{j=0}^{m-i} \sum_{k=0}^{m-i-j} \frac{C_{ijk}\ \rho^i \mu^{j+1} \psi^k}{(j+1)}$$

The only redundant operation that could be eliminated in this case is division by $j+1$, so there is almost no need to define an array of constants beforehand for sampling μ. Finally, the function for determining the coefficient array, denoted $C_j^\mu(\rho, \psi)$, is then (with j running from 0 to $m+1$):

$$C_j^\mu(\rho, \psi) \equiv \begin{cases} R_2 \mathscr{A}(\rho, \psi) & \text{if } k = 0; \\ -\frac{1}{j} \sum_{i=0}^{(m-j+1)} \sum_{k=0}^{(m-i-j+1)} C_{i(j-1)k}\ \rho^i \psi^k & \text{if } k > 0. \end{cases}$$

Equation B.11 is then:

$$\sum_{j=0}^{m+1} C_j^\mu(\rho, \psi) \mu^j = 0$$

Finally, the array returned by $C_j^\mu(\rho, \psi)$ (`CONDISTCOEFF`, see Section C.1.3) can be passed to `POLYSOLVE` to obtain μ, and the trial gauge is $R_3 f_c(\rho; \mu, \psi)$, where R_3 is a third uniformly-sampled random deviate in $[0, 1]$.

C Source Code

C.1 Functions

C.1.1 Evaluation and re-expression

```
FUNCTION DIFFC, X
COMMON TRIGEN, setup_out, p_m, seed, index, mcs, constswitch
; this function is passed to amoeba, so a few constraints
; must be imposed to keep it within D'

pi=3.14159265358979D0
bad=0.0D0
fence=1.0D1

FOR i=0, 2 DO BEGIN
   IF (X[i] LT 0.0D0) THEN BEGIN
      X[i]=0
      bad=bad+fence
   ENDIF
ENDFOR

IF (X[2] GT 2.0D0*pi) THEN BEGIN
   X[2]=2.0D0*pi
   bad=bad+fence
ENDIF

FOR i=0,1 DO BEGIN
   IF (X[i] GT 1.0D0) THEN BEGIN
      X[i]=1
      bad=bad+fence
   ENDIF
ENDFOR

ret=bad+genpoly(0,X)-pfunc(X)
RETURN, ret
END

FUNCTION DIFFS, X
COMMON TRIGEN, setup_out, p_m, seed, index, mcs, constswitch
COMMON SCALARS, pi, df_param, exp_param
; this function is passed to amoeba, so a few constraints must
; be imposed to keep it within D'

pi=3.14159265358979D0
bad=0.0D0
fence=1.0D1

FOR i=0,2 DO BEGIN
```

```
    IF (X[i] LT 0) THEN BEGIN
        X[i]=0
        bad=bad+fence
    ENDIF
ENDFOR

IF (X[2] GT 2*pi) THEN BEGIN
    X[2]=2*pi
    bad=bad+fence
ENDIF

FOR i=0,1 DO BEGIN
    IF (X[i] GT 1) THEN BEGIN
        X[i]=1
        bad=bad+fence
    ENDIF
ENDFOR

ret=bad+pfunc(X)-genpoly(1,X)
RETURN, ret
END

FUNCTION EMNG, mu
COMMON SCALARS, pi, df_param, exp_param
e=0.0D0
IF (mu GT 1.0D0) THEN e=0.0D0 ELSE IF (mu LT -1.0D0) THEN e=pi ELSE e=acos(mu)
RETURN, e
END

FUNCTION FC_SCALAR_EVAL, rho, mu, psi
ret=fc([rho,mu,psi])
RETURN, ret
END

FUNCTION FS_SCALAR_EVAL, rho, mu, psi
ret=fs([rho, mu, psi])
RETURN, ret
END

FUNCTION INNG, rho
COMMON SCALARS, pi, df_param, exp_param
i=0.0D0
IF (rho GT 1.0D0) THEN i=pi/2.0D0 ELSE IF (rho LT 0.0D0) THEN i=0.0D0 ELSE i=asin(sqrt(rho))
RETURN, i
END

FUNCTION LAT, i,e,psi
g=phg(i,e,psi)
latitude=0.0D0
```

```
IF ((e EQ 0) OR (i EQ 0)) THEN latitude=0.0D0 ELSE BEGIN
IF (i EQ e) THEN latitude=acos(cos(i)/cos(g/2)) ELSE BEGIN
IF (sin(psi) EQ 0) THEN L=0.0D0 ELSE BEGIN
; i /= 0 and e/=0 and i/=e and psi/=0, psi/=pi
a=sin(i+e)^2
b=-cos(psi/2.0D0)^2*sin(2.0D0*e)^2*sin(2.0D0*i)^2
c=(a+b)/(a+b+sin(i)^2*sin(e)^2*sin(psi)^2)
IF (c GE 1) THEN c=1
IF (c LE 0) THEN c=0
latitude=acos(sqrt(c))
ENDELSE
ENDELSE
ENDELSE

RETURN, latitude
END

FUNCTION LON, i,e,psi
g=phg(i,e,psi)
L=0.0D0

IF (e EQ 0) THEN L=0.0D0 ELSE BEGIN
IF (i EQ 0) THEN L=-e ELSE BEGIN
IF (i EQ e) THEN L=-g/2.0D0 ELSE BEGIN
IF (sin(psi) EQ 0) THEN L=e*sign(cos(psi)) ELSE BEGIN
; i /= 0 and e/=0 and i/=e and psi/=0, psi/=pi
signL=sign(cos(psi)*(cos(e)-cos(psi)*cos(i)*sin(e))+(cos(e)-1)*(sin(psi)/sin(i))^2)
coslat = cos(lat(i,e,psi))
IF (coslat NE 0) THEN L=signL*acos(cos(e)/coslat) ELSE L=0
ENDELSE
ENDELSE
ENDELSE
ENDELSE

RETURN, L
END

FUNCTION MULIMIT, rho
RETURN, [0.0D0, 1.0D0]
END

FUNCTION NEGPFUNC, X
COMMON SCALARS, pi, df_param, exp_param
pi=3.14159265358979D0

; This variable will be increased if the variable is out of bounds in
; order to discourage AMOEBA from searching outside the limit of
; variables that is desired
bad=0.0D0
```

```
fence=1.0D1

FOR i=0,2 DO BEGIN
    IF (X[i] LT 0) THEN BEGIN
        X[i]=0
        bad=bad+fence
    ENDIF
ENDFOR

IF (X[2] GT 2*pi) THEN BEGIN
    X[2]=2*pi
    bad=bad+fence
ENDIF

FOR i=0,1 DO BEGIN
    IF (X[i] GT 1) THEN BEGIN
        X[i]=1
        bad=bad+fence
    ENDIF
ENDFOR
ret=bad-pfunc(X)
RETURN, ret
END

FUNCTION PFUNC, X
COMMON SCALARS, pi, minnaert, steepness
g=acos(sqrt(x[0])*sqrt(1-X[1]^2)*cos(X[2])+sqrt(1-x[0])*X[1])
peval=exp(-steepness*g)*X[1]^(minnaert-1)*sqrt(1-x[0])^(minnaert)
; Disk function of Akimov
; peval=exp(-exp_param*g)*cos(g/2)*(cos(abs(lo)-g/2)^(df_param+1)
; -sin(g/2)*cos(la)^df_param)/(cos(lo)*(1-sin(g/2))^(df_param+1))
RETURN, peval
END

FUNCTION PFUNC_SCALAR_EVAL, rho, mu, psi
X=[rho,mu,psi]
ret=pfunc(X)
RETURN, ret
END

FUNCTION PHG, i,e,psi
g=acos(cos(psi)*sin(i)*sin(e)+cos(i)*cos(e))
RETURN, g
END

FUNCTION PSILIMIT, rho, mu
COMMON SCALARS, pi, df_param, exp_param
RETURN, [0.0D0, 2.0D0*pi]
END
```

```
FUNCTION SIGN, x
s=1
IF (x LT 0) THEN s=-1
RETURN, s
END
```

C.1.2 Setup and approximation

```
FUNCTION DUMMYEXECP, rho
COMMON SCALARS, pi, df_param, exp_param
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
mu=randomu(seed, /DOUBLE)
psi=randomu(seed, /DOUBLE)*2*pi
accept=(1 GT pfunc([rho,mu,psi]))
RETURN, 0
END

FUNCTION GENPOLY, l,X
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
m=mcs[l]
ret=const[index].generator.corr[l,m]

FOR i=0, m+1 DO BEGIN
    FOR j=0, m+1-i DO BEGIN
        FOR k=0, m+1-i-j DO BEGIN
            ret=ret+const[index].generator.c[m,i,j,k]*X[0]^i*X[1]^j*X[2]^k
        ENDFOR
    ENDFOR
ENDFOR
RETURN, ret
END

FUNCTION MEAN, x
length=(size(x))[1]
moment=0.0D0
FOR i=0, length-1 DO BEGIN
    moment=moment+x[i]
ENDFOR
moment=moment/length
RETURN, moment
END

FUNCTION MKCOEFF, m,i,j,k,X0
entry=0.0D0
rhom=X0[0]
mum=X0[1]
psim=X0[2]
FOR ii=i, m DO BEGIN
FOR jj=j, m-ii DO BEGIN
FOR kk=k, m-ii-jj DO BEGIN
entry=entry+((-rhom)^(ii-i)*(-mum)^(jj-j)*(-psim)^(kk-k))/(factorial(ii-i)*factorial(jj-j)*factorial(kk-k))*partia
ENDFOR
ENDFOR
ENDFOR
RETURN, entry/(factorial(i)*factorial(j)*factorial(k))
END

FUNCTION PARTIALD, i,j,k,rho,mu,psi
IF (i NE 0) THEN ret=ridders(i-1,j,k,0,rho,mu,psi)
ELSE IF (j NE 0) THEN ret=ridders(0,j-1,k,1,rho,mu,psi)
ELSE IF (k NE 0) THEN ret=ridders(0,0,k-1,2,rho,mu,psi)
ELSE ret=pfunc([rho,mu,psi])
RETURN, ret
END

FUNCTION PARTIALDISP, i,j,k,l,h,rho,mu,psi
ret=0.0D0
IF (l EQ 0) THEN ret=partiald(i,j,k,rho+h,mu,psi)
ELSE IF (l EQ 1) THEN ret=partiald(i,j,k,rho,mu+h,psi+h)
ELSE IF (l EQ 2) THEN ret=partiald(i,j,k,rho,mu,psi+h)
RETURN, ret
END

FUNCTION RIDDERS, i,j,k,l,rho,mu,psi
; based on Press et al (2007) pp.231-232

err=1.0D-3
nn=10
a=dblarr(nn,nn)
h=1.0D-3
safe=2.0D0
c1=1.4D0
c2=c1*c1
ret=0.0D0
fac=c2

a[0,0]=(partialdisp(i,j,k,l,h,rho,mu,psi)-partialdisp(i,j,k,l,-h,rho,mu,psi))/(2.0D0*h)
FOR n=1, nn-1 DO BEGIN
   h=h/c1
   a[0,n]=(partialdisp(i,j,k,l,h,rho,mu,psi)-partialdisp(i,j,k,l,-h,rho,mu,psi))/(2.0D0*h)
   fac=c2
   FOR m=1, n DO BEGIN
      a[m,n]=(a[m-1,n]*fac-a[m-1,n-1])/(fac-1.0D0)
      fac=c2*fac
      errt=max([abs(a[m,n]-a[m-1,n]),abs(a[m,n]-a[m-1,n-1])])
      IF (errt LE err) THEN BEGIN
         err=errt
         ret=a[m,n]
```

```
        ENDIF
    ENDFOR
    IF (abs(a[n,n]-a[n-1,n-1]) GE safe*err) THEN BREAK
ENDFOR
RETURN, ret
END

FUNCTION TAYLOR, m, X0
coeff=dblarr(m+1,m+1,m+1)
FOR i=0, m DO BEGIN
FOR j=0, m-i DO BEGIN
FOR k=0, m-i-j DO BEGIN
coeff[i,j,k]=mkcoeff(m,i,j,k,X0)
ENDFOR
ENDFOR
ENDFOR
RETURN, coeff
END

FUNCTION VARIANCE, x
moment=mean(x)
length=(size(x))[1]
deviation=0.0D0
FOR i=0, length-1 DO deviation=deviation+(x[i]-moment)^2
deviation=sqrt(deviation/(length-1.0D0))
RETURN, deviation
END
```

C.1.3 Trial point generation and rejection sampling

```
FUNCTION AREA, rho, psi
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
m=mcs[0]
area=0.0D0
FOR i=0, m DO FOR k=0, m-i DO area=area+const[index].generator.margconst[m,i,k]*rho^i*psi^k
RETURN, area
END

FUNCTION CONDISTCOEFF, rho, psi
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
m=mcs[0]
coeff=dblarr(m+2)
coeff[0]=RANDOMU(seed,/DOUBLE)*area(rho,psi)
coeff[1]=-const[index].generator.corr[0,m]*rho*psi
FOR j=1, m+1 DO FOR i=0, m-j+1 DO FOR k=0,m-i-j+1 DO coeff[j]=coeff[j]-1.0D0/j*const[index].generator.c[m,i,j-1,k
RETURN, coeff
END

FUNCTION CONSTRIGEN, rho
COMMON SCALARS, pi, df_param, exp_param
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
 trial=dblarr(3)
 trial[0]=randomu(seed,/DOUBLE)
 trial[1]=randomu(seed,/DOUBLE)*2*pi
 trial[2]=randomu(seed,/DOUBLE)*p_m
 accept=(trial[2] LE pfunc([rho,trial[0],trial[1]]))
 trialpt_out=CREATE_STRUCT("values", trial, "accept", accept)
RETURN, trialpt_out
END

FUNCTION FC, X
 ret=genpoly(0,X)
RETURN, ret
END

FUNCTION FS, X
 ret=genpoly(1,X)
 IF (ret LT 0) THEN ret=0
RETURN, ret
END

FUNCTION MARDISTCOEFF, rho
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
m=mcs[0]
 coeff=dblarr(m+2)
 coeff[0]=RANDOMU(seed,/DOUBLE)*volume(rho)
FOR k=1, m+1 DO FOR i=0, m-k+1 DO coeff[k]=coeff[k]+const[index].generator.margconst[m,i,k-1]*rho^i
RETURN, coeff
END

FUNCTION POLYSOLVE, a
 ret=0
 l=(size(a))[1]

 IF (l EQ 3) THEN ret=quadratic(a)
 IF (l EQ 4) THEN ret=cubic(a)
 IF (l EQ 5) THEN ret=quartic(a)

RETURN, ret
END

FUNCTION POLYTRIGEN, rho
COMMON SCALARS, pi, dh_param, exp_param
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch



 i=0
mu=0
 psi=0
 found=0
```

```
roots=0
accept=0
tcoeff=0
rootsize=0
trialpt_out=0

tcoeff=complex(mardistcoeff(rho))
roots=polysolve(tcoeff)
rootsize=(size(roots))[1]

FOR i=0, rootsize-1 DO BEGIN
    re=real_part(roots[i])
    im=imaginary(roots[i])
    IF (((abs(im) LT 1.0D-10) AND (re GE 0)) AND (re LE pi*2+1.0D-13)) THEN BEGIN
        psi=re
        found=1
        BREAK
    ENDIF
ENDFOR



IF (found NE 0) THEN BEGIN
    tcoeff=complex(condistcoeff(rho,psi))
    roots=polysolve(tcoeff)
    rootsize=(size(roots))[1]
    FOR i=0, rootsize-1 DO BEGIN
        found=0
        re=real_part(roots[i])
        im=imaginary(roots[i])
        IF (((abs(im) LT 1.0D-10) AND (re GE 0)) AND (re LE 1)) THEN BEGIN
            mu=re
            found=1
            BREAK
        ENDIF
    ENDFOR
    gauge=fc([rho,mu,psi])*RANDOMU(seed, /DOUBLE)
    trial=[mu,psi,gauge]
    accept=(trial[2] LE fs([rho,mu,psi]))
    trialpt_out=CREATE_STRUCT("values", trial, "accept", accept)
ENDIF ELSE IF (found EQ 0) THEN trialpt_out=constrigen(rho)

RETURN, trialpt_out
END

FUNCTION REJECTION, mu0
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
rho=1-mu0^2
trialpt=CREATE_STRUCT("values", [0.0D0,0.0D0,p_m+10],"accept", 0)
accept=0
```

```
WHILE (trialpt.accept NE 1) DO BEGIN
   IF constswitch THEN trialpt=constrigen(rho) ELSE trialpt=polytrigen(rho)
   IF ((trialpt.accept NE 1) AND (constswitch NE 1)) THEN BEGIN
      IF (trialpt.values[2] LE pfunc([rho,trialpt.values[0],trialpt.values[1]])) THEN BREAK
   ENDIF
ENDWHILE
RETURN, trialpt.values
END

FUNCTION VOLUME, rho
COMMON TRIGEN, const, p_m, seed, index, mcs, constswitch
vol=0.0D0
FOR i=0, mcs[0] DO vol=vol+rho^i*const[index].generator.volconst[mcs[0],i]
RETURN, vol
END
```

C.1.4 Solving the quartic, cubic and quadratic equations

```
FUNCTION CUBIC, ain
; the input array is complex
; a[3] :: 1
; a[2] :: a2
; a[1] :: a1
; a[0] :: a0
ret=complex([0,0,0])
IF (real_part(ain[3]) NE 0) THEN BEGIN
   a = ain/ain[3]
   q=a[1]/3.0D0-(a[2]^2.0D0)/9.0D0
   r=(a[1]*a[2]-3*a[0])/6.0D0-a[2]^3/2.7D1
   s1=(r+sqrt(q^3+r^2))^(1.0D0/3.0D0)
   s2=(r-sqrt(q^3+r^2))^(1.0D0/3.0D0)
   s3=s1-s2
   s4=s1+s2
   scomp= complex(-imaginary(s3),real_part(s3))*sqrt(3.0D0)/2.0D0

   z1=s4-a[2]/3.0D0
   z2=-s4/2.0D0-a[2]/3.0D0 + scomp
   z3=-s4/2.0D0-a[2]/3.0D0 - scomp

   ret=[z1,z2,z3]
ENDIF ELSE BEGIN
   print, "cubic error: expected cubic equation, got quadratic"
   ret=quadratic([ain[0],ain[1],ain[2]])
ENDELSE
RETURN, ret
END

FUNCTION QUADRATIC, a
```

```
; the input array is complex
; a[2] is a
; a[1] is b
; a[0] is c
ret=0

IF (real_part(a[2]) NE 0) THEN BEGIN
   s=sqrt(a[1]^2.0D0-4.0D0*a[2]*a[0])/(2.0D0*a[2])
   r=-a[1]/(2.0D0*a[2])
   ret=[r+s,r-s]
ENDIF ELSE IF (a[1] NE 0) THEN BEGIN
   ret=[a[0]/a[1]]
ENDIF ELSE BEGIN
   print, "quadratic error: coefficients are zero"
   ret=[complex(0.0D0)]
ENDELSE
RETURN, ret
END

FUNCTION QUARTIC, ain
; the input array is complex
; a[4] :: 1
; a[3] :: a3
; a[2] :: a2
; a[1] :: a1
; a[0] :: a0
z=0
IF (real_part(ain[4]) NE 0) THEN BEGIN
   a=ain/ain[4]
   z=complexarr(4)
   u=cubic([-(a[1]^2.0D0+a[0]*a[3]^2.0D0-4.0D0*a[0]*a[2]), $
            a[1]*a[3]-4.0D0*a[0],-a[2],complex(1.0D0)])
   found=0
   IF (abs(imaginary(u[1])) EQ 0) THEN BEGIN
      FOR i=0,2 DO BEGIN
         atmp=[(u[i]/2.0D0)^2-a[0],a[3]^2/4.0D0+u[i]-a[2], $
               complex(1.0D0)]
         IF ((real_part(atmp[0]) GT 0) AND (real_part(atmp[1]) GT 0)) THEN BEGIN
            rtmp=quadratic([u[i]/2.0D0-sqrt(atmp[0]), $
                           a[3]/2.0D0-sqrt(atmp[1]),atmp[2]])
            z[0]=rtmp[0]
            z[1]=rtmp[1]
            rtmp=quadratic([u[i]/2.0D0+sqrt(atmp[0]), $
                           a[3]/2.0D0+sqrt(atmp[1]),atmp[2]])
            z[2]=rtmp[0]
            z[3]=rtmp[1]
            found=1
         ENDIF
      ENDFOR
```

```
    ENDIF ELSE IF (found EQ 0) THEN BEGIN
        atmp=[sqrt((u[0]/2.0D0)^2-a[0]),sqrt(a[3]^2.0D0/4.0D0+u[0] $
                                                -a[2]),complex(1.0D0)]
        rtmp=quadratic([u[0]/2.0D0-atmp[0],a[3]/2.0D0-atmp[1], atmp[2]])
        z[0]=rtmp[0]
        z[1]=rtmp[1]
        rtmp=quadratic([u[0]/2.0D0+atmp[0],a[3]/2.0D0+atmp[1],atmp[2]])
        z[2]=rtmp[0]
        z[3]=rtmp[1]
    ENDIF
ENDIF ELSE BEGIN
    print, "quartic error: expected quartic equation, got cubic"
    z=cubic(ain[0],ain[1],ain[2],ain[3])
ENDELSE
RETURN, z
END
```

C.2 The SETUP procedure:

```
PRO SETUP, XSTART



COMMON SCALARS, pi, df_param, exp_param
COMMON TRIGEN, sto, p_m, seed, index, mcs, constswitch
COMMON INT_PFUNC, ip, performance_const_mean, performance_const_error
pisec=3.14159265358979D0
print, 'Setup beginning for index:', index

c=dblarr(3,4,4,4)
corr=dblarr(2,3)
volconst=dblarr(3,4)
margconst=dblarr(3,4,4)
areaconst=dblarr(3,4,4)
sto[index].start=XSTART

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;PERFORM TAYLOR EXPANSIONS;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
FOR m=0, 2 DO BEGIN
    ctmp=taylor(m+1,XSTART)
    FOR i=0, m+1 DO BEGIN
        FOR j=0, m+1-i DO BEGIN
            FOR k=0, m+1-i-j DO BEGIN
                c[m,i,j,k]=ctmp[i,j,k]
            ENDFOR
        ENDFOR
    ENDFOR
ENDFOR
; Store the coefficients
```

```
sto[index].generator.c=c
print, 'Taylor expansions finished...'

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;SAFETY ADJUSTMENTS;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
xtmp=dblarr(3)
FOR m=0, 2 DO BEGIN
   mcs[0]=m
   mcs[1]=m
   i=0
   iter=5000
   precision=1.0D-4
   WHILE (i LT 4) DO BEGIN
      xtmp=AMOEBA(precision, FUNCTION_NAME="diffc", NMAX=iter, $
                  SCALE=[5.0D-1,5.0D-1,pi], P0=[5.0D-1,5.0D-1,pi])
      IF ((size(xtmp))[0] EQ 0) THEN BEGIN
         i++
         precision=precision*10.0D0
         iter=iter+100.0D0^i
      ENDIF ELSE i=4
   ENDWHILE
   IF ((size(xtmp))[0] EQ 0) THEN BEGIN
      print, 'Failed to find maximum of function with sufficient accuracy.'
      STOP
   ENDIF ELSE BEGIN
      corr[0,m]=-diffc(xtmp)*(1+precision*2)
   ENDELSE
   i=0
   iter=5000
   precision=1.0D-4
   WHILE (i LT 4) DO BEGIN
      xtmp=AMOEBA(precision, FUNCTION_NAME="diffs", NMAX=iter, $
                  SCALE=[5.0D-1,5.0D-1,pi], P0=[5.0D-1,5.0D-1,pi])
      IF ((size(xtmp))[0] EQ 0) THEN BEGIN
         i++
         precision=precision*10.0D0
         iter=iter+100.0D0^i
      ENDIF ELSE i=4
   ENDWHILE
   IF ((size(xtmp))[0] EQ 0) THEN BEGIN
      print, 'Failed to find maximum of function with sufficient accuracy.'
      STOP
   ENDIF ELSE BEGIN
      corr[1,m]=diffs(xtmp)*(1+precision*2)
   ENDELSE
ENDFOR
; Store the 0th-order term corrections
sto[index].generator.corr=corr
```

```
print, 'Adjustments finished...'

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;INITIALIZE ARRAY CONSTANTS;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
FOR m=0, 2 DO BEGIN
   volconst[m,0]=(c[m,0,0,0]+corr[0,m])*(2*pi)
   margconst[m,0,0]=-c[m,0,0,0]-corr[0,m]
   areaconst[m,0,0]=c[m,0,0,0]+corr[0,m]
   FOR i=0, m+1 DO BEGIN
      FOR j=0, m+1-i DO BEGIN
         FOR k=0, m+1-i-j DO BEGIN
            IF (((i NE 0) OR (j NE 0)) OR (k NE 0)) THEN BEGIN
               volconst[m,i]=volconst[m,i]+c[m,i,j,k]*(2*pi)^(k+1)/((j+1.0D0)*(k+1.0D0))
               margconst[m,i,k]=margconst[m,i,k]-c[m,i,j,k]/((k+1.0D0)*(j+1.0D0))
               areaconst[m,i,k]=areaconst[m,i,k]+c[m,i,j,k]/(j+1.0D0)
            ENDIF
         ENDFOR
      ENDFOR
   ENDFOR
ENDFOR
; Store the constant arrays
sto[index].generator.volconst=volconst
sto[index].generator.margconst=margconst
sto[index].generator.areaconst=areaconst
print, 'Array constants finished, beginning performance testing...'

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;PERFORMANCE TESTING;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
performance=dblarr(50,3,3)
ic=dblarr(3)
is=dblarr(3)
FOR mc=0, 2 DO BEGIN
   mcs[0]=mc
   ic[mc]=INT_3D("fc_scalar_eval", [0.0D0, 1.0D0], "mulimit", "psilimit",48)
ENDFOR
FOR ms=0, 2 DO BEGIN
   mcs[1]=ms
   is[ms]=INT_3D("fs_scalar_eval", [0.0D0, 1.0D0], "mulimit", "psilimit",48)
ENDFOR

FOR i=0, 49 DO BEGIN
   rho=randomu(seed, /DOUBLE)
   FOR mc=0, 2 DO BEGIN
      FOR ms=0, 2 DO BEGIN
         mcs[0]=mc
         mcs[1]=ms
```

```
            PROFILER, /SYSTEM
            trialpoint=polytrigen(rho)
            PROFILER, DATA=preops, /REPORT

            PROFILER, /SYSTEM
            p=pfunc([rho,trialpoint.values[0],trialpoint.values[1]])
            accept=(trialpoint.values[2] LE p)
            PROFILER, DATA=exeops, /REPORT

            pre_numops=(size(preops))[1]
            exe_numops=(size(exeops))[1]

            pre=0.0D0
            exe=0.0D0
            FOR j=0, pre_numops-1 DO pre=pre+preops[j].TIME
            FOR j=0, exe_numops-1 DO exe=exe+exeops[j].TIME
            performance[i,mc,ms]=ic[mc]/(2*ip)*pre+(ic[mc]-is[ms])/ip*exe
        ENDFOR
    ENDFOR
ENDFOR
print, 'Performance test finished...'

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;MEAN/VARIANCE OF PERFORMANCE;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
performance_var=DBLARR(20)
mean=DBLARR(3,3)
error=DBLARR(3,3)
FOR mc=0, 2 DO BEGIN
    FOR ms=0, 2 DO BEGIN
        FOR i=0, 19 DO performance_var[i]=performance[i,mc,ms]
        mtmp=mean(performance_var)/performance_const_mean
        mean[mc,ms]=mtmp
        ertmp=variance(performance_var)
        ; here we use propagation of error on the quantity
        ; performance(polynomial)/performance(constant)
        error[mc,ms]=sqrt((ertmp/performance_const_mean)^2+ $
                        (performance_const_error*mtmp/performance_const_mean^2)^2)
    ENDFOR
ENDFOR
; Store the performance testing results
sto[index].mean=mean
sto[index].error=error

print, 'Setup complete for index:', index

END
```

C.3 The SEARCHSETUP procedure: re-implementation of SETUP

```
PRO SEARCHSETUP, npoints, s1, s2

; This procedure will set up a rejection-with-constant trial point
; generator, and then perform a search over npoints random points in
; the domain to find if there's an approximation starting point
; and order resulting in a trial point generator more efficient than
; the uniform trial point generator


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;COMMON BLOCK DEFINITIONS;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
COMMON SCALARS, pi, param_1, param_2
COMMON TRIGEN, setup_out, p_m, seed, index, mcs, constswitch
COMMON BESTPOINTS, winners, bestindices
COMMON INT_PFUNC, ip, performance_const_mean, performance_const_error

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;SCALAR & MISC. CONSTANTS;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
mcs=lindgen(2)
pi=3.14159265358979D0
param_1=s1
param_2=s2
seed=floor(10*systime(/JULIAN)/pi)
i=0
iter=5000
precision=1.0D-6
p_m=0
WHILE (i LT 5) DO BEGIN
   xmax=AMOEBA(precision, FUNCTION_NAME="negpfunc", NMAX=iter, $
               SCALE=[5.0D-1,5.0D-1,pi], P0=[5.0D-1,5.0D-1,pi])
   IF ((size(xmax))[0] EQ 0) THEN BEGIN
      i++
      precision=precision*10.0D0
      iter=iter+10.0D0^i
   ENDIF ELSE i=5
ENDWHILE
IF ((size(xmax))[0] EQ 0) THEN BEGIN
   print, 'Failed to find maximum of function with sufficient accuracy.'
   STOP
ENDIF ELSE BEGIN
   p_m=pfunc(xmax)*(1.0D0 + precision*2)
ENDELSE
ip=INT_3D("pfunc_scalar_eval", [0.0D0,1.0D0], "mulimit", "psilimit",96)

; data: 20 random starting points, choose error or mean (2), 3
; possible orders of approxmation (in the comparison function), and 3
; possible orders of approximation (in the squeeze function)
setup_constants = CREATE_STRUCT('c', dblarr(3,4,4,4), $
                                'corr', dblarr(2,3), $
                                'volconst', dblarr(3,4), $
                                'margconst', dblarr(3,4,4), $
                                'areaconst', dblarr(3,4,4), $
                                'start', dblarr(3))
setup_out = CREATE_STRUCT('startpoint', [0.0D0,0.0D0,0.0D0], $
                          'generator', setup_constants , $
                          'mean', dblarr(3,3), 'error', dblarr(3,3))
setup_out = REPLICATE(setup_out, npoints)
print, 'Declarations complete...'

; performance testing for constant, which will be used as a standard
; against which all other generators are compared
performance_const=dblarr(50)
FOR i=0, 49 DO BEGIN
    time_const=0.0D0
    rho=RANDOMU(seed, /DOUBLE)
    PROFILER, /SYSTEM
    trialpoint=constrigen(rho)
    PROFILER, DATA=ops, /REPORT
    numops=SIZE(ops)
    FOR j=0, numops[1]-1 DO time_const=time_const+ops[j].TIME
    performance_const[i]=time_const*pi*p_m/ip
ENDFOR
performance_const_error=VARIANCE(performance_const)
performance_const_mean=MEAN(performance_const)
print, 'Performance testing for uniform complete, beginning setup at:', $
       npoints, ' random points within the sample space...'

constswitch=0
FOR i=0, npoints-1 DO BEGIN
    xrand=[RANDOMU(seed,/DOUBLE), RANDOMU(seed, /DOUBLE), $
           2*pi*RANDOMU(seed, /DOUBLE)]
    index=i
    setup, xrand
ENDFOR

print, 'Setup complete for all points, beginning search for minimum...'
; FINALLY, the moment we've been waiting for: search for the
; indices containing with the lowest four mean marginal execution times and store
; them for plotting. For the full-blown simulation, only the
; "absolute_lowest" and "absolute_lowest_index" variables are needed
; in order to set the constants correctly.

; We'll start by filling the minimum mean execution time with
; something huge;
big=1.0D9
```

```
ind=1

; Store the winners in a structure that plotout will use:
winner=CREATE_STRUCT('index', ind, 'lowest', big)
winners=REPLICATE(winner,4)

; Globally most efficient orders/mean m.m.e.t. (to be changed)
absolute_lowest_index=lonarr(2)
absolute_lowest=1.0D9

FOR i=0, npoints-1 DO BEGIN
; Start with something big, work our way down;
   low_tmp=1.0D9
   FOR mc=0, 2 DO BEGIN
      FOR ms=0, 2 DO BEGIN
; Iterate over each order of approximation, find the lowest mean
; marginal execution time
         IF (low_tmp GT setup_out[i].mean[mc,ms]) THEN BEGIN
            low_tmp=setup_out[i].mean[mc,ms]
; Hunt for the absolute most efficient
            IF (absolute_lowest GE setup_out[i].mean[mc,ms]) THEN BEGIN
               absolute_lowest=setup_out[i].mean[mc,ms]
               absolute_lowest_index=[mc,ms]
            ENDIF
         ENDIF
      ENDFOR
   ENDFOR
; If there's a new contender for lowest 4, rearrange the array
; from greatest to least mean marginal execution time, put the new low
; in, and eliminate any that it defeated
   IF (low_tmp LT winners[0].lowest) THEN BEGIN
      print, 'New member of best 4 found at index:', i
      winners[0].index=i
      winners[0].lowest=low_tmp
      FOR j=1, 3 DO BEGIN
         IF (low_tmp LT winners[j].lowest) THEN BEGIN
            winners[j-1].index=winners[j].index
            winners[j-1].lowest=winners[j].lowest
            winners[j].index=i
            winners[j].lowest=low_tmp
         ENDIF
      ENDFOR
   ENDIF
ENDFOR
bestindices=lonarr(4)
FOR i=0,3 DO bestindices[i]=winners[i].index

; For the simulation: set the index/order of approximation to the
; optimal one that was found so that the generator uses it. If there
; was not found any polynomial trial point generators more efficient
; than the uniform trial point generator, just use the uniform one.
IF (absolute_lowest GT performance_const_mean) THEN constswitch=1
index=bestindices[3]
mcs=absolute_lowest_index

print, 'Initialization routine complete. Best four setup points were at indices: ', $
       bestindices

print, 'Saving data...'
; Put data into files for plotting:
plotout, 0.5, 1



END
```

C.4 Saving unformatted text data to files for plotting

```
PRO PLOTOUT, rho, npts, csw
; This procedure just puts all the data into files, to be sorted and
; plotted later in Mathematica (which I'm much more familiar with)

COMMON TRIGEN, setup_out, p_m, seed, index, mcs, constswitch
COMMON BESTPOINTS, winners, bestindices

openw, 1, 'out1.dat'
openw, 2, 'out2.dat'
openw, 3, 'out3.dat'
openw, 4, 'out4.dat'
openw, 5, 'rejection_out.dat'
openw, 6, 'start1.dat'
openw, 7, 'start2.dat'
openw, 8, 'start3.dat'
openw, 9, 'start4.dat'
FOR I=0, 2 DO BEGIN
   FOR j=0, 2 DO BEGIN
      printf, 1, setup_out[bestindices[0]].mean[i,j], setup_out[bestindices[0]].error[i,j]
      printf, 2, setup_out[bestindices[1]].mean[i,j], setup_out[bestindices[1]].error[i,j]
      printf, 3, setup_out[bestindices[2]].mean[i,j], setup_out[bestindices[2]].error[i,j]
      printf, 4, setup_out[bestindices[3]].mean[i,j], setup_out[bestindices[3]].error[i,j]
      printf, 6, setup_out[bestindices[0]].start
      printf, 7, setup_out[bestindices[1]].start
      printf, 8, setup_out[bestindices[2]].start
      printf, 9, setup_out[bestindices[3]].start
   ENDFOR
ENDFOR

constswitch=csw
rejtmp=0
```

```
FOR I=0, npts DO BEGIN
    FOR K=0, npts DO BEGIN
        rejtmp=REJECTION(rho)
        FOR J=0,2 DO BEGIN
            printf, 5, rejtmp[j]
        ENDFOR
    ENDFOR
ENDFOR

close, 1,2,3,4,5,6,7,8,9

END
```