Mathematical Methods for Minimizing the Internal Gravitational Field of a Cylindrical Satellite

Mark Edward Rupright

Tennessee Scholars Senior Thesis

May 5, 1992

Introduction

One of the benefits of the Space Program is the opportunity for researchers to carry out their experiments in a gravity-free environment. For example, some types of crystals cannot grow uniformly unless they are in a gravity-free environment. Scientists who engage in experiments which are extremely sensitive to gravitational fields have taken advantage of the "weightless" or "zero-g" environment of outer-space. However, this environment is not truly zero-g. An entire field of physics, called microgravity, has been developed to find ways to keep negative effects such as tidal forces, electric fields, etc. From spoiling an otherwise zero-g environment necessary for sensitive experiments. One of the causes of gravitational fields inside a satellite in free-fall is the mass of the satellite itself. If the satellite had perfect symmetry, there would be no internal fields due to its own mass. However, it is not usually practical to design a satellite with perfectly spherical symmetry.

One example of an experiment which is highly sensitive to the gravitational effects of the capsule which contains it is the Satellite Energy Exchange experiment proposed by Dr. Alvin J. Sanders and Dr. W. Edward Deeds of the University of Tennessee[1]. This experiment requires a cylindrical satellite with a mass on the order of 1000 kg. Figure 1 shows the gravitational field inside a uniform hollow cylinder with end caps with dimensions similar to those planned for the SEE project. The units of this plot are such that the reference field, which is on the order of the experimental fields in

---

[1] Sanders, A.J. and W.E. Deeds. "Proposed New Determination of the Gravitational Constant *G* and Tests of Newtonian Gravitation." Forthcoming in *Phys. Rev. D.* Scheduled for 15 July, 1992.

the project, is equal to one. The field due to the cylinder ought to be much smaller than the reference field, so that the field due to the cylinder does not have a strong effect on the interacting bodies of the experiment. The field near the ends can be many times the reference field. Indeed, even within the experimental working area, the fields can be almost as large as the reference. For this reason, there must be a way of reducing the internal field of this satellite due to its own walls.

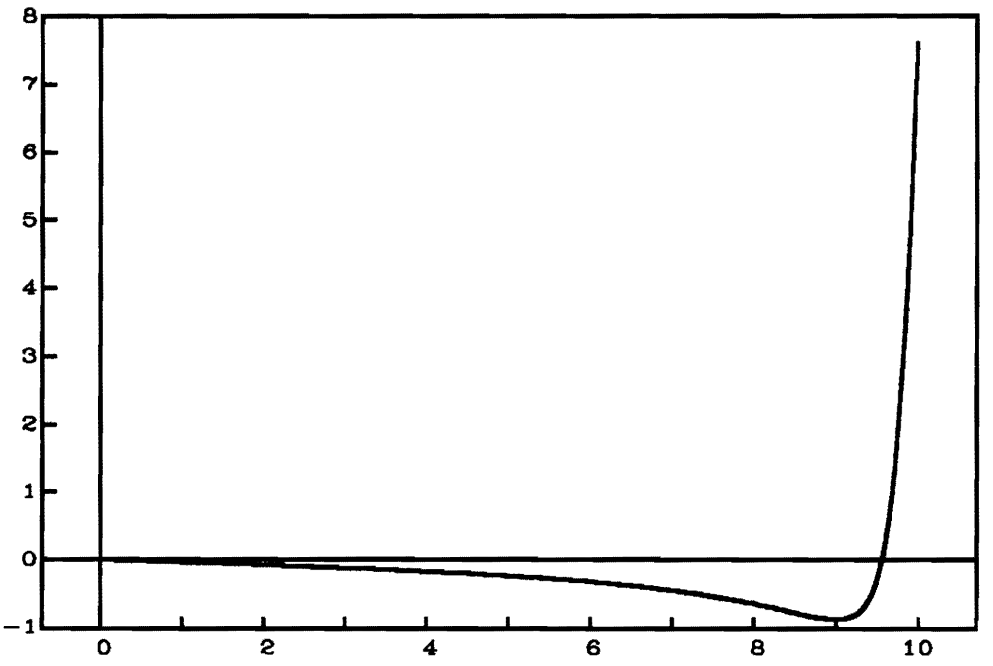


***Figure 1:** Gravitational field inside a hollow cylinder with end caps*

Sections I-III of my paper will focus on finding distributions of compensator masses on and around the SEE satellite which will reduce the field inside the satellite to several orders of magnitude below the reference. Section IV will focus on using compensator masses to reduce the gravitational fields on a space-based crystal-growing experiment.

Section I: Description of the Method

In order to find a method of minimizing the gravitational field inside the satellite, one must first decide on a criterion for what a "small" gravitational field is. For this paper, I have concentrated on reducing the root-mean-squared (RMS) field along the axis of the cylinder to a value of at least two orders of magnitude smaller than the reference field. Here, I am making the assumption that small fields on the axis of the cylinder will result in small fields off of the axis of the cylinder. The method actually entails minimizing an approximation to the mean-squared integral along the axis of the satellite, and calculating the resulting RMS field. This is not the only method available for minimizing the field, however. For example, another method involves minimizing the absolute maximum value of the field on the axis in the region of interest. However, this "minimax" method requires knowledge of the location along the axis of the absolute maximum, which may be difficult to obtain in practice. Also, as will be seen in the next two sections, minimizing the RMS field results in smaller values of the field within the region of interest, while the field grows in absolute value at points approaching the ends of the cylinder, which are not of interest for the SEE project.

My research has used variations on an optimization program, OPT.C, written by Eric Black during his original research[2] into minimizing the field inside the satellite. The programs were originally designed with two methods, a survey search, and a quadratic

---

[2] Black, E.D. "Measuring the Gravitational Constant G Using Artificial Satellites and Horseshoe Orbits." *Research Papers, 1990 Science Alliance Summer Research Fellows in Physics.* University of Tennessee Dept. of Physics. August, 1990.

interpolation, to find the parameters which minimize the mean-squared, and hence the RMS, field.

The survey search entails subdividing an n-dimensional "cube" in parameter space into a grid of evenly spaced points, and evaluating the mean-squared field at each of these points. If the grid were fine enough, then the point which gives the minimum mean-squared field on the grid should lie near the true minimum point which we are trying to locate.

The quadratic interpolation method, however, assumes that the mean-squared field on the axis of the cylinder is a quadratic function of each parameter to be optimized. In other words, the mean-squared force can be represented as a function of n number of parameters p_i by:

$$\mu^2 = \left(p_1 c_1 + p_2 c_2 + \cdots + p_n c_n \right)^2$$

where each c_i is a constant function dependent on the distribution of mass in question, but independent of the value of each parameter p_i. This is justified by a well-known result of Taylor's theorem of n variables states that any function is approximately quadratic when it lies near a local extremum in the space determined by its parameters. In fact, if μ^2 were locally quadratic, then the gradient of μ^2 is locally linear, so the equations for finding the minimum value of μ^2 are linear and can be solved by matrix methods. Eric Black included in his interpolation routine a method of solving linear equations using Gaussian elimination. The interpolation method requires that an initial guess be made of the value of each parameter, and a finite interval around each value be specified. The three points, the central guess and the two surrounding points, in each

direction of the parameter space, allows that there can be fit a unique n-dimensional quadratic function through these points. From this the minimum of this quadratic function can be found. Again, if the initial guess were near the minimum, the interpolation can be expected to predict a "minimum" value even closer to the actual minimum than the original guess. The results of the survey search can be used as a good original guess of the true minimum. This concludes a brief summary of Eric's work.

My enhancements to the above program are threefold:

- First, I added another interpolation routine which assumes the function (the mean-squared field) to be quadratic in the one-dimensional line (which I call the "s line") connecting the original guess to the "minimum" predicted by the previous interpolation. An interpolation in only one dimension requires significantly less computer time to calculate then an n-dimensional interpolation, so a better value can be determined without a significant increase in computer time.
- Second, I made the program able to handle repeated interpolations in both n-dimensions and the 1-dimensional s line. The user is allowed to change the value of the interval around each starting value to handle situations which require the interval be small or large.
- Finally, I added the option of having the program automatically repeat iterations until the predicted values converge (within a specified tolerance) to the true minimum.

The program OPT2.C (listing in Appendix III) has as its parameters the mass and radius of two rings concentric around the ends of the cylinder. Several iterations of the program are required to converge to the minimum value, because the mean-squared field

is only approximately quadratic with respect to these four parameters (as I will argue in the next section). However, the multiple runs, s line, and automatic reiteration routines added to the OPT program are not necessary for the program OPTCONT.C (listing in Appendix III), which has as its parameters the masses distributed continuously over the surface of the satellite, since the mean-squared field is *exactly quadratic* in this case (as I will argue in Section III and Appendix I). Therefore, the exact minimum can be found with a single quadratic fit.

Section II: Using Ring-Shaped Compensator Masses to Reduce the On-Axis RMS Field

Dr. Sanders' original idea for minimizing the on-axis RMS field inside the satellite involved placing thin rings concentric with the ends of the cylinder. Eric Black worked out many of the details of this design. When I began my research I took over the work he had already begun, modifying the OPT programs as described in the previous section. What follows are numerical results I found for the method of using ring-shaped masses, or "compensator rings," around the ends of the cylinder.

Current planning requires that the SEE experiment be conducted in the middle 16 meter length of a 20-meter long cylindrical satellite. My results from this section are those for minimizing the RMS on axis over this length. This is in contrast to later work where the RMS is minimized over the entire 10 meter region. In addition, the first results of this section followed the original requirements that (1) the cylinder radius be 0.4 meters, and (2) the end cap mass be fixed at 0.02, taking the cylinder walls to be unit mass. This corresponds of an end cap thickness of twice the thickness of the cylinder itself, which allows for additional mass to be placed at the ends of the cylinder. The results of Table 2.1 were calculated using a 80-interval multiple 10th-order Newton-Cotes integration. Originally, 4th-order Newton-Cotes integration was used with a high number of intervals, but I found that 10th-order integration worked better. 4th-order integration was used for all other results, which differ from the 10th-order only in the last decimal place of each parameter value (see Appendix II for a description of Newton-Cotes integration).

This section includes results for optimization using both two and three end rings with a fixed end cap. Table 2.1 gives the results for these configurations. The results are impressive, with the RMS field for three rings reduced to 0.0018% of the reference. Indeed, for each parameter, there is roughly an order of magnitude reduction of the RMS field.

	Two Ring	Three Ring
RMS (Reference = 1)	0.000811	0.000018
Radius 1 (meters)	1.36608	7.13038
Mass 1*	0.10417	0.26575
Radius 2 (meters)	5.03291	2.96274
Mass 2*	0.25984	0.12059
Radius 3 (meters)	---------	1.06341
Mass 3*	---------	0.06728

***Table 2.1:** Optimum Ring Parameters*
(cylinder walls have unit mass)*

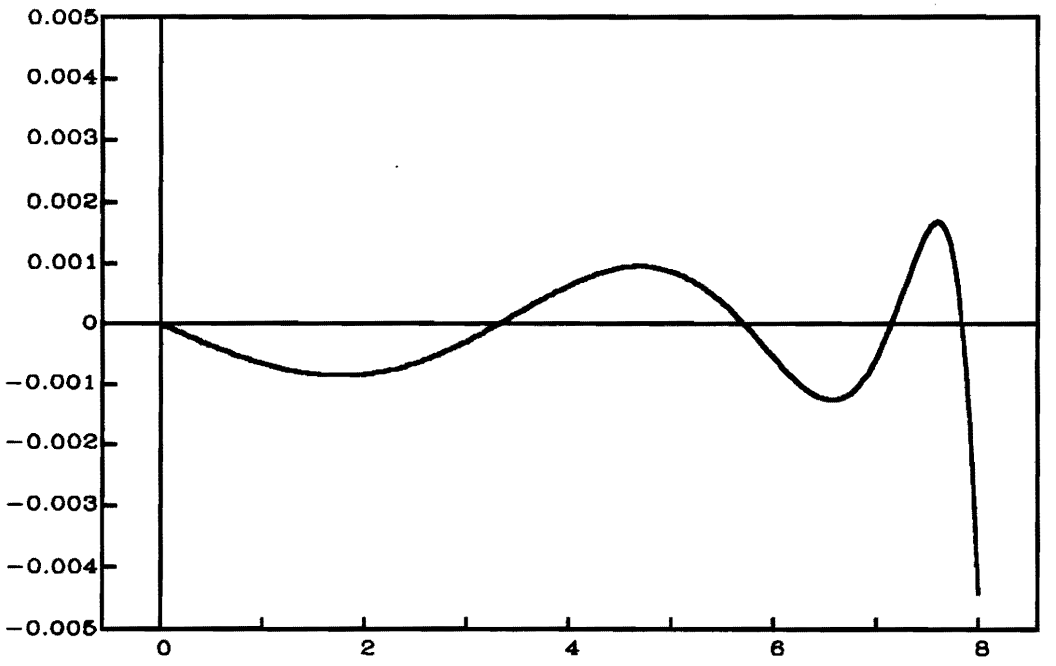


***Figure 2.1:** Plot of field minimized with two end rings*

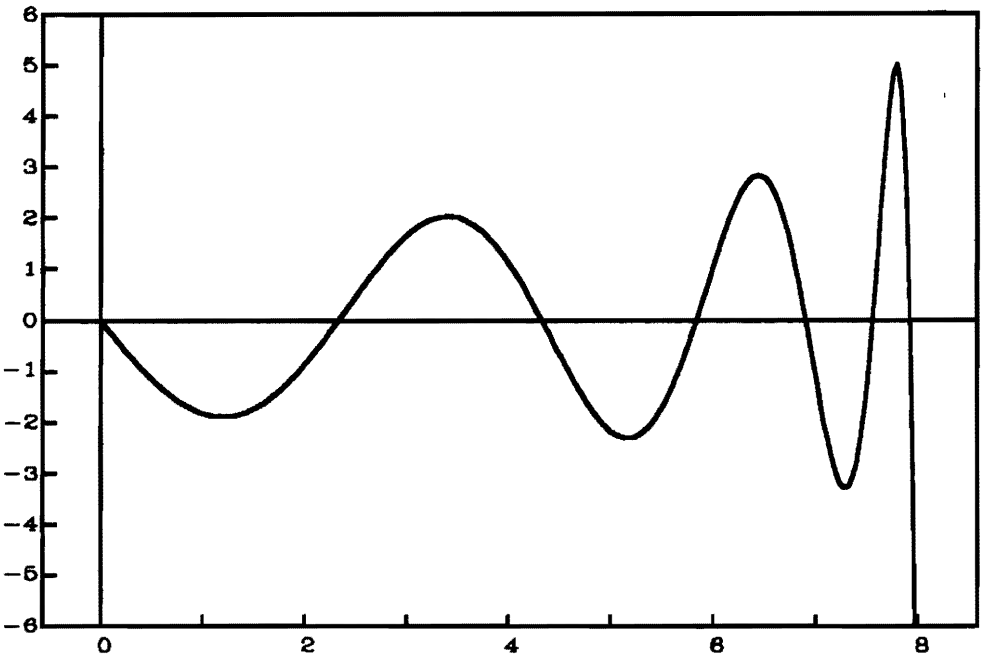


Figure 2.2: *Plot of field minimized with three end rings (Scale: 1 = 1e-5)*

Figures 2.1 and 2.2 show the on-axis field in the region of interest due to two and three-ring configurations. The fields are in terms of the reference field.

Although the previous results were for a fixed end cap mass of 0.02, it is worth studying how the RMS field and optimum ring parameters behave when the end cap mass is changed.

Cap Mass	RMS field	Radius 1	Mass 1	Radius 2	Mass 2
0	0.001046	1.12317	0.11542	4.83824	0.26045
0.01	0.00095	1.22648	0.10885	4.91382	0.26018
0.02	0.000811	1.36605	0.10417	5.0328	0.25983
0.03	0.000594	1.56646	0.10335	5.24514	0.2596
0.04	0.00024	1.87759	0.11178	5.71488	0.26177
0.05	0.000688	2.38995	0.14529	7.33969	0.31069
0.06	0.001982	2.9114	0.24916	14.1791	3.77269

Table 2.2: *RMS and Parameters as a function of End Cap Mass*

Table 2.2 shows the minimum RMS and ring parameters for a two-ring configuration with a variety of end cap masses. It is clear from the Table that the RMS field reaches a minimum for and end cap mass of around 0.04 (actually, the optimum is approximately 0.043). The RMS increases sharply for large values of cap mass. Moreover, increasing the cap mass much beyond this results in unmanageably large ring parameters (in proportion to the rest of the satellite). Thus, this relationship between end cap mass and ring size is a mixed blessing. On the one hand, it means that the SEE needs about four times as much mass at the ends as that which would be due to the wall thickness. This is good to some extent, because it allows instrumentation to be placed at the ends, which the designers feel is necessary. However, if this mass were much greater than 0.04, the minimum RMS field would be large and the optimum parameters would be impractical for use in the project.

Table 2.3 shows the RMS field and optimum parameters for a 2-ring configuration as dependent on the radius of the cylinder (end cap mass = 0.02).

Cylinder Radius	RMS Field	Radius 1	Mass 1	Radius 2	Mass 2
0.4	0.000811	1.36605	0.10417	5.0328	0.25983
0.6	0.000779	1.40772	0.10465	5.06202	0.26031
0.8	0.000745	1.45893	0.10503	5.09614	0.26097
1	0.000712	1.5168	0.1051	5.12999	0.26182
1.2	0.000686	1.57668	0.10462	5.15894	0.26291
1.5	0.000662	1.66267	0.10251	5.1866	0.2651

***Table 2.3:** RMS and ring parameters as a function of cylinder radius*

Clearly increasing the radius reduces the RMS, while requiring little change in the optimum parameters.

Although the results in this section show that compensator rings do a good job in reducing the field within the working length of the cylinder, there are problems. First, optimizing over only 8 meters does *not* guarantee that off-axis fields would be small (as I will argue in the next section). Second, it may not be practical to build a satellite with compensator rings as large as those found to be the optimum. This is especially true for a three-ring configuration. It is for this last reason that I changed the focus of my research to finding a *continuous* mass distribution over the surface of the cylinder, in which the thickness of the cylinder walls and end caps would vary with the position on the cylinder, which reduces the internal field.

Converging to the minimum values took a great deal of work. In fact, the predicted optimum parameters often varied from one iteration to the next, while varying only slightly in minimum RMS values. This led to the conclusion that the mean-square function may form a trough in parameter space. There is an obvious relationship between the mass and radius of each ring, with smaller rings having more influence on the axis and needing less mass. This relationship could easily result in a long, narrow range of optimum values in parameter space. Further evidence of this trough came when I originally tried to find the optimum three-ring parameters. I ran a survey search with six parameters and a fine grid over a long weekend so that I could have some idea what parameters I should use as the initial guess for the quadratic interpolation method. Table 2.4 shows the results of the survey search. Compare these with the interpolation results of Table 2.1. Although the RMS of the former is two orders of magnitude larger than that of the latter, it is much smaller than those of many points near the latter which were calculated by the survey search. It is easy to imagine how a thin trough in parameter

space could be missed by almost every point sampled in a survey search. The disadvantage is that there is no way getting around using the interpolation method to converge to the minimum, thus making the survey search method unnecessary. The advantage is that there should be a great deal of freedom in choosing ring parameters to fit the constraint of practicality of design without a significant increase in internal field over the optimum field strength.

RMS Field	0.001145
Radius 1	5.2
Mass 1	0.1
Radius 2	1.6
Mass 2	0.1
Radius 3	0.2
Mass 3	0.0125

***Table 2.4:** Results of Survey Search*

Section III: Minimizing the Internal Field Using a Continuous Mass Distribution

The majority of my research has been focused on finding a continuous mass distribution along the length and over the ends of the cylinder which minimizes the field inside the satellite. The justification for the existence of such a distribution comes from a well-known result of electrostatics. If the capsule were a conductor with a net charge on its surface, the charge would be uniquely distributed in such a way as to make the electric field zero at every point inside the cylinder. Extension of this result to gravitational fields suggests that there is a *unique* distribution of mass over the length and ends of the satellite which would make the gravitational field inside *exactly zero at every point inside the cylinder*. I will approximate this distribution by a Taylor series of even powers of z (the axial distance from the center of the cylinder) and ρ (the radial distance from the center of the end caps). Odd coefficients of the Taylor series vanish by symmetry.

One *major* change from the previous section involves integration of the squared field over the entire ten meter length of the capsule. The justification for using only eight meters was that this is the only region of importance for the SEE project. However, when I originally tried to find the continuous distribution by optimizing over only eight meters, the program predicted a negative end cap mass. This contradicts the physical situation as can be seen in the electrostatic case where, if the net charge were positive, this charge would not distribute itself so as to have both positive and negative values at different points on the surface. Moreover, there is no such thing as a negative mass

density, so that while negative masses are perfectly legitimate from a mathematical point of view, we take special care to make sure the optimum design of the capsule excludes this possibility. Re-optimizing over the full ten meters (actually 9.99 meters) gave a positive end mass.

This change of integration range can be further justified by appealing to the uniqueness argument of above. Since there is a unique distribution which makes the field exactly zero at every point inside the cylinder, this should be the same distribution as that which makes the field exactly zero at every point on the axis of the cylinder. Treating the field due to a sequence of distributions (of higher and higher numbers of parameters) as a sequence which converges to zero, it can be argued that *pointwise convergence* over the first eight meters, where the field approaches zero only in this region, does not guarantee *uniform convergence*, where the field approaches zero everywhere over the entire length. In fact, when the minimization was only over the first eight meters, the field outside of this region grew rapidly. Since there is no guarantee that the distribution which zeroes the field over the first eight meters approximates the ideal, there can be no guarantee that the off-axis field due to these parameters is small. Thus, an efficient way to insure against high off-axis fields is to try to zero the field over the entire length.

The version of the OPT program used in this section to find a continuous distribution uses as its parameters the masses which are distributed the powers of z and ρ and the mass of a ring placed at each end of the cylinder, with a radius equal to that of the cylinder. A brief consideration of the nature of a Taylor series suggests that a ring of

this type should act physically as the remainder term of both the z and the ρ expansions. The results from the previous section showed that the field "crosses the axis" once for each parameter. The results from this section will support these findings.

There is a significant benefit in optimizing a distribution, discrete or continuous, which has *masses* as its only parameters. Since the field at each point is simply a superposition of linear functions of each mass parameter, the mean-squared field will be a *quadratic* function of each parameter, as shown in Appendix I. Since OPT uses quadratic interpolation to predict the minimum mean-squared field, the program is guaranteed to find the exact minimum without iteration if the function is exactly quadratic. This results in a vast improvement in calculation time and allows the addition of many more parameters than were previously possible. It also eliminates the need for initial values to be close to the minimum in the running of the OPT program. I will describe an additional benefit of this new-found freedom later in this section.

Table 3.1 shows the ring mass and the minimum RMS for several combinations of j and k, which I am calling the maximum powers of z and ρ. These values show how the end ring mass and the minimum RMS field act as functions of maximum z and ρ powers. The RMS field is in units of the reference.

	j = 4	**j = 8**	**j = 12**	**j = 16**
k = 0	Mass = 0.03057 RMS = 0.12986	Mass = 0.02471 RMS = 0.02371	Mass = 0.02294 RMS = 0.00970	Mass = 0.02348 RMS = 0.00920
k = 2	----------	Mass = 0.02663 RMS = 0.02246	Mass = 0.01941 RMS = 0.00545	Mass = 0.01609 RMS = 0.00124
k = 4	----------	----------	----------	Mass = 0.01894 RMS = 0.00090

***Table 3.1:** End ring masses and minimum RMS field as a function of maximum powers of z and ρ*

I am only including specific results for a few distributions. I am excluding the results for those cases with very high number of ρ terms, because in these cases negative end cap density occurred near the edge. The rule of thumb we found is that we must have $k \leq \frac{j}{4}$ to assure that the end cap density is everywhere positive. In other words, an unphysical mass distribution on the end cap will result if this distribution is allowed too much freedom. Figures 3.1 - 3.3 show the field along the axis of the cylinder due to some of the optimum parameters found by the OPT program.

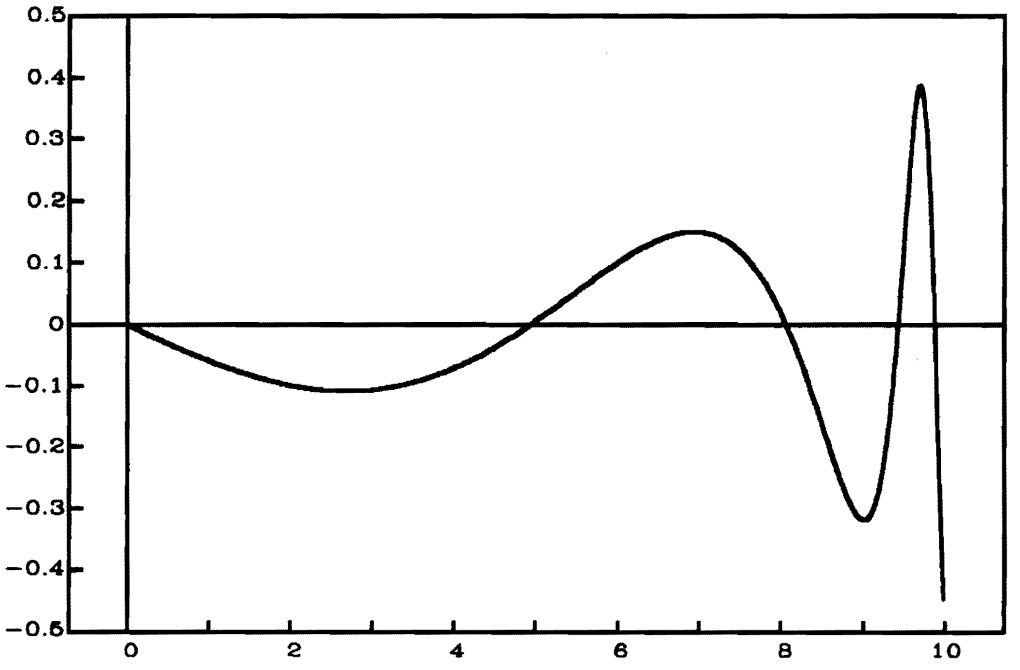


Figure 3.1: $j = 4, k = 0$

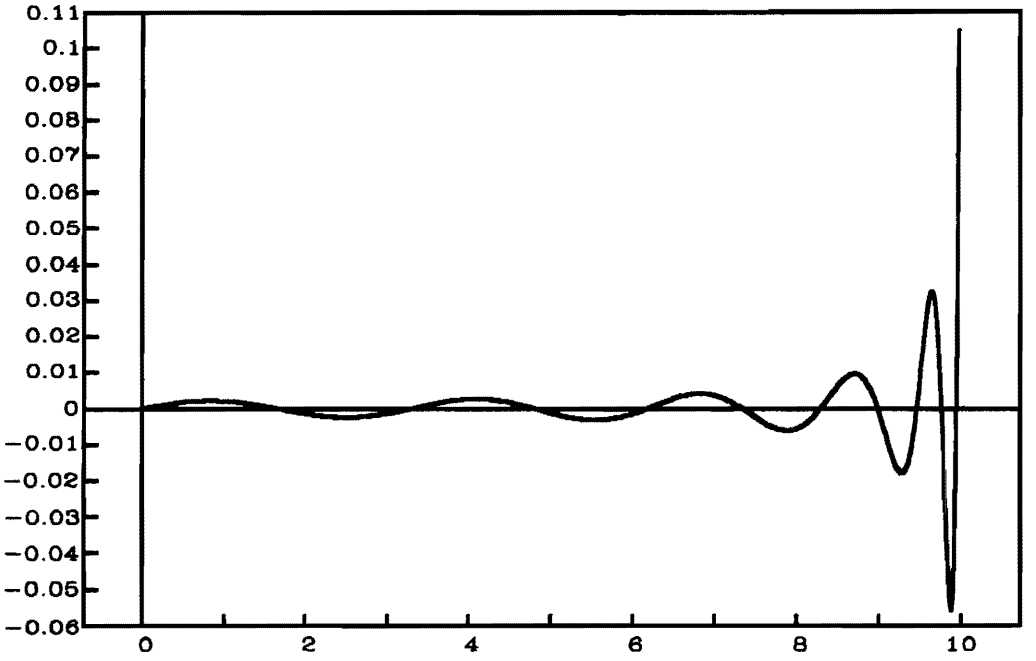


Figure 3.2: *$j = 16, k = 0$*

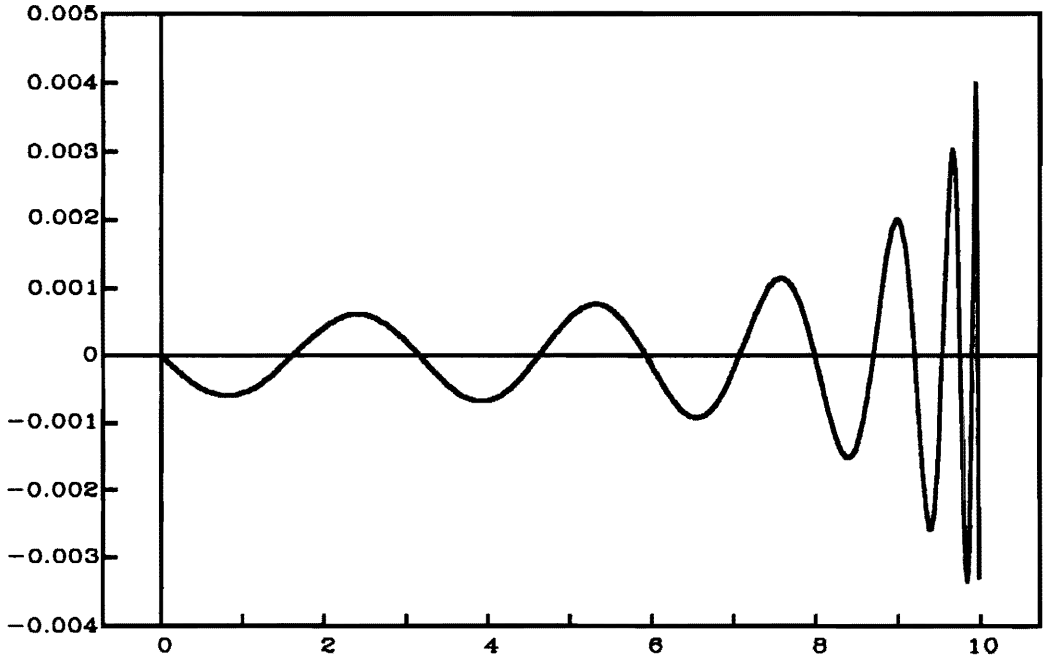


Figure 3.3: *$j = 16, k = 4$*

Notice that the optimum fields in the experimentally interesting region, $|x| \leq 8$, are never better than those for the two-ring case of the previous section. This is not surprising, since integration over the entire length "sacrifices" the inner region of the cylinder in order to reduce the field near the ends.

Comparison of Figures 3.1 and 3.2 shows that adding further powers of z essentially shortens the "quasi-period" of the field, i.e. the size of the intervals between crossings of the x-axis. On the other hand, comparison of Figures 3.2 and 3.3 shows that adding further powers of ρ results in a higher number of crossings near the *ends* of the cylinder. This also is not surprising, since any mass at the end of the cylinder has a limited range of influence. Thus, the z and ρ parameters can work together, the z focusing on the interior of the cylinder, while the ρ powers concentrate on the ends, to zero the field on the axis.

There is an analogy between the continuous distribution and the discrete distribution of rings. High powers of z and end rings of smaller radius have a short-range effect and influence the points only near the ends of the cylinder. Conversely, low powers of z have a long-range effect, and influence the fields over the entire length of the cylinder, as do end rings of large radius. Thus, if milling limitations preclude the possibility of designing a capsule with its mass distributed as a high power of z, then an alternative design might involve lower powers of z and end rings of smaller radius to approximate the high powers of z.

This reasoning also leads to an explanation of why the end ring did not act as the remainder term for the ρ distribution in this work and why high numbers of ρ parameters result in a negative mass density near the edge of the end cap. The addition of each

power of ρ results in another crossing of the axis close to the end of the cylinder. Integration techniques in the current version of the program are not accurate when the size of the intervals between crossings are smaller than the integration intervals, especially when there is a large variation between local extrema of the field. Since the addition of ρ parameters results in tightly spaced crossings near the ends of the cylinder, the integral over this region is not accurate, so the calculated optimum parameters should not be accurate.

Further Improvements

In order to carry out further studies on any distribution of mass which would reduce the internal field of the satellite, the following approaches might be useful. First, as long as the optimization parameters are only masses, a significant improvement in computer time can be achieved. Since the quadratic nature of the mean-squared field makes the initial value unimportant, I have arbitrarily an initial value of zero for each parameter. I did not realize at first the tremendous advantage of this choice. As long as all other parameters are zero, the second partials and mixed partials calculated by the computer are *independent of any parameter other than those involved in the derivative.* This powerful fact insures that a single run can be used for the largest expected number of parameters, and the matrix of second partials of every combination of smaller powers of z and ρ will consist of values which can be found in the matrix from the run of the large number of parameters. Therefore, the optimum values of any configuration of powers of z and ρ which are lower than some previously calculated maximum values can be found without calculating the matrix of second partials of this configuration.

My second recommendation involves a modification of the program's integration technique. Dr. Sanders suggested a Gauss-Chebyshev quadrature technique to get a better estimate of the integral. I concur that time should be taken to include this or something similar. However, we did not see a way to quickly implement this. A further benefit of Gaussian quadrature is that it can handle the case where the function has a singularity at the endpoints, as is the case with this project. Furthermore, it can be shown that any Gaussian technique gives a much better estimate of the integral than a Newton-Cotes technique can when using the same number of points. However, in doing so, one must also find a way to make the number of intervals over which the integration is performed and the width of each interval dependent on the number and type of parameters used and the position of each interval. For example, treating the axis crossings as the endpoints of the intervals allows significant weight to be placed on calculations at the ends of the cylinder, where any integral approximation would first break down. It is worth trying to find a way to predict approximately where each axis crossing would be so that this can be done.

Section IV: Reducing The Gravitational Fields In A Crystal-Growing Experiment

The previous sections have shown how distributions of compensator masses can be found which will nearly zero the internal gravitational field of a satellite due to its own walls and that the field can be zeroed to any accuracy. This section will present a method of using compensator masses to zero the field on one part of an experiment being conducted in one part of the satellite due to the other parts of the experiment. I apply the techniques used in previous sections to reduce the gravitational fields in space-based crystal-growing experiments. I intend to show that, while a satellite design such as the one in the previous section will significantly reduce the gravitational fields on space-based experiments, further reduction can occur by placing additional compensator masses to the capsule to cancel the effects of one part of an experiment on another.

In this study I assume that 5 experiments are being conducted in a cylindrical satellite which has been designed with a mass distribution as described in the previous section. I also make the further assumption that the ovens in which the experiments are being conducted are of unit mass and are similarly designed. Thus, the only gravitational fields affecting each experiment are those due to the *other ovens*. I have arbitrarily chosen to work with ovens which are 0.4 units wide and which are each separated by a 1-unit distance along the axis of the satellite. Finally, I have made the assumption that the gravitational field due to each oven is that of a point mass. The reasoning for this is that we do not know what the *exact* mass distribution of each oven would be. A point mass assumption is good to first approximation, which is all that is necessary since it is

the intention of this study to show that the proposed method *can work*, not to get exact values for a specific situation which would never occur in practice.

The gravitational field on an oven at position x due to an oven at a is the familiar $f = \frac{a-x}{|a-x|^3}$, (where G is 1). By symmetry, the field at zero (i.e. the center of the middle oven) is zero. However, the fields off center can get quite large (I am assuming much larger than desirable for a given experiment). Figure 4.1 shows the fields inside each oven due to the other ovens.

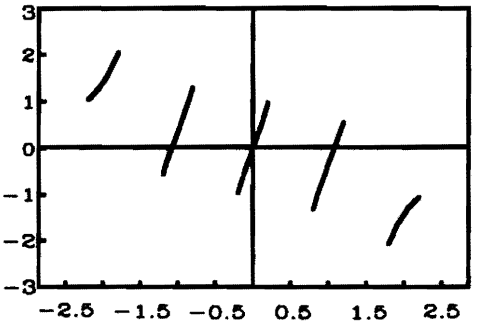


***Figure 4.1**: Uncorrected field in each oven due to all other ovens.*

All figures in this section show the fields inside each oven only, with all five ovens appearing in each graph. The x-axis in each graph is the position inside the satellite, while the y-axis is the field strength in arbitrary units. In addition, all graphs in this section have the same scale on the y-axis so the improvement at each step of the method becomes quite clear. The scale of the y-axis is such that $1 = 2.7 \times 10^{-9}$ g, where g is the strength of the gravitational field at the surface of the earth. In other words, the scale of the y-axis is 1 = 2.7 "nano-g" or ng.

The first step in reducing the internal fields of each oven is to use compensator masses to zero the fields at the center of each oven. Again, this is already the case for

$x = 0$. Somewhat arbitrarily, I have chosen to place the rings at $x = \pm 1.5$ and $x = \pm 2.5$ to zero the fields at $x = \pm 1$ and $x = \pm 2$. By symmetry, it is sufficient to zero the fields at the positive x values.

Now the gravitational field at x due to a thin ring of mass m, radius r, and centered at a is given by:

$$f = \frac{m(a-x)}{\left[(a-x)^2 + r^2\right]^{3/2}}.$$

It can easily be shown that the field is an extremum when $|a-x| = \frac{r}{\sqrt{2}}$. If a ring were placed at a distance of 0.5 units from the center of an oven, then a ring of radius $r = \frac{1}{\sqrt{2}} =$ 0.7071... would have the maximum influence on the center of the oven. For this study I initially concentrated on rings with radius r = 0.7071..., which gives the maximum influence of each ring on the oven centers. I also considered rings of radius r = 0.5 and r =0.3535... (which is $\frac{1}{2\sqrt{2}}$), to see what benefits might be gained by using smaller rings. It turned out that the rings with smaller radius work better.

To find the ideal ring masses which zero the fields at the center of the ovens, simply add up the influence of each ring and equate that total to the negative of the field at that point due to the ovens. This leads to two equations with two unknowns, the masses of the rings. Solving these equations for the 3 different ring radii gives the following optimum masses:

	$r = 0.707...$	$r = 0.5$	$r = 0.3535...$
$m_{1.5}$	-0.14709	0.030224	0.05936
$m_{2.5}$	1.798791	1.075429	0.731833

***Table 4.1**: Ring masses which zero fields at the oven centers.*

The fact that the case where r = 0.707... predicts a negative ideal mass for the ring at ±1.5 probably means that the placement I chose is not the best for rings this large. However, for the two smaller radii, the values are positive and therefore, realistic. Figure

4.2 is an example of the fields inside each oven when the fields at the centers of the ovens are zeroed. It is for the case $r = 0.5$, but the results are similar for each radius.

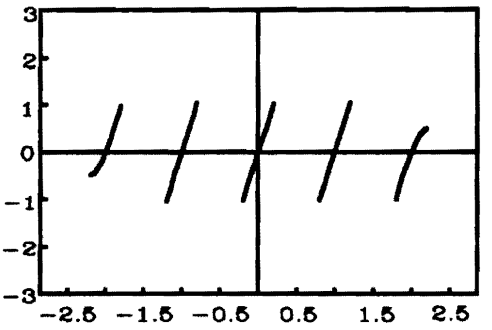


Figure 4.2: *Fields zeroed at the center of each oven.*

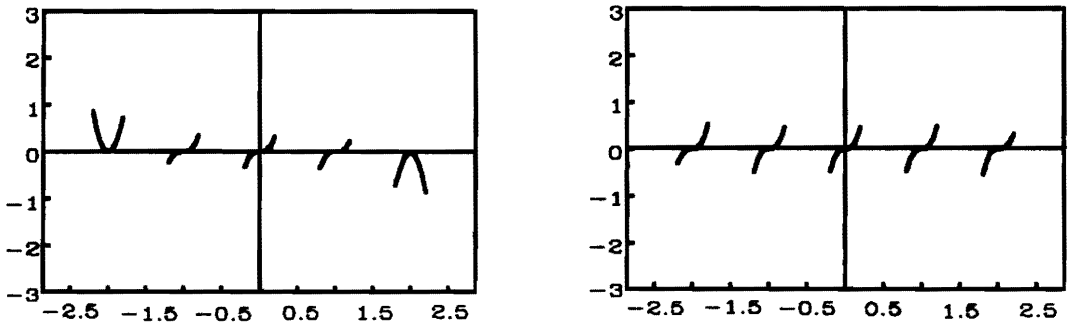
It is clear from Figure 4.2 that although the fields are zero at the centers of each oven, they can become large away from the centers. This is clearly due to the *gradient* at the center of each oven. Zeroing the gradient at the centers should reduce the total variation in the field over the entire width of each oven. The most obvious placement of the ring for the purpose of zeroing the gradient at a point x is to center the ring at x itself. Thus, the ring will have no effect on the point, but will influence everything on either side of the point x. To zero the gradient, simply compute the gradient at a point x due to *all* rings and equate it to the negative gradient at x due to *all ovens*. The ring masses which achieve this are found by solving a 3-by-3 matrix equation. However, to make this improvement worthwhile, it is still important to zero the fields at the centers with the rings at $x = \pm 1.5$ and $x = \pm 2.5$. Therefore, to find the ring masses which will zero the fields and the gradients at the center of each oven, one must solve a 5-by-5 matrix equation. Table 4.2 shows the solutions to this equation for each ring radius. Notice that for the case there $r = 0.707...$, there are no longer any negative masses so that

rearrangement of the ring positions is no longer necessary. However, the required masses for $r = 0.707...$ are large, and total almost twenty oven masses. This is clearly an undesirable result for applications in space.

	$r = 0.707...$	$r = 0.5$	$r = 0.3536...$
m_0	4.200686	0.902643	0.249408
m_1	4.092852	0.921817	0.261982
$m_{1.5}$	0.891787	0.118642	0.058355
m_2	2.062748	0.809259	0.306361
$m_{2.5}$	8.071369	1.914105	0.884184

***Table 4.2:** Masses which zero gradients at the oven centers.*

Figure 4.3 consists of two plots which show how the field is reduced by zeroing the gradients at the center. Note the difference between the fields around $x = 2$. This is not so surprising, since we should not expect there to be symmetry at $x = 2$ like that at $x = 0$.



$r = 0.7071...$ $r = 0.3536...$

***Figure 4.3:** Fields zeroed at the centers, gradients zeroed at the centers.*

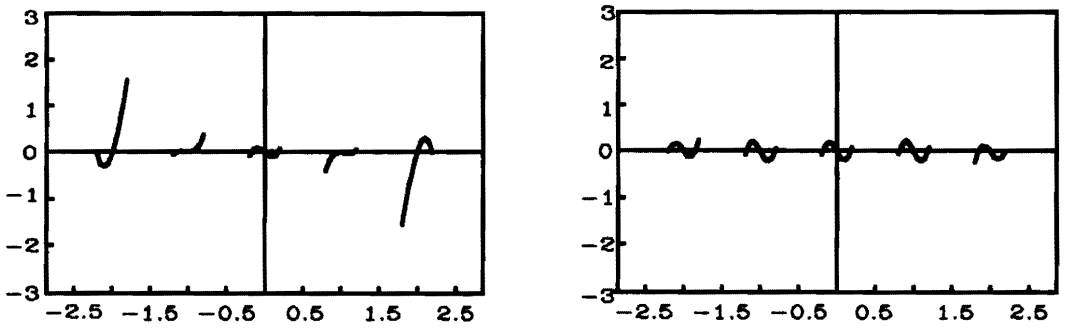
One more strategy for reducing the gravitational fields inside each oven was tried. This strategy requires no new rings, just simply a change in purpose of the existing ones. By symmetry, it is clear that if the gradient were zeroed at a point halfway to one

end of the central oven, the gradient at the point halfway to the other end would also be zeroed. In the process, this should reduce the amount of *variation* of the field inside the oven by more than if the gradient were zero at the center, since it is zero at more places inside the oven. Assuming that the symmetry argument were at least *somewhat true* for the off-center ovens, then zeroing the gradient at an off-center point should be helpful in each oven. Table 4.3 shows the optimum ring masses for this case. Again, all values are positive and realistic.

	$r = 0.7071...$	$r = 0.5$	$r = 0.3536$
m_0	4.921396	1.257528	0.418309
m_1	4.204325	1.242655	0.441241
$m_{1.5}$	2.103936	0.286011	0.074006
m_2	0.37551	0.785097	0.461635
$m_{2.5}$	9.661947	2.348502	1.003987

***Table 4.3:** Masses which zero off-center gradients.*

Figure 4.4 compares the plots of the fields due to the case where $r = 0.7071...$ and $r = 0.3536....$ The results are similar to those when the gradients at the centers were zeroed, but it is clear that the field is even smaller for this case.



$r = 0.7071...$ $r = 0.3536...$

***Figure 4.4:** Off-center gradients are zeroed*

This study has shown that the addition of only a few compensator masses to the capsule can significantly reduce the gravitational field inside each crystal-growing oven due to the other ovens. In fact, the smaller ring radius gives the best results and requires the least amount of mass to reduce the field in every oven. For the smallest rings, with radius r = 0.3536..., the maximum field is approximately 0.2 in the arbitrary units of these graphs, or about 0.5 ng. This is roughly a factor of ten decrease in the maximum uncorrected field (as seen in Figure 4.1). For the larger rings, the maximum field is not significantly reduced in the outer ovens with this configuration, however.

The intention of zeroing the gradients off-center was to make the field inside each oven resemble a 3rd-order Chebyshev polynomial, which satisfies a minimax requirement. The fields inside the middle oven for each ring radius and in every oven for the smallest ring radius do resemble this polynomial. However, due to asymmetry, the fields in the outer ovens for the two larger ring radii do not. Moreover, the maximum field inside the outer ovens in the case using the largest ring is not significantly smaller than the maximum uncorrected field in this oven. Therefore, if rings of larger radius are to be employed, one must look into placing them at positions other than exactly between two ovens. The new positions would be such that the minimax requirement would be satisfied. Further study needs to be done on this, since it is more reasonable to expect that the design of the crystal-growing experiment would require larger rings.

Appendices
(Including Program Listings)

Mark Edward Rupright

Tennessee Scholars Senior Thesis

May 5, 1992

Appendix I: Proof that the Mean-Squared Field is Quadratic With Respect to Mass Parameters

The gravitational field of a point mass is a linear function of the mass of the object, and is also a function of the distance from the object. Similarly, the field of an extended object is always a linear function of mass, and is also a function of the shape, size, and distance of the object. It is this linearity in mass which is important. Let $g_n = m_n f_n(r,s)$ represent the gravitational field at a point due to a mass m_n where $f_n(r,s)$ is the part of the field due to the size and shape of the object and the distance from the object. The total gravitational field F due to n masses is, simply,

$$F = \Sigma m_n f_n(r,s)$$

Now, the function we are trying to minimize is the mean-squared gravitational field, which is given by

$$\mu^2 = \frac{1}{L}\int_0^L F^2\,dx$$

or, expanded, by

$$\mu^2 = \frac{1}{L}\int_0^L \left[m_1 f_1(r,s) + m_2 f_2(r,s) + \cdots + m_n f_n(r,s)\right]^2$$

where L is the length of the cylinder. Thus, it is clear, then, that the mean-squared gravitational field due to a system of n bodies, is a quadratic function with respect to masses of the objects.

Tenth Order

$$\int_{x_0}^{x_{10}} f(x)\,dx \cong \frac{5h}{299376}[16067(f_0+f_{10})+106300(f_1+f_9) \\ -48525(f_2+f_8)+272400(f_3+f_7) \\ -260550(f_4+f_6)+427368f_5]$$

Notice that for the tenth-order integration, negative weights are used. This could cause significant problems, since while the squared field is always positive, the tenth-order Newton-Cotes integral of the squared field could be negative. The problem results from the fact that high-order interpolants to a function often do not come close to approximating the function. Thus, Newton-Cotes integration of high order is not an effective technique of finding the integral of complicated functions, such as the on-axis field in this study when there are several ρ parameters.

Appendix II: A Brief Description of Newton-Cotes Integration

Closed-form Newton-Cotes integration is a technique in which an n^{th}-order polynomial is fit through n+1 equally spaced points, with the first and last points being the endpoints of the integration interval. First-order Newton-Cotes integration is the familiar trapezoid rule, which results from integrating a linear interpolant to a set of data. Second-order Newton-Cotes integration, also known as Simpson's rule, results from integrating a quadratic interpolant to a set of data. For my research, I used fourth-order Newton-Cotes integration (also known as Bode's rule) for the early results, and tenth-order integration for the later results. What follows is the mathematical formulation for second, fourth, and tenth-order Newton-Cotes integration.

For each integration interval, the function to be integrated must be evaluated at n+1 equally spaced points x_i, with x_0 being the beginning the interval, and x_n being the end of the interval. Let h be the separation of the points, and let f_i be the value of the function evaluated at the point x_i. Then the following are the formulae for calculating the various Newton-Cotes integrals:

Second Order (Simpson)

$$\int_{x_0}^{x_2} f(x)\,dx \cong \tfrac{h}{3}[f_0 + 4f_1 + f_2]$$

Fourth Order (Bode)

$$\int_{x_0}^{x_4} f(x)\,dx \cong \tfrac{2h}{45}[7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4]$$

Appendix III: Program Listings

Opt2.C

Opt3.C

Optcont.C

```
/*                          PROGRAM OPT2                                */

     /* WORKS AS OF  May 5, 1992 */
     /* Program finds minimum of a quadratic function */
     /* from a space of dimension DIMS to the real line. */
     /* When changing dimensions, DIMS must be changed in */
     /* the #define DIMS expression, and DIM must be changed */
     /* accordingly in MATRIX4.C, which is a series of  */
     /* subroutines to calculate the inverse and determinant */
     /* of a matrix.    */

     /* Survey search works on any function. */
     /* Quadratic interpolation only works on quadratic functions. */
     /* Two rings, 10th order Newton-Cotes integration */

#include <stdio.h>
#include <time.h>
#include <math.h>
#include "matrix4.c"

        /*  Capsule constants */
        /*  DIMS :  Number of parameters to be optimized  */
        /*  CAPM :  Mass of each end cap (cylinder = 1)   */
        /*  RADIUS : Radius of the cylinder               */
        /*  LENGTH : Length of the cylinder               */

#define DIMS 4
#define CAPM .02
#define RADIUS .4
#define LENGTH 20
#define POW 21

double xhi,xlo;
double I[DIMS][DIMS];
int n,p;

main()
{
        /*      Important Variables     */
        /*   v[] : initial parameter values     */
        /*   beta[] : finite interval to calculate derivatives */
        /*   u11[] : optimum parameter values   */
        /*   A[][] : matrix of 2nd Partials     */
        /*   CON2[][] : Condensed matrix of 2nd partials */
        /*   b[] : matrix of 1st Partials       */
        /*   con1 : Condensed matrix of 1st partials    */
        /*   bestmsq : minimum mean-square found in survey */
        /*   dtd : distance along axis of parameter space  */
        /*         at which the function should be double  */

int i,j,k,l,choice,x,y,sum,fix[DIMS],m[DIMS],o,q;
double chinew,chimin,dummy,dis,tol,c;
double v[DIMS],beta[DIMS];
double u11[DIMS],u_11[DIMS],u1_1[DIMS],u_1_1[DIMS];
double A[DIMS][DIMS],b[DIMS],bestmsq[DIMS];
double CON1[DIMS],CON2[DIMS][DIMS];
double meansq(),det(),max();
double p[DIMS],dtd[DIMS];
double f1,f2,f3,gamma[DIMS],delta[DIMS],mid[DIMS];
double sb,sb2,sm,sm2,f4,best[DIMS],s2;
double factor,dtds,epsilon,t1,t2,ch[DIMS];

void inv();
        /*  Open File opt2.dat and write current time  */

FILE *stuff;
```

```
struct tm *timeptr;
time_t timer;
time(&timer);
timeptr=localtime(&timer);

stuff=fopen("opt2.dat","w");

fprintf(stuff,"%s\n",asctime(timeptr));

printf("**********PROGRAM OPT**********\n");

        /*  Set limits of integration on axis */
        /*  n=number of intervals */

xlo=0; xhi=8; n=80;


        /* fix[i] for i=0,1,2,3 set at zero. Later, if a */
        /* variable k is fixed, fix[k] will be set to 1. */

for(k=0;k<DIMS;k++)
 fix[k]=0;

printf("Which kind of search do you want to perform?\n");
printf("1. Survey search based on mean square\n");
printf("2. Quadratic interpolation of minimum mean square\n");
printf("Enter the number of your choice : ");
scanf("%d",&choice);
printf("\n");

        /* Here, and in the function, are the only places */
        /* where the physical meaning of the variables is */
        /* significant for coding. */

printf("\nvariable    physical meaning\n");
printf("  1           radius1\n");
printf("  2           mass1\n");
printf("  3           radius2\n");
printf("  4           mass2\n");


fprintf(stuff,"\nvariable    physical meaning\n");
fprintf(stuff,"  1           radius1\n");
fprintf(stuff,"  2           mass1\n");
fprintf(stuff,"  3           radius2\n");
fprintf(stuff,"  4           mass2\n");

fprintf(stuff,"\nRadius   = %lf\n",RADIUS);
fprintf(stuff,"Cap mass = %lf\n",CAPM);
fprintf(stuff,"Length   = %lf\n",LENGTH);




if(choice==1)     /*  Begin Survey Search  */
{
printf("Enter the parameters of the search.\n\n");

printf("First, enter the center point.\n");
printf("Type in the coordinates in %d-space\n",DIMS);
printf("all separated by commas :\n");

        /* THIS MUST BE CHANGED BEFORE COMPILING */
        /* TO CHANGE THE DIMENSION OF THE PROBLEM */
```

```
scanf("%lf,%lf,%lf,%lf",&v[0],&v[1],&v[2],&v[3]);

printf("\nEnter the spacing between the points.\n");
printf("(along the same format as above)\n");

        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

scanf("%lf,%lf,%lf,%lf",
     &beta[0],&beta[1],&beta[2],&beta[3]);


printf("Now enter the number of points ON EACH SIDE \n");
printf("at which you wish to calculate values.\n");
printf("(that is, if you want 3 points on each side,\n");
printf("then a total of 7 values will be calculated along\n");
printf("that particular axis)\n");
printf("enter %d values, each separated by commas\n",DIMS);

        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

scanf("%d,%d,%d,%d",&m[0],&m[1],&m[2],&m[3]);


fprintf(stuff,"Var. 1    Var. 2   Var. 3    Var. 4");
fprintf(stuff,"       MS\n");

chimin=10000;



        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */
        /* NEW LOOPS MUST BE CONSTRUCTED IN A NESTED */
        /* FASHION TO INCLUDE NEW DIMENSIONS */

u11[0]=v[0]-m[0]*beta[0];                    /* set var. 1 */
for(i= -m[0];i<=m[0];i++)                     /* var. 1 loop */
 {u11[1]=v[1]-m[1]*beta[1];                  /* set var. 2 */
 for(j= -m[1];j<=m[1];j++)                    /* var .2 loop */
 {u11[2]=v[2]-m[2]*beta[2];                  /* set var. 3 */
  for(k= -m[3];k<=m[2];k++)                    /* var. 3 loop */
  {u11[3]=v[3]-m[3]*beta[3];                 /* set var. 4 */
   for(l= -m[3];l<=m[3];l++)                    /* var. 4 loop */
{
  chinew=meansq(u11);

  if(chinew<chimin)
   {chimin=chinew;
    bestmsq[0]=u11[0];
    bestmsq[1]=u11[1];
    bestmsq[2]=u11[2];
    bestmsq[3]=u11[3];}

  if(chinew<0.0000000408)
   {fprintf(stuff,"%8.5lf %8.5lf",u11[0],u11[1]);
   fprintf(stuff,"%9.5lf %8.5lf",u11[2],u11[3]);
   fprintf(stuff,"   %e\n",chinew);}

  u11[3]=u11[3]+beta[3];} /* iterate variable 4 */
  u11[2]=u11[2]+beta[2];} /* iterate variable 3 */
  u11[1]=u11[1]+beta[1];} /* iterate variable 2 */
  u11[0]=u11[0]+beta[0];} /* iterate variable 1 */


fprintf(stuff,"\n\nMinimum root-mean-square found by dumb search\n");
fprintf(stuff,"%e\n\n",sqrt(chimin));
```

```
fprintf(stuff,"\n\nMaximum force for that geometry\n");
fprintf(stuff,"%e\n\n",max(bestmsq));

fprintf(stuff,"variable 1 = %15.41f\n",bestmsq[0]);
fprintf(stuff,"variable 2 = %15.41f\n",bestmsq[1]);
fprintf(stuff,"variable 3 = %15.41f\n",bestmsq[2]);
fprintf(stuff,"variable 4 = %15.41f\n",bestmsq[3]);
time(&timer);
timeptr=localtime(&timer);
fprintf(stuff,"%s\n",asctime(timeptr));
goto stop;
}    /*  End Survey Search  */


if(choice==2)  /*  Begin Quadratic Interpolation  */
{
  printf("Tolerance?");   /* Tolerance for Automatic Runs */
   scanf("%lf",&tol);
   printf("%lf",tol);

printf("Enter the parameters of the search.\n\n");
printf("First, enter the center point of the 4-cube.\n");
printf("Type in radius1, mass1, radius2, mass2\n");


        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

printf("all separated by commas :\n");
scanf("%lf,%lf,%lf,%lf",&v[0],&v[1],&v[2],&v[3]);

printf("\nEnter the spacing between the points.\n");
printf("(along the same format as above)\n");


        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

scanf("%lf,%lf,%lf,%lf",
       &beta[0],&beta[1],&beta[2],&beta[3]);

        /* Main interpolation loop */

        /* Set fix[i]'s to determine which variables */
        /* are fixed & subtract total from DIMS to  */
        /* determine how many (sum) are to vary. */


redo:   /* Begin Main Interpolation loop */

sum=DIMS;
for(k=0;k<DIMS;k++)
 {if(beta[k]==0) fix[k]=1;
  sum=sum-fix[k];}

printf("\n\nWorking...\n");

        /* Calculate Matrix Elements */

for(i=0;i<DIMS;i++)
 for(j=i;j<DIMS;j++)
  {for(k=0;k<DIMS;k++)
   {
        /* set finite difference vectors to center of grid */

   u11[k]=v[k]; u_11[k]=v[k]; u1_1[k]=v[k]; u_1_1[k]=v[k];}
```

```
 if(j==i)
  {u11[i]=u11[i]+beta[i];
   u_1_1[i]=u_1_1[i]-beta[i];


        /* If variable is fixed, set derivatives equal to zero, */
        /* otherwise, approximate them numerically. */

   if(fix[i]==1)
    {A[i][i]=0; b[i]=0;}

   else
    {A[i][i]=(meansq(u11)-2*meansq(u_11)+meansq(u_1_1))
            /(beta[i]*beta[i]);
     b[i]=(meansq(u_1_1)-meansq(u11))/(2*beta[i]);}
  }

   if(j!=i)
    {u11[i]=u11[i]+beta[i];
     u11[j]=u11[j]+beta[j];

     u_11[i]=u_11[i]-beta[i];
     u_11[j]=u_11[j]+beta[j];

     u1_1[i]=u1_1[i]+beta[i];
     u1_1[j]=u1_1[j]-beta[j];

     u_1_1[i]=u_1_1[i]-beta[i];
     u_1_1[j]=u_1_1[j]-beta[j];

        /* If variable is fixed, set derivative equal to zero, */
        /* otherwise, approximate it numerically. */

   if(fix[i]==1||fix[j]==1)
    A[i][j]=0;

   else
    A[i][j]=(meansq(u11)-meansq(u_11)-meansq(u1_1)+meansq(u_1_1))
            /(4*beta[i]*beta[j]);

   A[j][i]=A[i][j];}

}



 fprintf(stuff,"\nVariable numbers fixed = ");
 for(k=0;k<DIMS;k++)
  if(fix[k]==1) fprintf(stuff,"%d  ",k+1);
 if(sum==DIMS) fprintf(stuff,"none");
 fprintf(stuff,"\n");


        /* Condense DIMSxDIMS matrix of second partials into */
        /* a smaller (sum x sum), nonsingular matrix for */
        /* calculation purposes. */

 x=0;
 for(i=0;i<sum;i++)
  {while(fix[i+x]==1) x++;
   CON1[i]=b[i+x];
   y=0;
   for(j=0;j<sum;j++)
    {while(fix[j+y]==1) y++;
     CON2[i][j]=A[i+x][j+y];}
   }
```

```
fprintf(stuff,"\n\nMatrix of second partials\n");
for(i=0;i<DIMS;i++)
 {for(j=0;j<DIMS;j++)
   fprintf(stuff,"%10.7lf  ",A[i][j]);
  fprintf(stuff,"\n");}

fprintf(stuff,"\n\nCondensed matrix of second partials\n");
for(i=0;i<sum;i++)
 {for(j=0;j<sum;j++)
   fprintf(stuff,"%10.7lf  ",CON2[i][j]);
  fprintf(stuff,"\n");}

fprintf(stuff,"\n\ndeterminant of condensed matrix of second \n");
fprintf(stuff,"partials is\n");
fprintf(stuff,"%e\n",det(CON2,sum));

inv(CON2,sum);

        /* Multiply CON2's inverse with CON1 */

for(i=0;i<sum;i++)
{u_11[i]=0;
 for(k=0;k<sum;k++)
  u_11[i]=u_11[i]+I[i][k]*CON1[k];}



        /* Reexpand & add to center point. */

for(i=0;i<DIMS;i++)
 u11[i]=v[i];

x=0;
for(i=0;i<sum;i++)
 {while(fix[i+x]==1) x++;
  u11[i+x]=u11[i+x]+u_11[i];}



dummy=meansq(u11);
fprintf(stuff,
        "\n\nMinimum root-mean-square found by matrix method\n");
fprintf(stuff,"%e (in terms of the reference)\n\n",100*sqrt(dummy));


        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */
        /* Variables must be printed out and properly */
        /* identified. */

display:   /*  Display results */

fprintf(stuff,"                  minimum     center pt.  ");
fprintf(stuff,"   beta        change     dist to dbl \n");

t1=meansq(u11);
t2=meansq(v);

for(k=0;k<DIMS;k++)
{fprintf(stuff,"variable %d = %11.5lf  %11.5lf",k+1,u11[k],v[k]);
 fprintf(stuff,"  %11.5lf  %11.5lf",beta[k],u11[k]-v[k]);
```

```
if(t1<t2)
  dtd[k]=sqrt(2*t1/abs(A[k][k]));
else dtd[k]=sqrt(2*t2/(A[k][k]));

if(fix[k]==1) {fprintf(stuff,"       NA\n"); dtd[k]=0;}
else fprintf(stuff,"  %11.8lf\n",dtd[k]);

ch[k]=ul1[k]-v[k];
printf("variable %d = %11.5lf  %11.5lf",k+1,ul1[k],v[k]);
printf("  %11.5lf  %11.5lf",beta[k],ul1[k]-v[k]);
if(fix[k]==1) printf("       NA\n");
else printf("  %11.8lf\n",dtd[k]);}


      /* Follow along the given path */
      /* This is the s-line routine  */
      /* The program finds the minimum along */
      /* The line connecting v and ul1  */

line:
  {for(k=0;k<DIMS;k++)
   {delta[k]=(ul1[k]-v[k])/2;
    mid[k]=v[k]+delta[k];}
   f1=meansq(v);
   f2=meansq(mid);
   f3=meansq(ul1);
   printf("        First                Middle                Last");
   printf("                  Best\n");
   fprintf(stuff,
        "        First                Middle                Last");
   fprintf(stuff,"              Best\n");
   sb2=0;
   sm2=0;
   s2=0;
   for(k=0;k<DIMS;k++)
    {gamma[k]= -(f3-f1)*delta[k]/(2*(f3+f1-2*f2));
     best[k]=mid[k]+gamma[k];
     s2=s2+(ul1[k]-v[k])*(ul1[k]-v[k]);
     sb2=sb2+(best[k]-v[k])*(best[k]-v[k]);
     sm2=sm2+(mid[k]-v[k])*(mid[k]-v[k]);}
   sb=sqrt(sb2/s2);
   sm=sqrt(sm2/s2);
   dtds=sqrt((2*f2)*(0.25)/(f1+f3-2*f2));
   f4=meansq(best);
   if((f4>f1)||(f4>f2)||(f4>3))
     {printf("WARNING!!!  THIS IS NOT A MINIMUM\n");
      fprintf(stuff,"WARNING!!!  THIS IS NOT A MINIMUM\n");}
   printf("s=  %16.11f  %16.11f  %16.11f  %16.5lf\n",0.0,sm,1.0,sb);
   fprintf(stuff,"s=  %16.11f  %16.5lf  %16.11f  %16.5lf\n"
                      ,0.0,sm,1.0,sb);
   printf("f=     %e     %e     %e    %e\n",f1,f2,f3,f4);
   fprintf(stuff,"f=     %e     %e     %e     %e\n",f1,f2,f3,f4);

  for(k=0;k<DIMS;k++)
   {printf("%d  %16.6lf %16.6lf %16.6lf",k+1,v[k],mid[k],ul1[k]);
   printf(" %16.6lf\n",best[k]);
   fprintf(stuff,"%d  %16.6lf   %16.6lf",k+1,v[k],mid[k]);
   fprintf(stuff,"   %16.6lf   %16.6lf\n",ul1[k],best[k]);}

   printf("\n\n");
   fprintf(stuff,"\n\n");

              /*  Is the point Sufficiently close?   */


dis=0;
```

```
  for(k=0;k<DIMS;k++)
   {dis=dis+(best[k]-mid[k])*(best[k]-mid[k]);}
  dis=sqrt(dis);
  if(dis>sqrt(DIMS)*tol)
    c=1.0;
  else c=2.0;


   if((f1<f2)&&(f1<f3)&&(f1<f4))
     {for(k=0;k<DIMS;k++) best[k]=v[k];}
   else if((f2<f1)&&(f2<f3)&&(f2<f4))
     {for(k=0;k<DIMS;k++) best[k]=mid[k];}
   else if((f3<f1)&&(f3<f2)&&(f3<f4))
     {for(k=0;k<DIMS;k++) best[k]=u11[k];}

   for(k=0;k<DIMS;k++)
     u11[k]=best[k];
  }
                /*  Scale Beta Based on Previous Change */
 c=0;
  for(k=0;k<DIMS;k++)
   {c=c+ch[k]*ch[k];}
  c=sqrt(c);
  printf("%e",c);
  if(c<tol) c=0;
  else c=1.0/c;

epsilon=0.0;

if(c!=0.0)
 {epsilon=1.0/c;
  for(k=0;k<DIMS;k++)
   {beta[k]=dtd[k]/c ;
    v[k]=u11[k];}
  goto redo;}
stop:;
}
}
```

```
        /* Function MEANSQ is what is minimized. */

double meansq(v)
double v[DIMS];

{double x,f,g,f2,h,chinew,p,weight;
 int d;

 double force();

  chinew=0;

  x=xlo;
  for(d=0;d<=n;d++)
  {     /*  10th order Newton-Cotes integration */
   weight=0;
   if(d%10==0) weight=32134;
   if((d==0)||(d==n)) weight=16067;
   if((d%10==1)||(d%10==9)) weight=106300;
   if((d%10==2)||(d%10==8)) weight= -48525;
```

```
    if((d%10==3)||(d%10==7)) weight=272400;
    if((d%10==4)||(d%10==6)) weight= -260550;
    if(d%10==5) weight=427368;

    p=force(v,x);
    chinew=chinew+weight*p*p;

    x=x+(xhi-xlo)/n;
   }
    chinew=5*chinew/299376/n;

return(chinew);
}



                /*  Old Routine Used for Minimax */
double max(v)
double v[DIMS];

{double x,f,g,f2,h,max;
 int k;

 double force();

  max=0;           /* set max for retention */

  x=xlo;              /* set x */
  for(k=0;k<=n;k++)    /* x loop */
  {
   if(max<fabs(force(v,x))) max=force(v,x);
   x=x+(xhi-xlo)/n;   /* iterate x */
  }

 return(max);
}



        /* Function FORCE calculates the on-axis force felt by */
        /* a test particle inside the capsule. */
        /* In this form, the capsule consists of a uniform */
        /* cylinder with two sets of compensating rings on the */
        /* ends and flat end caps. */

double force(v,x)
double v[DIMS],x;

{double f,g,f2,g2,h;
 double q,r,L;

 r=RADIUS/LENGTH;   L=LENGTH/2;


        /* Force due to cylinder */

   f=(1/(2*L*L))*pow((1+x/L)*(1+x/L)+4*r*r,-.5);
   f=f-(1/(2*L*L))*pow((1-x/L)*(1-x/L)+4*r*r,-.5);

        /* Force due to ring 1 */

   g=v[1]*(L-x)*pow((L-x)*(L-x)+v[0]*v[0],-1.5);
   g=g-v[1]*(L+x)*pow((L+x)*(L+x)+v[0]*v[0],-1.5);

        /* Force due to ring 2 */
```

```
f2=v[3]*(L-x)*pow((L-x)*(L-x)+v[2]*v[2],-1.5);
f2=f2-v[3]*(L+x)*pow((L+x)*(L+x)+v[2]*v[2],-1.5);

     /* Force due to end caps */

g2=CAPM*(L+x)/((2*r*r*L*L*L)*sqrt(4*r*r+(1+x/L)*(1+x/L)));
g2=g2-CAPM*(L-x)/((2*r*r*L*L*L)*sqrt(4*r*r+(1-x/L)*(1-x/L)));

     /* Total Force */
h=f+g+f2+g2;


return(h);}
```

```
        /* MATRIX is a set of useful functions that can */
        /* invert or calculate the determinant of a square */
        /* matrix. Matrices are passed to the functions */
        /* from other programs. */
        /* The algorithms work for any size matrix, but */
        /* current memory allocation requires that any matrix */
        /* passed to a function be dimensioned as 4x4. */
        /* To change this value, reset DIM to the desired */
        /* value in the line #define DIM 4 and recompile. */

        /* Works as of 12:30 pm Sunday Feb 3, 1991. */

        /* Function INV calculates the inverse of an nxn */
        /* square matrix by reducing it to the identity */
        /* matrix. The same operations are simultaneously */
        /* performed on a matrix that starts out as */
        /* the identity, and that produces the inverse of M. */
        /* The array I[][] that will hold the inverse of M */
        /* must be defined globally by the program that calls */
        /* INV. */

#define DIM 4
#include <stdio.h>

void inv(M,n)
double M[DIM][DIM];
int n;
{int i,j,k,l;
double div,c;
double A[DIM][DIM];
extern double I[DIM][DIM];
FILE *record;

        /* Set up the data file INVFN.REC as a record of */
        /* the inversion. This will contain the matrix to */
        /* be inverted, its inverse, and the product of the */
        /* two, which should be the identity. */

record=fopen("invfn.rec","w");

 fprintf(record,"\nThe matrix M passed to INV is\n\n");

        /* Print M to a data file as a record of the inversion. */

 for(i=0;i<n;i++)
  {for(j=0;j<n;j++)
    fprintf(record," %8.5lf  ",M[i][j]);
   fprintf(record,"\n");}

   /* Store the values of M in A to later verify the inversion. */

for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  A[i][j]=M[i][j];

        /* Assign the identity matrix to I. */

for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  {I[i][j]=0;
   if(i==j) I[i][j]=1;}

        /* Perform Gaussian elimination on M to reduce it to */
        /* the identity matrix. Perform the same operations on */
        /* I to produce inv(M). */
```

```
for(k=0;k<n;k++)
{
for(i=k;i<n;i++)
 {div=M[i][k];
  for(j=0;j<n;j++)
  {if(div==0) break;

   if(i==k)
    {M[i][j]=M[i][j]/div;
     I[i][j]=I[i][j]/div;}

   if(i>k)
    {M[i][j]=M[i][j]-M[k][j]*div;
     I[i][j]=I[i][j]-I[k][j]*div;}
   }
 }
 }

 for(k=n-1;k>=0;k--)
 {
 for(i=k-1;i>=0;i--)
  {div=M[i][k];
   for(j=0;j<n;j++)
    {if(div==0) break;
     M[i][j]=M[i][j]-M[k][j]*div;
     I[i][j]=I[i][j]-I[k][j]*div;}
}}

        /* Write the inverse of M in the data file as a record */
        /* of the inversion. */

 fprintf(record,"\nThe inverse of M is\n\n");

 for(i=0;i<n;i++)
  {for(j=0;j<n;j++)
    fprintf(record," %8.5lf  ",I[i][j]);
   fprintf(record,"\n");}

        /* Multiply A (the original M) and I (the inverse of M) */
        /* to verify that the product is the identity matrix. */
        /* Also store this product in the data file as proof that */
        /* the inversion worked. */

fprintf(record,"\nProduct of the original M and its calculated\n");
fprintf(record,"inverse.\n");
for(i=0;i<n;i++)
 {for(j=0;j<n;j++)
  {c=0;
   for(k=0;k<n;k++)
    c=c+A[i][k]*I[k][j];
   fprintf(record," %15.12lf  ",c);}
  fprintf(record,"\n");}

}



        /* Function DET2 calculates the determinant of a 2x2 */
        /* matrix. This is the last function called in */
        /* calculating the determinant of an nxn matrix */
        /* from function DET. */

double det2(M)
double M[DIM][DIM];
{return(M[0][0]*M[1][1]-M[0][1]*M[1][0]);}
```

```
        /* Function DET calculates the determinant of an nxn */
        /* matrix by cofactor expansion along the first row. */
        /* Secondary determinants are calculated by DET calling */
        /* itself. */

double det(M,n)
double M[DIM][DIM];
int n;
{int i,j,k,l,m;
double determ,guts[DIM][DIM];
double det(),det2();

        /* m=1 if column index number is odd; m=-1 if it is even. */
        /* Determinant is set at zero for later summation. */

m=1;  determ=0;

        /* Go accross the top row & multiply the terms be m */
        /* and the determinant of the sub matrix. */

for(k=0;k<n;k++)
 {l=0;

        /* Define guts[][] to be the sub matrix relevant to */
        /* the M[0][k] term. */

for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  {if(j>=k) l=1;
   if(j<k) l=0;
   guts[i][j]=M[i+1][j+l];}

        /* Sum up the terms in pursuit of the determinant. */
        /* Here either DET or DET2 is called, depending on */
        /* the dimension of guts[][]. */

if(n>3)
 determ=determ+m*M[0][k]*det(guts,n-1);

if(n==3)
 determ=determ+m*M[0][k]*det2(guts);

        /* Change the sign of m for the next iteration. */

m=-m;}

        /* Return the value of the determinant. */

return(determ);}
```

```
/*                          PROGRAM OPT3                                */

     /* WORKS AS OF  May 5, 1992 */
     /* Program finds minimum of a quadratic function */
     /* from a space of dimension DIMS to the real line. */
     /* When changing dimensions, DIMS must be changed in */
     /* the #define DIMS expression, and DIM must be changed */
     /* accordingly in MATRIX6.C, which is a series of  */
     /* subroutines to calculate the inverse and determinant */
     /* of a matrix.    */

     /* Survey search works on any function. */
     /* Quadratic interpolation only works on quadratic functions. */
     /* Three rings, 10th order Newton-Cotes integration */

#include <stdio.h>
#include <time.h>
#include <math.h>
#include "matrix6.c"

        /*  Capsule constants */
        /*  DIMS :  Number of parameters to be optimized  */
        /*  CAPM :  Mass of each end cap (cylinder = 1)   */
        /*  RADIUS : Radius of the cylinder               */
        /*  LENGTH : Length of the cylinder               */

#define DIMS 6
#define CAPM .02
#define RADIUS .4
#define LENGTH 20
#define POW 21

double xhi,xlo;
double I[DIMS][DIMS];
int n,p;

main()
{
        /*      Important Variables     */
        /*   v[] : initial parameter values     */
        /*   beta[] : finite interval to calculate derivatives */
        /*   u11[] : optimum parameter values   */
        /*   A[][] : matrix of 2nd Partials     */
        /*   CON2[][] : Condensed matrix of 2nd partials */
        /*   b[] : matrix of 1st Partials       */
        /*   con1 : Condensed matrix of 1st partials    */
        /*   bestmsq : minimum mean-square found in survey */
        /*   dtd : distance along axis of parameter space  */
        /*         at which the function should be double  */

int i,j,k,l,i2,j2,choice,x,y,sum,fix[DIMS],m[DIMS],o,q;
double chinew,chimin,dummy,dis,tol,c;
double v[DIMS],beta[DIMS];
double u11[DIMS],u_11[DIMS],u1_1[DIMS],u_1_1[DIMS];
double A[DIMS][DIMS],b[DIMS],bestmsq[DIMS];
double CON1[DIMS],CON2[DIMS][DIMS];
double meansq(),det(),max();
double p[DIMS],dtd[DIMS];
double f1,f2,f3,gamma[DIMS],delta[DIMS],mid[DIMS];
double sb,sb2,sm,sm2,f4,best[DIMS],s2;
double factor,dtds,epsilon,t1,t2,ch[DIMS];

void inv();
        /*  Open File opt2.dat and write current time  */

FILE *stuff;
```

```
struct tm *timeptr;
time_t timer;
time(&timer);
timeptr=localtime(&timer);

stuff=fopen("opt3.dat","w");

fprintf(stuff,"%s\n",asctime(timeptr));

printf("**********PROGRAM OPT**********\n");

        /*  Set limits of integration on axis */
        /*  n=number of intervals */

xlo=0; xhi=8; n=80;


        /* fix[i] for i=0,1,2,3 set at zero. Later, if a */
        /* variable k is fixed, fix[k] will be set to 1. */

for(k=0;k<DIMS;k++)
 fix[k]=0;

printf("Which kind of search do you want to perform?\n");
printf("1. Survey search based on mean square\n");
printf("2. Quadratic interpolation of minimum mean square\n");
printf("Enter the number of your choice : ");
scanf("%d",&choice);
printf("\n");

        /* Here, and in the function, are the only places */
        /* where the physical meaning of the variables is */
        /* significant for coding. */

printf("\nvariable    physical meaning\n");
printf("  1            radius1\n");
printf("  2            mass1\n");
printf("  3            radius2\n");
printf("  4            mass2\n");
printf("  5            radius3\n");
printf("  6            mass3\n");


fprintf(stuff,"\nvariable    physical meaning\n");
fprintf(stuff,"  1            radius1\n");
fprintf(stuff,"  2            mass1\n");
fprintf(stuff,"  3            radius2\n");
fprintf(stuff,"  4            mass2\n");
fprintf(stuff,"  5            radius3\n");
fprintf(stuff,"  6            mass3\n");

fprintf(stuff,"\nRadius   = %lf\n",RADIUS);
fprintf(stuff,"Cap mass = %lf\n",CAPM);
fprintf(stuff,"Length   = %lf\n",LENGTH);



if(choice==1)     /*  Begin Survey Search  */
{
printf("Enter the parameters of the search.\n\n");

printf("First, enter the center point.\n");
printf("Type in the coordinates in %d-space\n",DIMS);
printf("all separated by commas :\n");
```

```
        /* THIS MUST BE CHANGED BEFORE COMPILING */
        /* TO CHANGE THE DIMENSION OF THE PROBLEM */

scanf("%lf,%lf,%lf,%lf,%lf,%lf",
        &v[0],&v[1],&v[2],&v[3],&v[4],&v[5]);

printf("\nEnter the spacing between the points.\n");
printf("(along the same format as above)\n");

        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

scanf("%lf,%lf,%lf,%lf,%lf,%lf",
     &beta[0],&beta[1],&beta[2],&beta[3],&beta[4],&beta[5]);


printf("Now enter the number of points ON EACH SIDE \n");
printf("at which you wish to calculate values.\n");
printf("(that is, if you want 3 points on each side,\n");
printf("then a total of 7 values will be calculated along\n");
printf("that particular axis)\n");
printf("enter %d values, each separated by commas\n",DIMS);

        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

scanf("%d,%d,%d,%d,%d,%d",&m[0],&m[1],&m[2],&m[3],&m[4],&m[5]);


fprintf(stuff,
        "Var. 1    Var. 2   Var. 3    Var. 4    Var. 5    Var.6");
fprintf(stuff,"       MS\n");

chimin=10000;



        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */
        /* NEW LOOPS MUST BE CONSTRUCTED IN A NESTED */
        /* FASHION TO INCLUDE NEW DIMENSIONS */

u11[0]=v[0]-m[0]*beta[0];                    /* set var. 1 */
for(i= -m[0];i<=m[0];i++)                     /* var. 1 loop */
 {u11[1]=v[1]-m[1]*beta[1];                  /* set var. 2 */
 for(j= -m[1];j<=m[1];j++)                    /* var .2 loop */
 {u11[2]=v[2]-m[2]*beta[2];                  /* set var. 3 */
  for(k= -m[3];k<=m[2];k++)                    /* var. 3 loop */
  {u11[3]=v[3]-m[3]*beta[3];                 /* set var. 4 */
   for(l= -m[3];l<=m[3];l++)                    /* var. 4 loop */
   {u11[4]=v[4]-m[4]*beta[4];                 /* set var. 5 */
    for (i2= -m[4];i2<=m[4];i2++)               /* var. 5 loop */
    {u11[5]=v[5]-m[5]*beta[5];                /* set var. 6 */
     for (j2= -m[5];j2<=m[5];j2++)               /* var. 6 loop */
{
  chinew=meansq(u11);

  if(chinew<chimin)
   {chimin=chinew;
    bestmsq[0]=u11[0];
    bestmsq[1]=u11[1];
    bestmsq[2]=u11[2];
    bestmsq[3]=u11[3];}

  if(chinew<0.0000000408)
   {fprintf(stuff,"%8.5lf %8.5lf",u11[0],u11[1]);
   fprintf(stuff,"%9.5lf %8.5lf",u11[2],u11[3]);
   fprintf(stuff,"%9.5lf %8.5lf",u11[4],u11[5]);
```

```
   fprintf(stuff,"   %e\n",chinew);}

  u11[5]=u11[5]+beta[5];} /* iterate variable 6 */
  u11[4]=u11[4]+beta[4];} /* iterate variable 5 */
  u11[3]=u11[3]+beta[3];} /* iterate variable 4 */
  u11[2]=u11[2]+beta[2];} /* iterate variable 3 */
  u11[1]=u11[1]+beta[1];} /* iterate variable 2 */
  u11[0]=u11[0]+beta[0];} /* iterate variable 1 */


fprintf(stuff,"\n\nMinimum root-mean-square found by dumb search\n");
fprintf(stuff,"%e\n\n",sqrt(chimin));

fprintf(stuff,"\n\nMaximum force for that geometry\n");
fprintf(stuff,"%e\n\n",max(bestmsq));

fprintf(stuff,"variable 1 = %15.4lf\n",bestmsq[0]);
fprintf(stuff,"variable 2 = %15.4lf\n",bestmsq[1]);
fprintf(stuff,"variable 3 = %15.4lf\n",bestmsq[2]);
fprintf(stuff,"variable 4 = %15.4lf\n",bestmsq[3]);
fprintf(stuff,"variable 5 = %15.4lf\n",bestmsq[4]);
fprintf(stuff,"variable 6 = %15.4lf\n",bestmsq[5]);

time(&timer);
timeptr=localtime(&timer);
fprintf(stuff,"%s\n",asctime(timeptr));
goto stop;
}    /*  End Survey Search  */


if(choice==2)  /*  Begin Quadratic Interpolation  */
{
  printf("Tolerance?");   /* Tolerance for Automatic Runs */
   scanf("%lf",&tol);
   printf("%lf",tol);

printf("Enter the parameters of the search.\n\n");
printf("First, enter the center point of the 4-cube.\n");
printf("Type in radius1, mass1, radius2, mass2\n");


        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

printf("all separated by commas :\n");
scanf("%lf,%lf,%lf,%lf,%lf,%lf",
        &v[0],&v[1],&v[2],&v[3],&v[4],&v[5]);

printf("\nEnter the spacing between the points.\n");
printf("(along the same format as above)\n");


        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */

scanf("%lf,%lf,%lf,%lf,%lf,%lf",
       &beta[0],&beta[1],&beta[2],&beta[3],&beta[4],&beta[5]);

        /* Main interpolation loop */

        /* Set fix[i]'s to determine which variables */
        /* are fixed & subtract total from DIMS to  */
        /* determine how many (sum) are to vary. */


redo:   /* Begin Main Interpolation loop */

sum=DIMS;
```

```
for(k=0;k<DIMS;k++)
 {if(beta[k]==0) fix[k]=1;
  sum=sum-fix[k];}

printf("\n\nWorking...\n");

        /* Calculate Matrix Elements */

for(i=0;i<DIMS;i++)
 for(j=i;j<DIMS;j++)
  {for(k=0;k<DIMS;k++)
   {
        /* set finite difference vectors to center of grid */

   u11[k]=v[k]; u_11[k]=v[k]; u1_1[k]=v[k]; u_1_1[k]=v[k];}


 if(j==i)
  {u11[i]=u11[i]+beta[i];
   u_1_1[i]=u_1_1[i]-beta[i];


        /* If variable is fixed, set derivatives equal to zero, */
        /* otherwise, approximate them numerically. */

   if(fix[i]==1)
    {A[i][i]=0; b[i]=0;}

   else
    {A[i][i]=(meansq(u11)-2*meansq(u_11)+meansq(u_1_1))
            /(beta[i]*beta[i]);
     b[i]=(meansq(u_1_1)-meansq(u11))/(2*beta[i]);}
  }

   if(j!=i)
    {u11[i]=u11[i]+beta[i];
     u11[j]=u11[j]+beta[j];

     u_11[i]=u_11[i]-beta[i];
     u_11[j]=u_11[j]+beta[j];

     u1_1[i]=u1_1[i]+beta[i];
     u1_1[j]=u1_1[j]-beta[j];

     u_1_1[i]=u_1_1[i]-beta[i];
     u_1_1[j]=u_1_1[j]-beta[j];

        /* If variable is fixed, set derivative equal to zero, */
        /* otherwise, approximate it numerically. */

   if(fix[i]==1||fix[j]==1)
    A[i][j]=0;

   else
    A[i][j]=(meansq(u11)-meansq(u_11)-meansq(u1_1)+meansq(u_1_1))
            /(4*beta[i]*beta[j]);

   A[j][i]=A[i][j];}

}



 fprintf(stuff,"\nVariable numbers fixed = ");
 for(k=0;k<DIMS;k++)
  if(fix[k]==1) fprintf(stuff,"%d  ",k+1);
```

```
 if(sum==DIMS) fprintf(stuff,"none");
 fprintf(stuff,"\n");


        /* Condense DIMSxDIMS matrix of second partials into */
        /* a smaller (sum x sum), nonsingular matrix for */
        /* calculation purposes. */

 x=0;
 for(i=0;i<sum;i++)
  {while(fix[i+x]==1) x++;
   CON1[i]=b[i+x];
   y=0;
   for(j=0;j<sum;j++)
    {while(fix[j+y]==1) y++;
     CON2[i][j]=A[i+x][j+y];}
   }



fprintf(stuff,"\n\nMatrix of second partials\n");
for(i=0;i<DIMS;i++)
 {for(j=0;j<DIMS;j++)
   fprintf(stuff,"%10.7lf  ",A[i][j]);
  fprintf(stuff,"\n");}

fprintf(stuff,"\n\nCondensed matrix of second partials\n");
for(i=0;i<sum;i++)
 {for(j=0;j<sum;j++)
   fprintf(stuff,"%10.7lf  ",CON2[i][j]);
  fprintf(stuff,"\n");}

fprintf(stuff,"\n\ndeterminant of condensed matrix of second \n");
fprintf(stuff,"partials is\n");
fprintf(stuff,"%e\n",det(CON2,sum));

inv(CON2,sum);

        /* Multiply CON2's inverse with CON1 */

for(i=0;i<sum;i++)
{u_11[i]=0;
 for(k=0;k<sum;k++)
  u_11[i]=u_11[i]+I[i][k]*CON1[k];}



        /* Reexpand & add to center point. */

for(i=0;i<DIMS;i++)
 u11[i]=v[i];

x=0;
for(i=0;i<sum;i++)
 {while(fix[i+x]==1) x++;
  u11[i+x]=u11[i+x]+u_11[i];}



dummy=meansq(u11);
fprintf(stuff,
        "\n\nMinimum root-mean-square found by matrix method\n");
fprintf(stuff,"%e (in terms of the reference)\n\n",100*sqrt(dummy));
```

```
        /* MUST BE CHANGED WHEN CHANGING DIMENSIONS */
        /* Variables must be printed out and properly */
        /* identified. */

display:   /*  Display results */

fprintf(stuff,"                  minimum     center pt.  ");
fprintf(stuff,"   beta        change     dist to dbl \n");

t1=meansq(u11);
t2=meansq(v);

for(k=0;k<DIMS;k++)
{fprintf(stuff,"variable %d = %11.5lf  %11.5lf",k+1,u11[k],v[k]);
 fprintf(stuff,"  %11.5lf  %11.5lf",beta[k],u11[k]-v[k]);

 if(t1<t2)
   dtd[k]=sqrt(2*t1/abs(A[k][k]));
 else dtd[k]=sqrt(2*t2/(A[k][k]));

 if(fix[k]==1) {fprintf(stuff,"       NA\n"); dtd[k]=0;}
 else fprintf(stuff,"  %11.8lf\n",dtd[k]);

 ch[k]=u11[k]-v[k];
 printf("variable %d = %11.5lf  %11.5lf",k+1,u11[k],v[k]);
 printf("  %11.5lf  %11.5lf",beta[k],u11[k]-v[k]);
 if(fix[k]==1) printf("       NA\n");
 else printf("  %11.8lf\n",dtd[k]);}


        /* Follow along the given path */
        /* This is the s-line routine  */
        /* The program finds the minimum along */
        /* The line connecting v and u11  */

line:
  {for(k=0;k<DIMS;k++)
   {delta[k]=(u11[k]-v[k])/2;
    mid[k]=v[k]+delta[k];}
   f1=meansq(v);
   f2=meansq(mid);
   f3=meansq(u11);
   printf("        First                Middle                Last");
   printf("                 Best\n");
   fprintf(stuff,
        "       First                Middle                Last");
   fprintf(stuff,"              Best\n");
   sb2=0;
   sm2=0;
   s2=0;
   for(k=0;k<DIMS;k++)
    {gamma[k]= -(f3-f1)*delta[k]/(2*(f3+f1-2*f2));
     best[k]=mid[k]+gamma[k];
     s2=s2+(u11[k]-v[k])*(u11[k]-v[k]);
     sb2=sb2+(best[k]-v[k])*(best[k]-v[k]);
     sm2=sm2+(mid[k]-v[k])*(mid[k]-v[k]);}
   sb=sqrt(sb2/s2);
   sm=sqrt(sm2/s2);
   dtds=sqrt((2*f2)*(0.25)/(f1+f3-2*f2));
   f4=meansq(best);
   if((f4>f1)||(f4>f2)||(f4>3))
     {printf("WARNING!!!  THIS IS NOT A MINIMUM\n");
      fprintf(stuff,"WARNING!!!  THIS IS NOT A MINIMUM\n");}
   printf("s=  %16.11f  %16.11f  %16.11f  %16.5lf\n",0.0,sm,1.0,sb);
   fprintf(stuff,"s=  %16.11f  %16.5lf  %16.11f  %16.5lf\n"
```

```
                          ,0.0,sm,1.0,sb);
   printf("f=     %e     %e     %e    %e\n",f1,f2,f3,f4);
   fprintf(stuff,"f=     %e     %e     %e     %e\n",f1,f2,f3,f4);

  for(k=0;k<DIMS;k++)
   {printf("%d  %16.6lf %16.6lf %16.6lf",k+1,v[k],mid[k],u11[k]);
   printf(" %16.6lf\n",best[k]);
   fprintf(stuff,"%d  %16.6lf   %16.6lf",k+1,v[k],mid[k]);
   fprintf(stuff,"   %16.6lf   %16.6lf\n",u11[k],best[k]);}

   printf("\n\n");
   fprintf(stuff,"\n\n");

               /*  Is the point Sufficiently close?   */


 dis=0;
  for(k=0;k<DIMS;k++)
   {dis=dis+(best[k]-mid[k])*(best[k]-mid[k]);}
  dis=sqrt(dis);
  if(dis>sqrt(DIMS)*tol)
    c=1.0;
  else c=2.0;


   if((f1<f2)&&(f1<f3)&&(f1<f4))
     {for(k=0;k<DIMS;k++) best[k]=v[k];}
   else if((f2<f1)&&(f2<f3)&&(f2<f4))
     {for(k=0;k<DIMS;k++) best[k]=mid[k];}
   else if((f3<f1)&&(f3<f2)&&(f3<f4))
     {for(k=0;k<DIMS;k++) best[k]=u11[k];}

   for(k=0;k<DIMS;k++)
     u11[k]=best[k];
  }
               /*  Scale Beta Based on Previous Change */
 c=0;
  for(k=0;k<DIMS;k++)
   {c=c+ch[k]*ch[k];}
  c=sqrt(c);
  printf("%e",c);
  if(c<tol) c=0;
  else c=1.0/c;

epsilon=0.0;

if(c!=0.0)
 {epsilon=1.0/c;
  for(k=0;k<DIMS;k++)
   {beta[k]=dtd[k]/c ;
    v[k]=u11[k];}
  goto redo;}
stop:;
}
}
```

```
        /* Function MEANSQ is what is minimized. */

double meansq(v)
```

```
double q,r,L;

r=RADIUS/LENGTH;    L=LENGTH/2;


        /* Force due to cylinder */

  f=(1/(2*L*L))*pow((1+x/L)*(1+x/L)+4*r*r,-.5);
  f=f-(1/(2*L*L))*pow((1-x/L)*(1-x/L)+4*r*r,-.5);

        /* Force due to ring 1 */

  g=v[1]*(L-x)*pow((L-x)*(L-x)+v[0]*v[0],-1.5);
  g=g-v[1]*(L+x)*pow((L+x)*(L+x)+v[0]*v[0],-1.5);

        /* Force due to ring 2 */

  f2=v[3]*(L-x)*pow((L-x)*(L-x)+v[2]*v[2],-1.5);
  f2=f2-v[3]*(L+x)*pow((L+x)*(L+x)+v[2]*v[2],-1.5);

        /* Force due to ring 3 */

  f3=v[5]*(L-x)*pow((L-x)*(L-x)+v[4]*v[4],-1.5);
  f3=f3-v[5]*(L+x)*pow((L+x)*(L+x)+v[4]*v[4],-1.5);

        /* Force due to end caps */

  g2=CAPM*(L+x)/((2*r*r*L*L*L)*sqrt(4*r*r+(1+x/L)*(1+x/L)));
  g2=g2-CAPM*(L-x)/((2*r*r*L*L*L)*sqrt(4*r*r+(1-x/L)*(1-x/L)));

        /* Total Force */
  h=f+g+f2+g2+f3;


return(h);}
```

```
double v[DIMS];

{double x,f,g,f2,h,chinew,p,weight;
 int d;

 double force();

  chinew=0;

  x=xlo;
  for(d=0;d<=n;d++)
  {     /*  10th order Newton-Cotes integration */
   weight=0;
   if(d%10==0) weight=32134;
   if((d==0)||(d==n)) weight=16067;
   if((d%10==1)||(d%10==9)) weight=106300;
   if((d%10==2)||(d%10==8)) weight= -48525;
   if((d%10==3)||(d%10==7)) weight=272400;
   if((d%10==4)||(d%10==6)) weight= -260550;
   if(d%10==5) weight=427368;

   p=force(v,x);
   chinew=chinew+weight*p*p;

   x=x+(xhi-xlo)/n;
  }
   chinew=5*chinew/299376/n;

return(chinew);
}



                /*  Old Routine Used for Minimax */
double max(v)
double v[DIMS];

{double x,f,g,f2,f3,h,max;
 int k;

 double force();

  max=0;           /* set max for retention */

  x=xlo;              /* set x */
  for(k=0;k<=n;k++)    /* x loop */
  {
   if(max<fabs(force(v,x))) max=force(v,x);
   x=x+(xhi-xlo)/n;   /* iterate x */
  }

 return(max);
}



        /* Function FORCE calculates the on-axis force felt by */
        /* a test particle inside the capsule. */
        /* In this form, the capsule consists of a uniform */
        /* cylinder with two sets of compensating rings on the */
        /* ends and flat end caps. */

double force(v,x)
double v[DIMS],x;

{double f,g,f2,g2,h;
```

```
/*                    PROGRAM OPTCONT.C                              */

/*  This modification of the opt program treats as its               */
/*  the masses distributed as Taylor series along the                */
/*  and ends of the cylindrical satellite, and the mass              */
/*  of an end ring.  The function is exactly quadratic,              */
/*  so no survey search, s-line, etc. are needed.                    */
/*  Matrix routines are included, so no need to #include             */

#include <stdio.h>
#include <time.h>
#include <math.h>

/*      Capsule Dimensions & parameters                              */
/*  DIM : number of parameters to be optimized                       */
/*  POLDIM : Number of powers of z                                   */
/*  ENDDIM : Number of powers of rho (end caps)                      */
/*  Note: the first z parameter is to the second power               */
/*  while the first rho is to the zero power.                        */

#define DIM 10
#define POLDIM 6
#define ENDDIM 3
#define RADIUS 0.5
#define LENGTH 20
#define xhi 9.99
#define xlo 0
#define INTPTS 1000
#define POW 32     /* make sure matrices are big enough  */

extern double I[DIM][DIM];
main()
{
int c,i,j,k,l,x,y,sum,fix[DIM],m[DIM],o,q;
double tt,chinew,chimin,dummy,scale;
double v[DIM],beta[DIM],A[DIM][DIM],b[DIM],bestmsq[DIM];
double u11[DIM],u_11[DIM],u1_1[DIM],u_1_1[DIM];
double con1[DIM],con2[DIM][DIM],p[DIM],dtd[DIM];
double force(),meansq(),det(),det2();
double f1,f2,f3,gamma[DIM],delta[DIM],mid[DIM];
double sb,sb2,sm,sm2,f4,best[DIM],s2;
double factor,dtds,epsilon,t1,t2;
double fact[POW],pos;
int xpon[POW];

/*                      Important variables                          */
/*  v[] : initial parameter values (v[0]=ring)                       */
/*  beta[] : finite difference for derivatives                       */
/*  u11 : optimum parameter values                                   */
/*  A[][], CON2 : matrix/condensed of second partials                */
/*  b[],CON1 : matrix/condensed of first partials                    */
/*  dtd : distance along axis in parameter space for                 */
/*      which force is doubled                                       */
/*  Note, many of the declared variables are old and                 */
/*      are not used in the program                                  */

void inv();

        /* open optcont.dat and write time      */

FILE *stuff;

struct tm *timeptr;
time_t timer;
time(&timer);
```

```
timeptr=localtime(&timer);

stuff=fopen("optcont.dat","w");

fprintf(stuff,"%s\n",asctime(timeptr));

printf("********** OPT **********\n");

{

for(k=0;k<DIM;k++)
    fix[k]=0;

/* define factorials -- to be used for binomial expansion */
fact[0]=1;
for(k=1;k<POW;k++)
   fact[k]=k*fact[k-1];

/* set initial values to zero   */

v[0]=0.0;   beta[0]=0.01;    xpon[0]=0;
fprintf(stuff,"variable 0 = ring mass");
for(k=1;k<POLDIM+1;k++)
 {v[k]=0.0; beta[k]=0.01; xpon[k]=k*2;
  fprintf(stuff,"variable %d = mass as z^%d\n",k,2*k);}
for(k=POLDIM+1;k<DIM;k++)
 {v[k]=0.0; beta[k]=0.01; xpon[k]=(k-POLDIM-1)*2;
  fprintf(stuff,"variable %d = mass as rho^%d\n",k,xpon[k]);}

for(k=0;k<DIM;k++)
 printf("#=%d  v=%f  beta=%f  xpon=%d\n",k,v[k],beta[k],xpon[k]);

sum=DIM;
for(k=0;k<DIM;k++)
  {if(beta[k]==0) fix[k]=1;
   sum=sum-fix[k];}

printf("\n\nWorking...\n");

        /* Calculate Matrix Elements */

for(i=0;i<DIM;i++)
 for(j=i;j<DIM;j++)
  {for(k=0;k<DIM;k++)
   {    /* Set finite difference vectors to center of grid */
    u11[k]=v[k]; u_11[k]=v[k]; u1_1[k]=v[k]; u_1_1[k]=v[k];}

if(j==i)
 {u11[i]=u11[i]+beta[i];
  u_1_1[i]=u_1_1[i]-beta[i];

        /* If variable is fixed, set derivatives equal to zero, */
        /* otherwise, approximate them numerically. */

  if(fix[i]==1)
   {A[i][i]=0; b[i]=0;}

  else
   {A[i][i]=(meansq(u11,fact,xpon)-2*meansq(u_11,fact,xpon)
            +meansq(u_1_1,fact,xpon))/(beta[i]*beta[i]);
    b[i]=(meansq(u_1_1,fact,xpon)-meansq(u11,fact,xpon))/(2*beta[i]);}
 }

if(j!=i)
 {u11[i]=u11[i]+beta[i];
  u11[j]=u11[j]+beta[j];
```

```
  u_11[i]=u_11[i]-beta[i];
  u_11[j]=u_11[j]+beta[j];

  u1_1[i]=u1_1[i]+beta[i];
  u1_1[j]=u1_1[j]-beta[j];

  u_1_1[i]=u_1_1[i]-beta[i];
  u_1_1[j]=u_1_1[j]-beta[j];

        /* If variable is fixed, set derivative equal to zero, */
        /* otherwise, approximate it numerically. */

if(fix[i]==1||fix[j]==1)
  A[i][j]=0;

else
  A[i][j]=(meansq(u11,fact,xpon)-meansq(u_11,fact,xpon)
           -meansq(u1_1,fact,xpon)+meansq(u_1_1,fact,xpon))
         /(4*beta[i]*beta[j]);

  A[j][i]=A[i][j];}
  printf("%d   %d   %lf\n",i,j,A[i][j]);
}


fprintf(stuff,"\nVariable numbers fixed = ");
for(k=0;k<DIM;k++)
 if(fix[k]==1) fprintf(stuff,"%d  ",k+1);
 if(sum==DIM) fprintf(stuff,"none");
fprintf(stuff,"\n");

        /* Condense DIM*DIM matrix of second partials into */
        /* a smaller sum*sum nonsingular matrix for */
        /* calculation purposes */

x=0;
for(i=0;i<sum;i++)
 {while(fix[i+x]==1) x++;
  con1[i]=b[i+x];
  y=0;
  for(j=0;j<sum;j++)
   {while(fix[j+y]==1) y++;
    con2[i][j]=A[i+x][j+y];}
 }


inv(con2,sum);

        /* Multiply con2's inverse with con1 */

for(i=0;i<sum;i++)
{u_11[i]=0;
 for(k=0;k<sum;k++)
  u_11[i]=u_11[i]+I[i][k]*con1[k];}


        /* Re-expand and add to center point. */

for(i=0;i<DIM;i++)
 u11[i]=v[i];

x=0;
for(i=0;i<sum;i++)
 {while(fix[i+x]==1) x++;
  u11[i+x]=u11[i+x]+u_11[i];}
```

```
dummy=meansq(u11,fact,xpon);
fprintf(stuff,"\n\nMinimum root-mean-square found by matrix method\n");
fprintf(stuff,"%e\n",10000*sqrt(dummy));
fprintf(stuff,"(percent of reference force)\n\n");

        /* Must be changed when changing dimensions */
        /* Variables must be printed out and properly */
        /* identified. */

display:


t1=meansq(u11,fact,xpon);
t2=meansq(v,fact,xpon);

for(k=0;k<DIM;k++)
{       fprintf(stuff,"variable %d = %10.8e ",k+1,u11[k]);
        printf("variable %d = %10.8e ",k+1,u11[k]);

        /* Let dist to dbl be that for the lowest point */

if(t1<t2)
  tt=t1;
else
  tt=t2;

dtd[k]=sqrt(fabs((2*tt)/(A[k][k])));

if(fix[k]==1)
 {fprintf(stuff,"       NA\n");
         printf("       NA\n");}
else
 {fprintf(stuff,"  %11.8e\n",dtd[k]);
         printf("  %11.8e\n",dtd[k]);}
}

fprintf(stuff,"\n");
printf("mean-squared force is %e",t1);
fprintf(stuff,"mean-squared force is %e\n",t1);

/*  Write force values for plotting purposes    */
pos=0.0;
for(j=0;j<500;j++)
 { fprintf(stuff,"%f,%e\n",pos,force(u11,pos,fact,xpon));
   pos=pos+.02;
 }
}
}


        /* Function Meansq is what is minimized */

double meansq(v,fact,xpon)
double v[DIM],fact[POW];
int xpon[DIM];
{double diff,div,x,f,g,f2,h,chinew,p,weight;
 int d;

double force();

chinew=0;  x=xlo;

        /* 10th order Newton-Cotes integration */
d=0;
```

```
for(d=0;d<=INTPTS;d++)
 {
  weight=0;
  if(d%10==0) weight=32134;
  if(d==0)  weight=16067;
  if(d==INTPTS)  weight=16067;
  if((d%10==1)||(d%10==9))  weight=106300;
  if((d%10==2)||(d%10==8))  weight= -48525;
  if((d%10==3)||(d%10==7))  weight=272400;
  if((d%10==4)||(d%10==6))  weight= -260550;
  if(d%10==5) weight=427368;

  p=force(v,x,fact,xpon);
  chinew=chinew+weight*p*p;

  diff=xhi-xlo;
  div=diff/INTPTS;
  x=x+div;
 }

 chinew=chinew*5/INTPTS/299376;
return chinew;
}



        /* Function Force calculates the on-axis force felt by */
        /* A test particle inside the capsule. */
        /* In this form, the capsule has variable density along */
        /* its length and on its ends.  */

double force(v,x,fact,xpon)
double v[DIM],x,fact[POW];
int xpon[DIM];
{int k,j,big,top,jmax;
 double f,g,f2,g2,h,coef[POW],prod;
 double endrec[POW],endrec2[POW],rl;
 double q,r,L,rad2,rl,r2,recint[POW];
 double polyint[POW+1],massint[POW],pint[POW];
 double endforce[ENDDIM],endint[POW],endint2[POW],endp[POW],endn[POW];
 double endp2[POW],endn2[POW],rlmx,rlpx;


rl=RADIUS/LENGTH;
L=LENGTH/2;
r=RADIUS;
rad2=RADIUS*RADIUS;
rl=sqrt(r*r +(L-x)*(L-x));
r2=sqrt(r*r+(L+x)*(L+x));

        /* Force due to cylinder */

  f=(1/(2*L*L))*pow((1+x/L)*(1+x/L)+4*rl*rl,-0.5);
f=f-(1/(2*L*L))*pow((1-x/L)*(1-x/L)+4*rl*rl,-0.5);

        /* Force due to Ring 1 */

g=0.0;
  g=v[0]*(L-x)*pow((L-x)*(L-x)+rad2,-1.5);
g=g-v[0]*(L+x)*pow((L+x)*(L+x)+rad2,-1.5);

/* Force due to polynomial */

for(j=1;j<POLDIM+1;j++)
  {coef[j] = v[j]*(xpon[j]+1)/pow(L,xpon[j]+1);
   }
```

```
big=0;
for(j=1;j<POLDIM+1;j++)
  {if(big<xpon[j])
   {big=xpon[j];
    jmax=j;}}

/* Find Recursive Integrals -- RECINT */
recint[0]=log(r1+L-x)-log(r2-L-x);
recint[1]=r1-r2;

for(k=2;k<big+1;k++)
 {recint[k]=(pow(L-x,k+1)+rad2*pow(L-x,k-1))/(k*r1);
  recint[k]=recint[k]-(pow(-(L+x),k+1)+rad2*pow(-(L+x),k-1))/(k*r2);
  recint[k]=recint[k]-(rad2*(k-1)/k)*recint[k-2];
 }

/* Find Polynomial Integrals, POLYINT */
polyint[0]=(L-x)/(rad2*r1)+(L+x)/(rad2*r2);
polyint[1]= -1/r1+1/r2;

for(k=2;k<big+2;k++)
   polyint[k]=(-pow(L-x,k-1)/r1)+(pow(-(L+x),k-1)/r2)+(k-1)*recint[k-2];

for(k=1;k<POLDIM+1;k++)
 {massint[xpon[k]]=0;
  for(j=0;j<xpon[k]+1;j++)
    {prod=fact[xpon[k]]/(fact[j]*fact[xpon[k]-j])
          *pow(x,xpon[k]-j)*polyint[j+1];
     massint[xpon[k]]=massint[xpon[k]]+prod;}
  }

f2=0;
for(j=1;j<POLDIM+1;j++)
  f2=f2+coef[j]*massint[xpon[j]];

/* Force due to end caps */

rlmx=r1+(L-x);
rlpx=r2+(L+x);

endint[1]=r*r/(r1*(L-x)*rlmx);
endint2[1]=r*r/(r2*(L+x)*rlpx);

endint[3]=pow(r,4)/(r1*pow(rlmx,2));
endint2[3]=pow(r,4)/(r2*pow(rlpx,2));

endint[5]=pow(r,6)/(3*r1*pow(rlmx,3))*(r1+3*(L-x));
endint2[5]=pow(r,6)/(3*r2*pow(rlpx,3))*(r2+3*(L+x));

endint[7]=pow(r,8)/(5*r1*pow(rlmx,4))*(r1*r1+4*r1*(L-x)+5*(L-x)*(L-x));
endint2[7]=pow(r,8)/(5*r2*pow(rlpx,4))*(r2*r2+4*r2*(L+x)+5*(L+x)*(L+x));

endint[9]=pow(r,10)/(35*r1*pow(rlmx,5))*(5*pow(r1,3)+25*r1*r1*(L-x)
                    +47*r1*pow(L-x,2)+35*pow(L-x,3));
endint2[9]=pow(r,10)/(35*r2*pow(rlpx,5))*(5*pow(r2,3)+25*r2*r2*(L+x)
                    +47*r2*pow(L+x,2)+35*pow(L+x,3));

g2=0.0;

for(j=0;j<ENDDIM;j++)
 {endforce[j]=(xpon[POLDIM+j+1]+2)*v[POLDIM+j+1]/pow(r,xpon[POLDIM+j+1]+
2)
                *(-(L+x)*endint2[xpon[POLDIM+j+1]+1]+
                   (L-x)*endint[xpon[POLDIM+j+1]+1]);
 g2=g2+endforce[j];
```

```
 }

        /* Total Force */

h = f + g + f2 + g2;
return(h);}


void inv(M,n)
double M[DIM][DIM];
int n;
{int i,j,k,l;
double div,c,A[DIM][DIM];
extern double I[DIM][DIM];

FILE *record;


record=fopen("invfn.rec","w");

 fprintf(record,"\nThe matrix M passed to INV is\n\n");

for(i=0;i<n;i++)
 {for(j=0;j<n;j++)
  fprintf(record," %8.5lf  ",M[i][j]);
  fprintf(record,"\n");}

for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  A[i][j]=M[i][j];

for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  {I[i][j]=0;
   if(i==j) I[i][j]=1;}

for(k=0;k<n;k++)
{for(i=k;i<n;i++)
  {div=M[i][k];
   for(j=0;j<n;j++)
   {if(div==0) break;

    if(i==k)
     {M[i][j]=M[i][j]/div;
      I[i][j]=I[i][j]/div;}

    if(i>k)
     {M[i][j]=M[i][j]-M[k][j]*div;
      I[i][j]=I[i][j]-I[k][j]*div;}
    }
   }
 }

for(k=n-1;k>=0;k--)
 {for(i=k-1;i>=0;i--)
  {div=M[i][k];
   for(j=0;j<n;j++)
    {if(div==0) break;
     M[i][j]=M[i][j]-M[k][j]*div;
     I[i][j]=I[i][j]-I[k][j]*div;}
   }
  }


fprintf(record,"\nThe inverse of M is\n\n");
```

```
for(i=0;i<n;i++)
 {for(j=0;j<n;j++)
  fprintf(record," %8.5lf  ",I[i][j]);
  fprintf(record,"\n");}


fprintf(record,"\nProduct of the original M and its calculated\n");
fprintf(record,"inverse.\n");
for(i=0;i<n;i++)
 {for(j=0;j<n;j++)
  {c=0;
   for(k=0;k<n;k++)
    c=c+A[i][k]*I[k][j];
   fprintf(record," %15.12lf  ",c);}
  fprintf(record,"\n");}
}



double det2(M)
double M[DIM][DIM];
{return(M[0][0]*M[1][1]-M[0][1]*M[1][0]);}



double det(M,n)
double M[DIM][DIM];
int n;
{int i,j,k,l,m;
 double determ,guts[DIM][DIM];
 double det(),det2();

m=1;  determ=0;

for(k=0;k<n;k++)
 {l=0;

  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
    {if(j>=k) l=1;
     if(j<k) l=0;
     guts[i][j]=M[i+1][j+1];}

if(n>3)
 determ=determ+m*M[0][k]*det(guts,n-1);

if(n==3)
 determ=determ+m*M[0][k]*det2(guts);

m= -m;

return(determ);}
}
```