Development of software for the computation of the properties of electrostatic electro-optical devices via both the direct ray tracing and paraxial approximation techniques.

MPhys Final Year Project Dissertation by Andrew Jackson

Abstract: This project concerns the development of software designed to simulate electro-optical devices; the potential being calculated via the finite element method, and the optical behaviour determined by paraxial and direct ray racing. The direct ray tracing technique, previously only used for finite difference grids, has been developed by using new interpolation techniques (to cope with the variability in mesh point distribution inherent in the finite element technique), and also by testing various different integration algorithms under these conditions. The new software has been found to be act as a useful tool for visualisation of lens behaviour, and has been found to give reasonable quantitative results for the predicted lens properties in comparison to the paraxial approximation and to the results published in the literature for the standardised two-tube electron lens.

1. Introduction

Charged particle optics forms an important part of modern physical research, and the simulation of such devices forms an important part of their development. The computational techniques used in the software written to assist the design process can be broken down into two parts. Firstly, the determination of the electric and/or magnetic fields inside the device (via the boundary element[1,11], finite difference[1,3,4,5] or finite element[1,6,9] methods), and secondly the simulation of the trajectories of charged particles through those fields (via the paraxial approximation[1,5,9] or by direct ray tracing[1,3,4]). While the existing software is capable of producing reliable and accurate data in most cases, it is generally difficult to use and restrictive in the kinds of behaviour it can simulate. The difficulty for the user arises from the fact that the software usually requires a grid or mesh in order to find the electric and/or magnetic fields, and that this mesh must be completely defined by the user (as opposed to implementing any form of automatic mesh generation). The restriction on the kind of behaviour to be simulated is a consequence of the dependance of most of the software on the paraxial approximation, which while accurate for particle trajectories near the optical axis, cannot simulate the particle motion near the boundaries of the device.

This project concerned the development of a user-friendly piece of software designed to allow the finite element solution of the electric field of any rotationally symmetric electrostatic electro-optical device (assisted by semi-automatic mesh generation), and the simulation of the electron trajectories inside the

device both under the paraxial approximation and by direct ray tracing. Before this project began a significant amount of code had already been written, but while the electric field solution and paraxial ray tracing parts of the code were thought to be working reasonably well, the code designed to carry out the direct ray tracing was working somewhat inaccurately, leading to the prediction of clearly non-physical behaviour. Thus, the aim of this project was primarily to re-write the direct ray tracing code to give more realistic behaviour and more accurate results, and also to check the electric field and paraxial ray solutions produced by the original code. In order to ascertain the accuracy of the code, results from the different techniques can be compared with the published results of others, for example the predicted potential and lensing properties of a standard configuration two-tube electron lens[3,5,11]. Once the direct ray tracing results have been compared with those of the paraxial approximation and the theoretical and experimental results of others, then the validity of this implementation of the direct ray technique can be determined, it's weaknesses noted and possibilities for further development can be proposed.

2. Theory

2.1 Physcial Principles

The software under development concerns only electrostatic devices, and so the field problem reduces to the solution of the Laplace equation for the electrostatic potential:

$$\subseteq^2 V = 0 \tag{1}$$

This simplification, coupled with the rotational symmetry of the devices we wish to investigate, means that the software needs only to consider the axial plane in order to completely describe the behaviour of the system. In this way the problem becomes the determination the two-dimensional potential V(r,z), such that:

$$\frac{\delta^2 V(r,z)}{\delta r^2} + \frac{1}{r}\frac{\delta V(r,z)}{\delta r} + \frac{\delta^2 V(r,z)}{\delta z^2} = 0 \tag{2}$$

Which is the expanded form of the Laplace equation in cylindrical polar coordinates for the axial plane. The potential is thus defined by the boundary conditions of the system, in other words, by the charged plates which make up the electrostatic electro-optical device being simulated. Once the potential has been found, then the electric field is defined by the gradient of the potential:

$$\mathbf{E} = -\subseteq V \tag{3}$$

Which for this two-dimensional problem is equivalent to:

$$E_r = -\frac{\delta V}{\delta r} \qquad E_z = -\frac{\delta V}{\delta z} \tag{4}$$

Direct ray tracing can now be carried out by calculating the trajectories of electrons as they move through the device, their motion being governed by the Lorentz force:

$$\frac{d}{dt}(m\mathbf{v}) = e\mathbf{E}(\mathbf{r}) \tag{5}$$

Where the magnetic field component has been neglected. For this two dimensional problem, we find:

$$\frac{d}{dt}(mv_r) = eE_r(r,z) \qquad \frac{d}{dt}(mv_z) = eE_z(r,z) \tag{6}$$

Thus the high accuracy integration of the Newtonian equations of motion of an electron being acted on by the Lorentz force, with a range of initial conditions (position and velocity), can be used to investigate the properties of the electro-optical device.

It should be noted that this formulation of the problem is not complete and does place some restrictions on the kinds of behaviour being simulated. Firstly, the behaviour of a beam of electrons is calculated by the simulation of single electron trajectories through the electrostatic field, and so this approach cannot take the space charge effect into account. Fortunately, this effect is only significant for relatively high charge densities, and electron beams of sufficient concentration only occur in some electron guns. Secondly, the equations of motion are classical, and so relativistic effects are not taken into account. However, these effects only become noticable for beam energies greater than 1keV[8, pp 91], and so we only consider devices which function at energies below this level.

2.2 The Paraxial Approximation

Much of the work on electron optical devices makes use of the concepts and terminology of ray optics, as these concepts provide a concise way of describing the behaviour of a given device. The basis of ray optics theory is the paraxial approximation, and added on to this is the theory of aberrations which accounts for the deviations from first order (paraxial) behaviour.

The paraxial approximation[1,9,7] assumes that the electrons remain close to the optical axis of the lens, and this allows the variation of the field perpendicular to the optical axis to be simplified. By using this simplification we find we only need to know the electric field along the optical axis to carry out our calculations. However, when the electrons are not moving close to the axis, then the basic approximation begins to fail and aberrations start to form. Although there are various types of aberration, especially for non-rotationally symmetric devices, for our purposes the most important is spherical aberration.

Spherical aberration occurs when a change in the angle at which the electron beam enters the lens causes the focal point of the beam to move along the optical axis. Figure 1 compares the idealised paraxial approximation behaviour with the behaviour of the more realistic case of spherical aberration for a plane lens. From this diagram it can be seen that spherical aberration alters the effective focal length of the lens, forming not a point focus, but a circle of least confusion (or spot). Thus the paraxial approximation has to be modified in order that spherical aberration may be taken into account. This

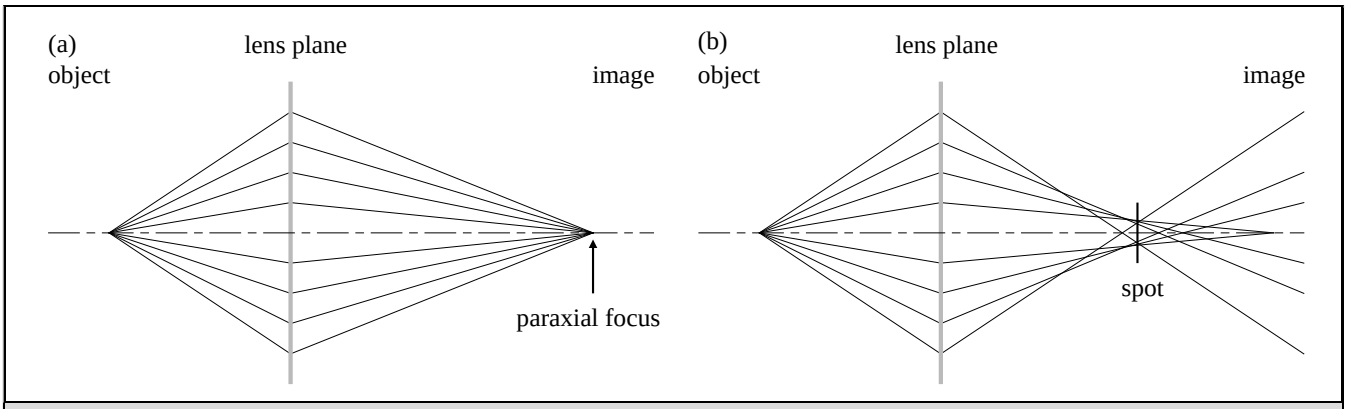

Figure 1: Focussing properties of a plane lens demonstrating (a) idealised paraxial behaviour and (b) spherical abberation, showing the position of the circle of least confusion, or spot.

behaviour is best described in terms of an aberration coefficient such that:

$$\Delta r = MC_s \theta^3 \quad (7)$$

Where M is the linear magnification, and Δr is the transverse displacement at the paraxial image plane of a ray which leaves the point object at an angle θ to the axis[11]. The scaling factor C_s is known as the third order aberration coefficient. However, when carrying out direct ray tracing, it is more convenient to use the form:

$$z_f = z_{fp} - C_s \tan^2\alpha \quad (8)$$

Where z_{fp} is the paraxial focal length, and where z_f is the focal point and α is the angle of the calculated electron trajectory as it crosses the optical axis[4]. The fifth order abberation term has been omitted as this relatively small effect is negligible whilst this piece of software is under development. It should be noted that the electron optical lenses considered here generally behave more like thick lenses than thin lenses, and so the focal length and aberration coefficents will depend on whether the rays enter the lens from the left or right hand side of the lens.

While spherical aberration is by far the most prominent, it should also be noted that two other aberrations can become significant in some cases. The first is relativistic aberration, where the focal point moves as a consequence of the relativistic mass of the particle, and the second is chromatic aberration, where the fact that any charged particle beam will have some variation in the velocities of each of the particles means that the focal point becomes blurred. Relativistic aberration is significant only for devices using relatively high potentials, and as noted above, we can ignore these cases and still produce useful results. Unfortunately, the same cannot generally be said for chromatic aberration, because for the focussing properties of any device using a small aperture chromatic aberration will dominate over spherical aberration[12]. However, the effect does not significantly affect the focal properties of devices using weaker potentials, and so for the time being we ignore the chromatic aberration in the direct ray calculation.

2.3 Examples of Electro-Optical Devices

The simplest electro-optical device is the two-tube electrostatic electron lens, and numerous examples of the predicted properties of this standardised form of this device have been published in the past[3,5,11]. The device is composed of two cylindrical electrodes of identical diameter, with a small gap between them (see figure 2). In order to ensure the device has a standard definition, the parameter for the gap, g, and for the lengths of the two cylinders, L_1 and L_2, are all expressed in terms of the lens diameter D. The properties of the device can then tabulated for a range of g/D, and for a range of potential differences, V_2/V_1.

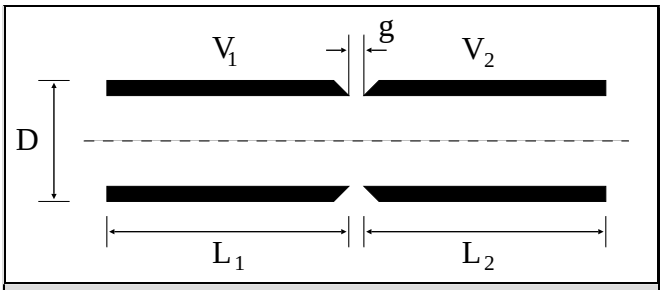

Figure 2: Schematic representation of a simple two-tube electrostatic electron lens.

The work of Natali et al[3] provides an excellent example of the high accuracy computation of the potential, and so can be used to check the finite element code, whereas the papers by Liu & Ximen[5] and Read et al[11] provide tabulated values for the focal length and spherical abberation coefficients, and so can be used to check the paraxial and direct ray tracing.

3. Method & Development

As mentioned in the introduction, a significant amount of code had already been written when this project began. This meant that much time was spend simply getting used to the code, understanding the way in which devices are describe and how to use and adapt the user interface. Also, as the main aim of this project was to improve the direct ray tracing code, only a basic outline of the finite element and paraxial approximation ray tracing code will be given here. However, the following breakdown of the various techniques that have been incorporated into the code will follow the structure of the program and also of it's development.

3.1 Method for the Finite Element Solution of the Electrostatic Potential

When we wish to simulate a device using this software, we must first define the physical space which the device occupies so that it may be broken up into a finite element mesh, and then define the potentials that lie along the boundaries of that mesh. First we note that the symmetry of the problem means that given the rotational axis at r=0, we only need define the problem for r>0, as V(-r,z) must be equal to V(r,z). From this geometrical basis, the structure of the device is broken down to a set of nodes, which when entered into the program can be joined up to form a set of (usually quadrilateral)

polygons covering the domain of the device. Then, an automatic mesh generation program is used to break each polygon up into a set of triangular elements according to user specified information on how fine the mesh should be, and on how the mesh within each polygon should be graded (ie whether the mesh should be uniform of whether it should become finer towards one side or corner).

In order to solve this finite element problem, the potential must be specified along the boundaries of the mesh. These boundaries fall into one of two catagories, Neumann and Dirichlet. The optical axis (r=0) represents a Neumann boundary, a where the gradient of the potential across the boundary is known (due to symmetry, the gradient must be zero). All the other boundaries are Dirichlet boundaries, where the value of the potential is known. A set of electrodes, each of a given potential, is entered by the user, along with any gaps (in which the program assumes the potential to be varying linearly with z). This information, combined with the left and right hand end potentials, completely defines the problem.

The finite element solution then proceeds by the solution of a matrix problem (ie a set of linear simultaneous equations defined by the mesh and it's boundaries). In order to make the solution of the matrix problem more efficient, the mesh points are re-numbered so that the matrix becomes tri-diagonal in form. For more details on the nature of the finite element method, see Kikuchi[10].

Once the potential has been found, the optical properties can be calculated by tracing a set of electrons thought the potential, from a point electron source with a user specified position, energy and angular range. This calculation was carried out by the subroutine TRAJEC, and the development of the program consisted of alterations and modifications to this part of the code (supplied in appendix B).

3.2 Ray Tracing Via The Paraxial Approximation

This part of the code creates a cubic spline of the axial potential for interpolation purposes (subroutine AXPOT), and then uses this potential V(z) to find for the paraxial trajectories by solving the differential equation describing the first order properties of the device, as derived from the theory of Hamiltonian optics (in AXPATHS). The exact form of this approach in not considered here (see refs. [1,4,5,9,12]), and it is sufficient simply to note that the paraxial approximation plots the calculated trajectories on a plot of the lens, and reports the focal point position to the user, along with the third order spherical and first order chromatic aberration coefficients (PARAXABBERS, ABBERS)

3.3 Direct Ray Tracing

While the finite difference and paraxial approximation code have undergone no development during this project, the direct ray tracing code (controlled by the PATHS routine) has been altered many times, with many different algorithms being tested. The direct ray tracing calculation breaks down into three main tasks. The accurate interpolation of the finite element mesh, the differentiation of the potential to

find the electric field, and the integration of the equations of motion of the electron. The following three sections addresses each of these tasks by explaining the initial state of the code, and then covering the other techniques which have been implemented.

3.3.1 Interpolation of the potential mesh:

In order to solve the equations of motion for the electron in this electrostatic potential, it is necessary to be able to calculate the potential at any point, V(z,r), in other words to find the value of the potential by interpolation inside each mesh element. While this poses no problem in a finite difference calculation (where the mesh points follow a simple geometrical form, eg cartesian, for which high accuracy interpolation is well understood[1,3,4,5]), the finite element method can allow any arrangement of mesh points, and so while it is more flexible in terms of the kinds of device geometry it can accurately represent, it requires that the interpolation technique is equally as flexible. It should be noted that of all the published work that has been examined, none have used direct ray tracing in combination with the finite element approach, presumably because of the difficulties in creating an accurate interpolation algorithm.

3.3.1.1 Nearest Neighbour Averaged Linear Interpolation:

This technique uses the routine FINDEL to find the reference number of the element at the current point, and the uses FINDNN to find the reference numbers of the three neighbouring elements. The geometrical centres of each of the neighbouring elements can then be determined, and the value of the potential at the three elements centres calculated by linear interpolation of the values of the potential at the three nodes of each element (using LINPOT). The way in which this is done refers back to the roots of the finite element method (by expressing the geometrical centre of each element in terms of areal coordinates), and is equivalent to fitting a plane to each of the elements nodes (in z,r,V space) and then using this plane to find the value of V at the midpoint. The three sets of (z,r,V) coordinates for the cell centres are then expressed as another plane, and this can then be used to find the electric field. The main problem with this method is that it uses a very simple linear approximation over a relatively wide physical range, and so can miss the finer details of the shape of the potential surface.

3.3.1.2 Linear Interpolation:

This approach is closely related to the nearest neighbour averaging algorithm above, but much simplified. Instead of averaging the potential over the three nearest neighbour elements, it simply takes the linear interpolation of the three nodes of the currently occupied element. This means that the potential being used is exactly that which was calculated by the finite element method, and while it take no account of the general form of the potential in the region surrounding the current element, and so is not a particularly accurate approach, it does have the advantage of simplicity, and of forming a reliable

base on which more complex techniques can be developed.

3.3.1.3 Localised Nth Order Spline:

This technique attempts to better describe the potential surface by fitting a non-linear spline it's form as opposed to using a linear approximation. The technique is related to the Chebyshev algorithm (see Press et al[2]), except that the spline is fitted locally to the point at which we require to know the potential (as opposed to a set of splines over the whole domain), and that it must work in two dimensions. Each spline has the form:

$$y = a_{n+1}x^n + a_n x^{n-1} + a_{n-1}x^{n-2} + \ldots + a_2 x + a_1 \quad (9)$$

And so clearly an nth order spline required n+1 coefficients (a_i) in order for it to be properly defined, and so we need n+1 data points (x_i,y_i) in order to form the n+1 linear simultaneous equations required to find the set of coefficients a_i). This straightforward concept is complicated by the two-dimensional nature of the problem in that in order to fit a nth order spline surface to the potential we need enough sets of n+1 points to create at least two splines. For this problem two sets have been used, one parallel to the z axis, and one parallel to the r axis (using these two perpendicular splines not only improves the accuracy of the interpolated potential surface, but also assists the differentiation of the surface to find the electric field). This concept is illustrated in figure 3 below:

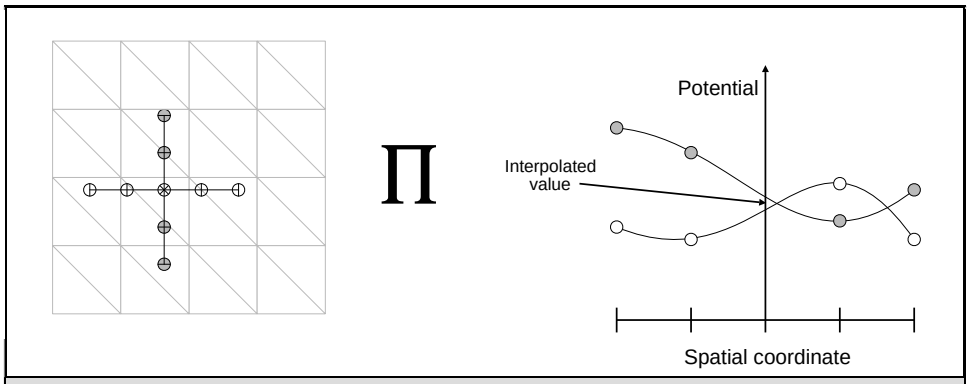

Figure 3: Schematic example of averaging two perpendicular cubic splines for the purpose of two dimensional interpolation.

In order for this approach the work, the code must find the appropriate length scales to fit the splines over, and this must take into account of the variation of element size in the positive and negative z and r directions. This is achieved by finding the relative positions of the centres of the nearest neighbour elements (using FINDNN and FINDELCENT), and then finding the overall z/r span of this set of spatial vectors in both the positive and negative directions. These span lengths are then scaled according to the order of the approximation so that enough different elements will be covered to supply enough different information to make the approximation meaningful, although the best value for the scaling parameter is by no means immediately apparent. The n+1 points are distributed over these z/r ranges, and then LINPOT is then used to find the potential at these 2(n+1) points (note that this means LINPOT now has to call FINDEL as the 2(n+1) point are not necessarily inside the current element).

This method also requires that no spline points lie outside the boundary of the finite element mesh, where the potential is undefined, and so the code must also find the maximum spline in the positive and negative z and r directions and make sure the interpolation splines fit within these maximum values (MinSplin). The only exception to this is the case of interpolation near the rotational axis, where the spline must be allowed to fall outside the mesh (ie into negative r values) by using the fact that V(-r,z) is equal to V(r,z).

The computational solution proceeds as follows. The subroutine MeshSpline uses FINDEL to find the element at the current point, then uses FINDNN to find it's nearest neighbours, and then uses FINDELCENT to calculate the centres of these cells. This data is then used to construct the characteristic cell dimensions in the positive and negative z and r directions (Zmax, Zmin & Rmax, Rmin). These characteristic lengths are then modified such that they cover a sufficiently large spatial range, using the expressions:

Zmin := nthsc × (n+1) × Zmin
Zmax := nthsc × (n+1) × Zmax
Rmin := nthsc × (n+1) × Rmin
Rmax := nthsc × (n+1) × Rmax

Where nthsc is the scaling parameter mentioned above. Once a set of spline lengths have been found, the code calls MinSplin, which returns the largest allowed values of Zmax, Zmin, Rmax and Rmin as defined by the boundary conditions for the current point. These two sets of values are then compared, and the spline lengths are made to fit the boundaries of the device where necessary.

The process continues by breaking the spline lengths down into n+1 evenly spaced point (z_1 - z_{n+1} and r_1 - r_{n+1}), and using LINPOT to find the values of the potential at these points (ie $V_i(z_i,r)$ and $V_i(z,r_i)$). These two sets of data are then passed to the routine nthapprox, which uses a fully-pivoting Gauss-Jordan matrix routine (from Press et al[2]) to solve for the spline coefficients for each set of data. Once the coefficients have been found, nthaeval can be used to evaluate the approximation to the potential for any point z,r in the region of the spline.

3.3.1.4 Localised Nth Order Least-Squares Fit Polynomial:

This fourth technique is closely related to the localised spline calculation, and differs only on the fact that instead of using n+1 points to find an nth order spline to approximate the potential surface, it uses 2n points and fits a least squares polynomial to potential data. As this technique uses more data from the mesh to form the same order of interpolation over the same spatial area, the form of the polynomial should more closely approximate the potential surface. The routine used to perform this approximation is SVDFIT (from Press et al[2]) which replaces my own nthapprox routine, and has the advantage of

using single value decomposition as opposed to Gauss-Jordan elimination (which can cope better with extreme cases such as near singular matricies).

In order to find out which of the above methods of interpolation performs the best, the code can be made to produce a interpolated version of the axial potential, and then this can be compared with the potential on each of the nodes along the axis. If the interpolation is working well, the form of the interpolated potential curve should be smooth, and should agree with the potential from the finite element calculation at each node. In this way it should be possible to determine which approach works the best, and in the case of the latter two to find the optimum values for the order of the interpolation (n) and the scaling parameter (nthsc).

It should be noted here that during the development of the program, some of the most basic routines, which existed before this project began, were found to be working incorrectly. Firstly, the routine to identify the element whose area includes a given point r,z, FINDEL, was searching the element array incorrectly, and somtimes assumed a point to be outside the finite element domain when this was not the case. Also, the routine designed to find the nearest neighbour elements (FINDNN) was failing, and returning invalid element reference numbers. Both these faults have bee corrected. Also, the optimum element numbering for the finite element solution need not correspond the the spatial arrangement of the cells, but FINDEL works more quickly when the reference numbers of the cells are arranged so that they correspond to the spatial order, and so the code runs much faster if the elements have been sorted according to their position. Before the project began, a very basic sorting was being used (taking tens of seconds to sort the mesh data), and during the codes development this routine has been replaced by QuickSort from Press et al[2], with the consequence that the sorting process is now at least one order of magnitude faster than before.

3.3.2 Calculation of the Electric Field:

Given the potential in it's interpolated form, we need to find the derivative of the potential at the current point, in other words the electric field (E_z, E_r), in order to integrate the equations of motion of the electron. Two different schemes were compared for this calculation.

3.3.2.1 Linear Surface Gradient:

This technique was applied to both the linear interpolation technique results, and is the original routine from the code before this project began. When called, the routine GRAD takes the plane formed by the interpolation code and returns the gradient of the plane in the z and r directions as the values for the electric field. The main drawback of this approach is that as the interpolation uses the same single linear approximation for each element, then no matter where the electron lies in that element the

electric field will be found to be the same. In other words, this technique cannot take the general form of the potential into account, and so it can be expected to perform badly when it comes to finding the derivative of the potential.

3.3.2.2 Nth order polynomial derivative:

In the case of the two nth order polynomial interpolation routines, the derivative of the potential at the current point can be calculated by direct differentiation of equation (9) such that:

$$dy/dx = n\,a_{n+1}x^{n-1} + (n-1)\,a_n x^{n-2} + (n-2)\,a_{n-1}x^{n-3} + \ldots + 2a_3x + a_2 \qquad (10)$$

As the interpolation routine takes a broad physical range into account, this approach should form a better approximation to the form of the potential and so form a much more accurate representation of the electric field than the linear gradient approach.

The validity of these routines can be checked in a similar way to the interpolation. Instead of plotting V as a function of z, the two derivatives E_z and E_r can be plotted as a function of z and r respectively for any arbitrary trajectory through the device. Although there is no exact form of the electric field with which we can compare these results, as there was with the interpolation check, we can still use these plots to check that the calculated electric field is varying smoothly and consistently. Also, if the results of the tests of the different interpolation are inconclusive as to which is better, then the more challenging differentiation check can be used.

3.3.3 Integration of the equations of motion

Whilst the details of the different integration routines vary, the general structure of the integration does not. Each is designed to solve a set of N first-order differential equations, and so in order to solve the problem the equations of motion (6) are re-expressed to form a set of four coupled first order differential equations:

$$\frac{dv_z}{dt} = \frac{e\,E_z}{m} \qquad \frac{dz}{dt} = v_z \qquad (11)$$

and

$$\frac{dv_r}{dt} = \frac{e\,E_r}{m} \qquad \frac{dr}{dt} = v_r \qquad (12)$$

The integration routine, when called from PATHS, calculates the change in velocity and position for one time step, h, and the DERIVS routine supplies the values of the electric field and velocity at the current position (using one of the sets of interpolation and differentiation routines outlined previously). The new point of the trajectory is then stored and plotted on the screen.

For this process to work at all, it is necessary to know what value of h would be reasonable for the integration, and this cannot be set arbitrarily because the potential of the system (and so the velocities involved) can differ greatly between devices. To get around this problem, PATHS looks at the potentials on the plates and uses this to estimate the speed of an electron moving through the device along the axis. This is then used as the basis for the estimation of the required timestep. The different integration algorithms that have been implemented are as follows:

3.3.3.1 Adaptive Runge-Kutta

This routine, taken from Press et al[2], takes the fourth order Runge-Kutta routine (see 3.3.3.3 below), and modifies it's implementation such that the time step, h, can be altered during the course of the integration. In this way many small steps can be used to tiptoe through the quickly varying parts of the potential, and a few much larger steps employed to cover the smoother areas, thus making the process more time-efficient. However, this routine was suspected not to be working correctly, and so was replaced with a somewhat simpler method.

3.3.3.2 Simple Euler Integration

This routine is extreme easy to implement, and represents the simplest possible method of integration, where equations of the form:

$$dx/dt = f(x)$$

are solved using

$$\Delta x = \Delta t\, f(x)$$

expressed in the program as:

$$x_{n+1} = x_n + h\, f(x)$$

While this routine is not particularly accurate (error ~ h^2), and occasionally unstable, it has the advantage of being almost impossible to implement wrongly. In this way, the possible failure of the adaptive Runge-Kutta algorithm can be checked.

3.3.3.3 Runge-Kutta

The Runge-Kutta algorithm is based of the principle of taking set of Euler type steps, and then using the information obtained to fit a Taylor expansion up to some higher order. The routine RK4 (from Press et al[2] used here represents the form of this technique when four Euler steps are taken, leading to a Taylor approximation with an error of the order of h^5. While this technique represents a great increase in accuracy over the Euler approach, a simple way to improve it is by using extrapolation.

3.3.3.4 Extrapolative Runge-Kutta

This technique, taken from Hawkes & Kasper[1], increases the accuracy of the Runge-Kutta routine by means of an extrapolation of the form:

$$x_{n+1} = x^{(2)}_{n+1} + {}^{1}/_{15}\,(x^{(2)}_{n+1} - x^{(1)}_{n+1})$$

where $x^{(1)}_{n+1}$ represents the results of a single Runge-Kutta step over an interval h, and where $x^{(2)}_{n+1}$ represents the results obtained by using two steps of interval ${}^{h}/_{2}$. The main draw back is that the Runge-Kutta routine needs to evaluate the electric field four times, and so the extrapolating algorithm requires 16 evaluations per time step. This can lead to a rather slow integration process when, as in this case, the value of the electric field requires quite a degree of computation. Because of this the structure of the code was changed so that instead of constructing a new interpolation for the mesh every time the DERVIS routine is called, the interpolation is only constructed for every time step. This makes the assumption that the form of the potential does not change significantly over the distance covered in the time interval h, and as this is required to be true in order for the integration to work accurately anyway, this assumption perfectly reasonable.

3.3.3.5 Hamming Predictor-Corrector

This multistep algorithm (also from Hawkes & Kasper[1], and similar to that used by Natali et al[3]) differs from the previous routines in that instead of using only the current point to predict the next point via Euler type steps, the new point is calculated by forming an appropriate linear combination of the preceeding ones. Since this can be done in different ways, it automatically provides an accuracy control, Whereas all of the other routines tend to propagate and accumilate errors as the integration proceeds. The Hamming algorithm breaks down into two parts, the predictor (which uses the previous four positions to predict the next one), and the corredtor (which corrects the predicted position by using information supplied by the DERIVS routine). The accuracy check comes from comparing the predicted and corrected results, such that the time step is halved if the accuracy falls below some user defined level (errs), and doubled if the accuracy is significantly better than errs (two hundred times better in this particular case). While this changing time step should mean that the code works efficiently and accuratly, the code does have the drawback that it is not self-starting, requiring four Runge-Kutta steps to be carried out before it can be used, and also for restarting the algorithm when the time step is altered (as the stored points correspond to the old time step). From Natali et al[3], this drawback should lead to an error of only 0.01%.

Clearly, the extrapolative Runge-Kutta will perform better than the Euler and basic Runge-Kutta routines, and so the question that remains is whether the Adaptive Runge-Kutta, the Extrapolative Runge-Kutta or the Hamming Predictor corrector is the most accurate. This can be ascertained firstly be simply checking that the electron trajectories have a single, well defined focal point for rays near the

axis, and also by checking that at higher angles the focal point follows the expected form due to spherical aberration. As well as this, we know that the potential and particle form a energetically closed system, and so if we add together the potential energy for the electron at it's current position and it's kinetic energy, then we can check whether this figure is constant, as it should be. This technique can also be used to investigate the quality of the interpolation and differentation algorithms.

3.4 Calculation of Lens Properties:

As the electron trajectories are calculated, part of the PATHS routine checks whether the last step has caused the electron to cross the optical axis, in other words to see if the focal point has been calculated. Using this condition the routine then stores the focal point (z_f) and the slope of the ray ($\tan(\alpha)$) in an array, which can then be used to calculate the paraxial focus (z_{fp}) and the third-order spherical aberration coefficient (C_s) using equation 8. This is achieved by the FINELABBERS routine by using the SVDFIT least-square algorithm to finding the coefficients of a polynomial of the form:

$$y = a_2x^2 + a_1$$

Where x corresponds to the slope $\tan(\alpha)$, y corresponds to the direct ray focal point z_f, and a_1 and a_2 correspond to the paraxial focus and the aberration coefficient respectively. When the coefficients have been found they are printed to the screen, along with an estimate of the error in each figure (calculated by SVDVAR, another routine from Press et al[2]). Unlike the paraxial routine, the direct ray calculation only returns the value of the spherical aberration when it is actually affecting the focal length, and the quality of the fit depends on the number of trajectories that have been calculated.

4. Results & Discussion

4.1 Validation Of The Finite Element Potential

The axial potential of a standardised two-tube electrostatic electron lens was compared with the published results of Natali et al[3]. As the published data consists of tabulated potential values (for one half of the electron lens), the finite element mesh was defined such that the points on the axis for the mesh match up with the tabulated values (Appendix A.1 shows the finite element mesh used). The tabulated values are presented below, along with a graph of the axial potential from both tables, with the present result plotted as points and the previous paper's potential plotted as a line (figure 4). Just by visual comparison the two results clearly agree, and analysis of the tabulated data has shown that the difference between the two calculations was only 0.02%. On the basis of this evidence, the finite element potential calculation is seen to be working well.

z/D	Electrostatic Potential	
	Present results	Natali et al
0.00000000	0.50000000	0.50000000
0.02500000	0.46714915	0.46711500
0.05000000	0.43459568	0.43453200
0.07500000	0.40262572	0.40253800
0.10000000	0.37150388	0.37139900
0.12500000	0.34146465	0.34135100
0.15000000	0.31270599	0.31259200
0.17500000	0.28538560	0.28527900
0.20000000	0.25961956	0.25952800
0.22500000	0.23548332	0.23541300
0.25000000	0.21301910	0.21296900
0.30000000	0.17307084	0.17307900
0.35000000	0.13948118	0.13956600
0.40000000	0.11168299	0.11185100
0.45000000	0.08895959	0.08921200
0.50000000	0.07055776	0.07089700
0.60000000	0.04412242	0.04443600
0.70000000	0.02741136	0.02767600
0.80000000	0.01697056	0.01717800
0.90000000	0.01048736	0.01064200
1.00000000	0.00647471	0.00658600
1.10000000	0.00399538	0.00407400
1.20000000	0.00246481	0.00251600
1.30000000	0.00152037	0.00155700
1.40000000	0.00093774	0.00096200
1.50000000	0.00057837	0.00059500
1.60000000	0.00035671	0.00036800
1.70000000	0.00022000	0.00022700
1.80000000	0.00013568	0.00014100
1.90000000	0.00008368	0.00008700
2.00000000	0.00005161	0.00005400
2.10000000	0.00003183	0.00003300
2.20000000	0.00001963	0.00002000
2.30000000	0.00001211	0.00001300
2.40000000	0.00000747	0.00000800
2.50000000	0.00000461	0.00000500

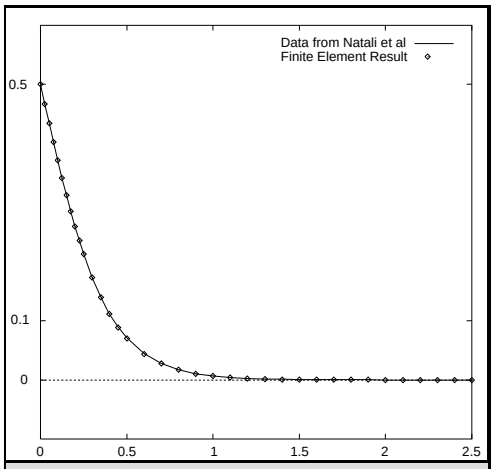

Figure 4: Comparison of the present results with those of Natali et al[3].

4.2 Analyis And Development Of The Direct Ray Tracing Approach

All the following results were taken for the same thin lens as that used by Liu & Ximen[5] (the form of the finite mesh used to describe it is supplied in appendix A.2).

4.2.1 Interpolation Of The Axial Potential From The Finite Element Mesh

The first test carried out on the interpolation was to check that the interpolated axial potential agreed with the values of the potential on the finite element nodes along the axis. Figure 5 compares the four interpolation techniques against the node potential.

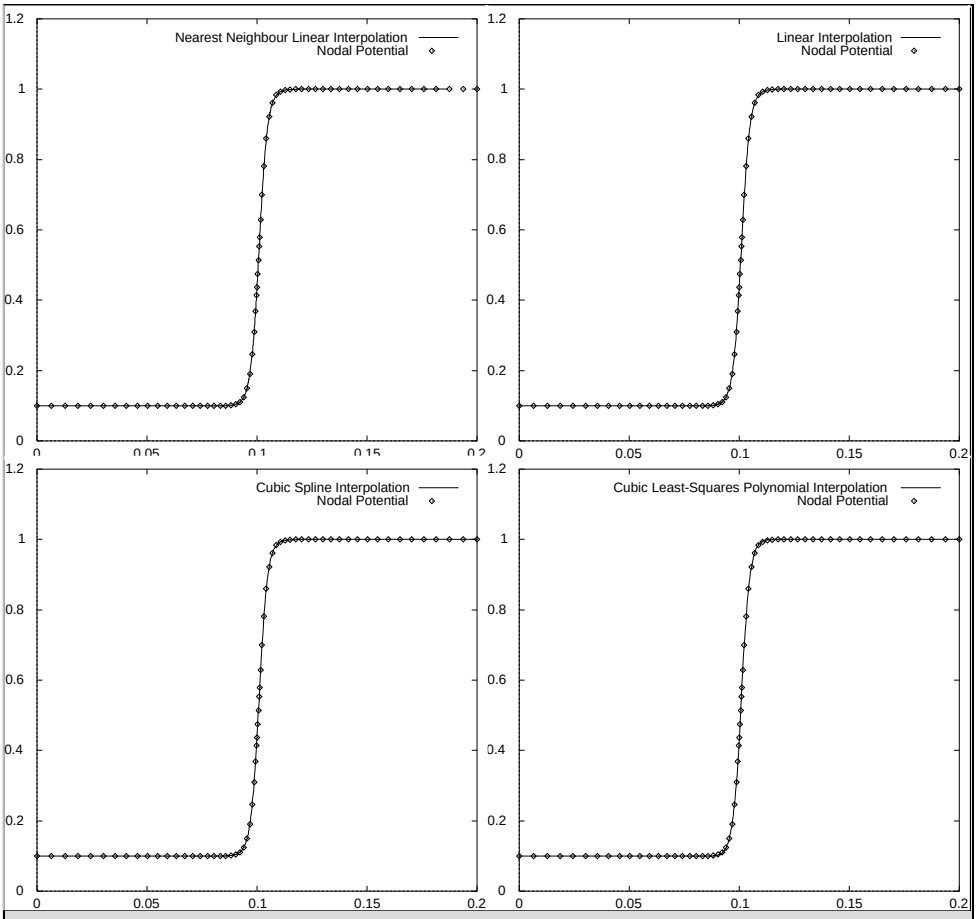

Figure 5: Comparison of the axial potential for the four methods of interpolation with the nodal potential from the finite element calculation.

Unfortunately, these results give no clear evidence of any one routine performing any better than any of the others, all of them agreeing reasonably with the finite element node potentials. This being the case, the differential of the potential must be examined in order to determine which between of the techniques is the best.

4.2.2 Differentiation Of The Electrostatic Potential

Figure 6 presents the z component of the electric field for the two linear interpolation techniques, and figures 7 and 8 compare the same quantity calculated by the spline and least-squares approaches respectively. For these latter two, the left-hand plot corresponds to linear interpolation, the middle plot to quadratic and the right-hand plot to cubic interpolation. For each of these a range of scaling factors are compared (nthsc = 1, 2, 3).

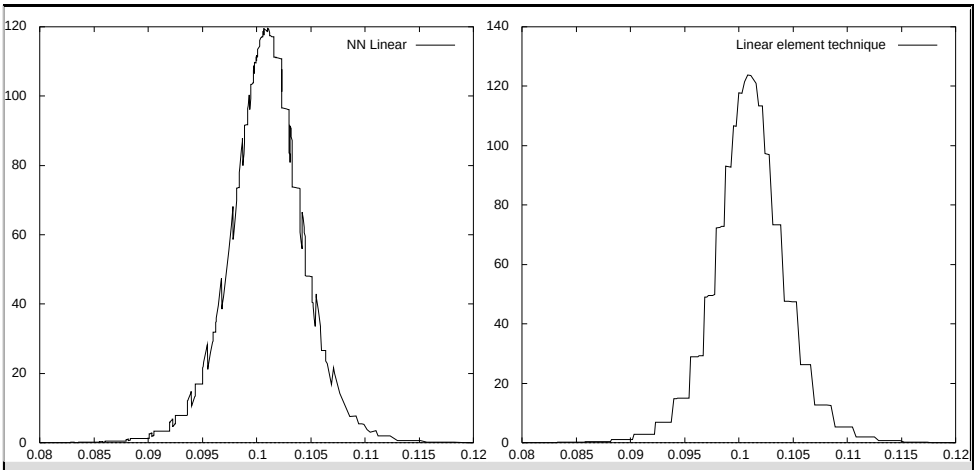

Figure 6: Plots of E_z as calculated by the nearest neighbour and single element linear interpolation.

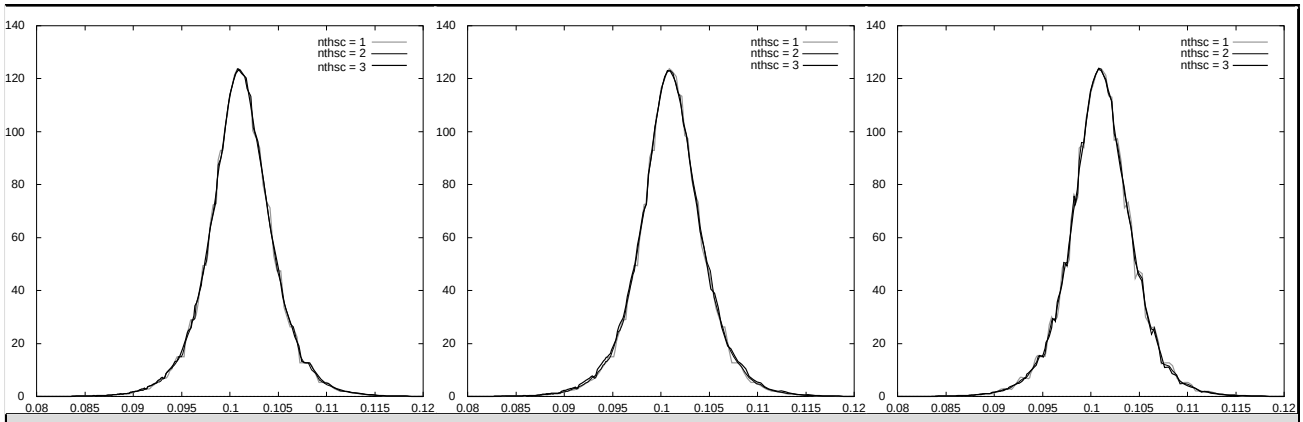

Figure 7: Comparison of the spline interpolation routines for linear, quadratic and cubic splines left to right. Each diagram compares the results from using a scaling factor of 1, 2 and 3.

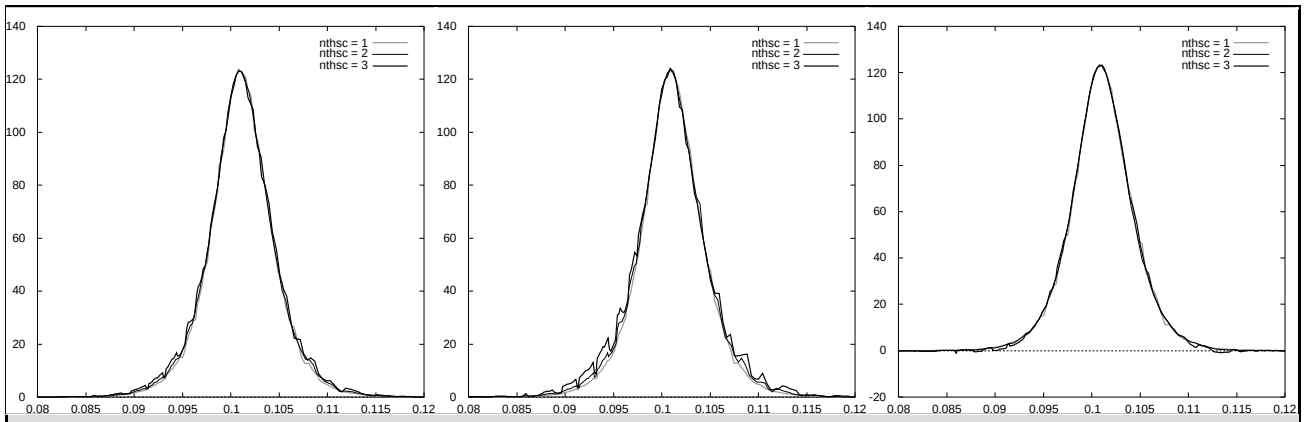

Figure 8: Comparison of the least-squares polynomial interpolation routines for linear, quadratic and cubic splines left to right. Each diagram compares the results from using a scaling factor of 1, 2 and 3.

Both of the linear interpolation techniques can clearly be seen to form a poor approximation to the electric field, and closer inspection of the results in figures 7 and 8 (by checking which best conserves the total energy of the potential and the electron) has shown that the least-squares cubic approximation forms the smoothst interpolation, whilst also being least dependant on the scaling factor. This means that the technique should be able to cope better with the different variations in any finite element mesh it is applied to. The routine was refined by searching for the value for the scaling factor which minimised the error in the total energy for a ray passing through the lens, and this investigation found that the best value of the scaling factor for the cubic interpolation is nthsc = 1.75 (which correspond energy being conserved with 0.01% accuracy.

4.2.3 Integration Of The Equations Of Motion

When the Adaptive Runge-Kutta routine was removed, and then replaced with the theoretically less accurate Euler routine, there was an immediate improvement in the quality of the results. This apparent paradox occurred because while the Adaptive Runge-Kutta works well with most smoothly varying data, it appeared to be unable to cope with data that has a very small degree of variation. After discovering this, the Euler alrogithm was replace with the fourth order Runge-Kutta routine, and then with the extrapolative Runge-Kutta, with the accuracy of the results improving along the way. However, it was not immediately apparent whether the Hamming Predictor-Corrector or the extrapolative Runge-Kutta was giving the best results, and this section deals with the differentiation between the quality of these two routines.

The extrapolative Runge-Kutta routine was tested by calculating the error in the total energy for a single ray passing through the lens at a fixed angle (1° to the optical axis, starting on the axis at the left-hand side of the device) for a range of values for the time step. The time step was altered by taking the estimated time step value and multiplying it by an constant less than or equal to 1.0. In the case of the Hamming predictor corrector routine, the same calculation was carried out using the best time step from the Runge-Kutta results as the initial time step and then setting the error parameter (errs) to a range of values to see how it affected the results. The data from these tests is presented in the table overleaf.

Runge-Kutta			Hamming Precitor-Corrector	
Time Step ×	Energy Error		Accuracy Factor	Energy Error
1.000	1.166e-02		1.0e-02	7.481e-05
0.550	4.735e-04		1.0e-03	7.460e-05
0.525	1.974e-04		1.0e-04	7.434e-05
0.500	7.439e-05		1.0e-05	7.439e-05
0.475	1.942e-04		1.0e-06	1.661e-03
0.450	1.339e-04		1.0e-07	1.401e-03
0.400	2.735e-04		1.0e-08	1.212e-03
0.300	2.570e-03		1.0e-09	8.180e-04
0.200	1.206e-03		1.0e-10	8.219e-04
0.100	9.000e-03		1.0e-11	8.138e-04
0.050	1.038e-03		1.0e-12	8.111e-04

The result for the extrapolative Runge-Kutta routine is as would be expected for any single step algorithm. While decreasing the time step initially decreases the error in the results, for very small time steps the accumulated errors begin to become significant, and so the accuracy of the integration decreases. The balance between these two types of behaviour is therefore found to occur at a time step factor value of 0.5, and so this value gives the best results for the Runge-Kutta integration. This figure was then used to find the initial time-step for the Hamming routine.

The behaviour of the Hamming routine can be explained in terms of the accuracy factor (errs) as follows. For the larger values of the accuracy factor (errs = 1.0e-2 - 1.0e-5), the initial time step is sufficient to satisfy the error criterion, but as it is decreased furthur the routine has to change the time-step in order to satisfy the same accuracy condition. Unfortunately, when this happens the Hamming code has to be restarted with the extrapolative Runge-Kutta, and this causes errors to form in the results (especially if the time step is changing rapidly, in which case the Runge-Kutta accumulates errors as before). However, in this region, the calculated error is approximately constant for the different values of errs, which implies that while the Hamming code is less accurate than an extrapolative Runge-Kutta routine with an optimised time step, it is more consistently accurate than the Runge-Kutta for a range of accuracy conditions. This means that the Hamming code is more flexible and so more likely to work well for different problems, where the optimum time step may differ from the value found here.

The only drawback is that while the Runge-Kutta is always reasonable quick (~ 20-30s per ray), the Hamming code slows down as the accuracy criterion starts to work, and so when a device is being simulated it is best to start with errs ~ 1.0e-4 (to get a rough estimate of lens behaviour) and then change this to errs ~ 1.0e-9 when more accuracy is required. Although the accuracy factor can be set higher than this, the code starts to run very slow indeed (~ 2-3 minutes per ray) and there is very little increase in accuracy (due to the Runge-Kutta/Hamming interface).

4.3 Example Ray Diagrams For The Different Techniques

The overall ray diagram results from various techniques are presented here for direct visual comparison. Figure 9 shows the ray diagram for a simple two-tube electron lens as calculated by the paraxial approximation, and figures 10 and 11 compare the results for the same lens as calculated by the direct ray tracing code before this project began with the results the code now produces.

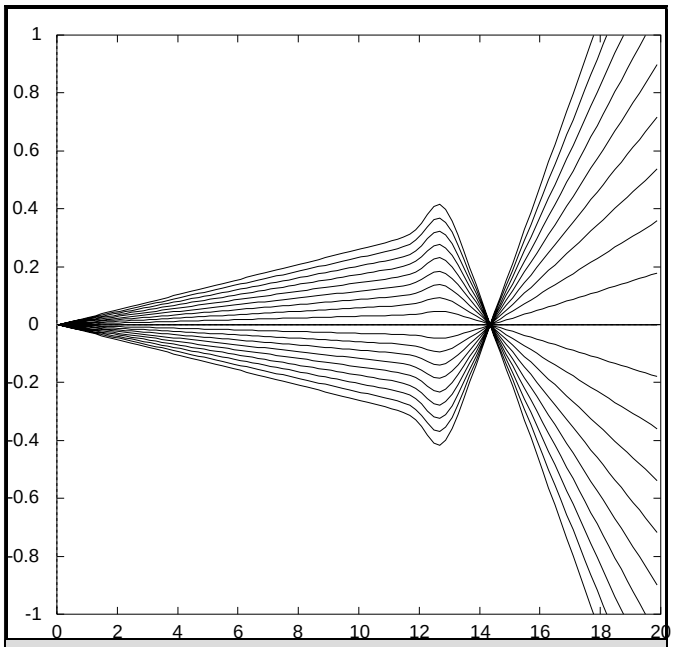

Figure 9: Ray diagram for a simple two-tube electron lens as calculated by the paraxial approximation.

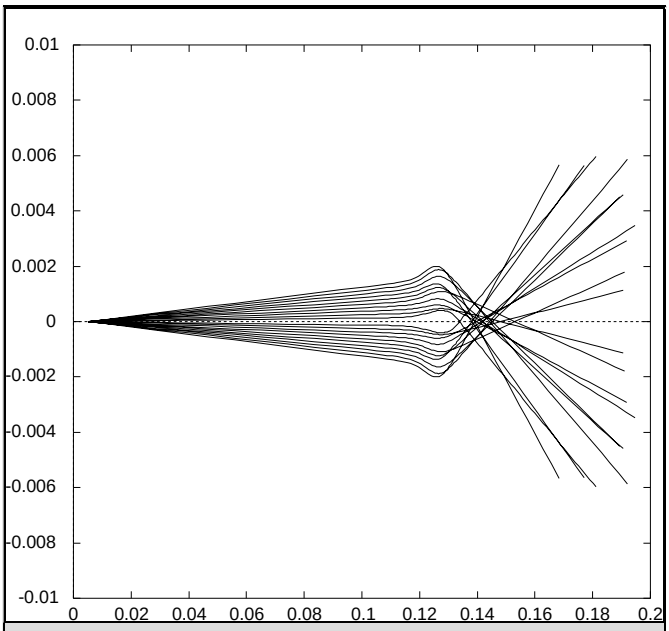

Figure 10: Ray diagram for a simple two-tube electron lens as calculated by the direct ray tracing code before this project began.

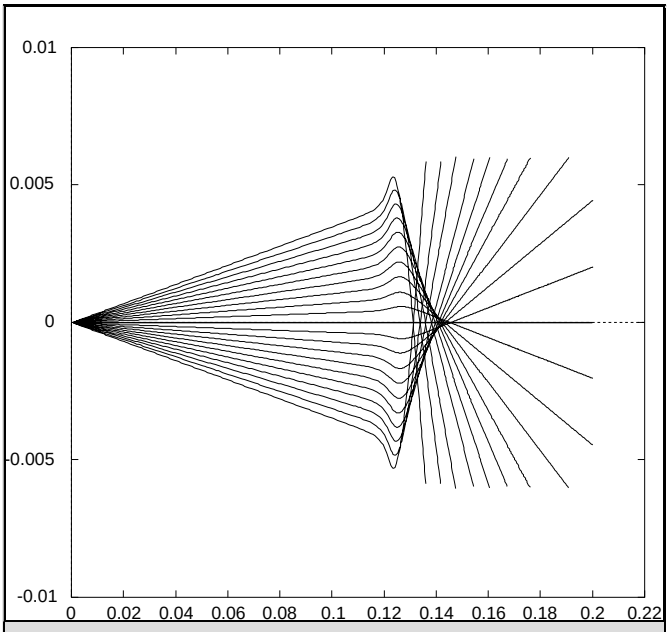

Figure 11: Ray diagram for a simple two-tube electron lens as calculated by the new direct ray trading algorithm.

The non-physical behaviour of the old code has been eliminated from the direct ray tracing code to the point that the ray diagrams now agree with the paraxial ray diagrams for low angle rays. Moreover, the effect of spherical aberration is now clear, with the direct ray focus moving inward from the paraxial focal point as the angle increases. The degree of agreement between the paraxial and direct ray tracing will be examined further in the next section.

4.4 Comparison With Published Data For The Two-Tube Electron Lens

The table below gives the results from a series of simulations of the standardised two-tube electron lens (with g/D = 0.1) for a range of voltage ratios. The paraxial focus and third order spherical aberration coefficients from the direct ray calculation are compared with those from the paraxial approximation code and the published results of Liu & Ximen[5]. The direct ray calculations were carried out using the Hamming PC code (errs = 0.1e-6) and the mesh interpolation performed by the cubic least-squares approximation (nthsc = 1.75).

	f			Cs		
V1/V2	Direct	Paraxial	Published	Direct	Paraxial	Published
5	2.4561	2.4414	2.465	1308.03	1194.20	57.83
10	1.1770	1.1661	1.179	11.7703	15.11	10.10
20	0.6426	0.6314	0.630	8.3571	4.88	3.511
40	0.3332	0.3218	0.319	4.7209	1.96	1.922

Examining the progression in the focal point with increasing voltage ratio, it appears that while the direct raytracing code works well at lower voltage ratios, as that ratio increases the accuracy of the code decreases. This is because the more rapidly changing electric field is more difficult to describe accurately, and so greater errors are made by the approximation which are then carried through into the integration. The reason for the discrepancy between the paraxial approximation and the published results is unknown, as the details of the paraxial algorithm have not been studied here. However, the paraxial approximation consistently gives a reasonable approximation to the published results.

The direct calculation of the third-order aberration coefficient is somewhat more troublesome. This is mainly because the direct technique requires a number of rays to be simulated where the effect of the aberration is fairly pronounced (due to the errors in the ray tracing process), and so when the aberration is small, it becomes more difficult to get the simulation to actually display spherical aberration behaviour so that it can be calculated. The discrepancy between the published results and the results of both the paraxial and direct ray tracing code is of unknown origin, but the fact that the paraxial code and direct code approximately agree (coupled with the consistently reasonable behaviour of these methods for all the other results) implies that the published results may be incorrect for this particular lens definition.

5. Conclusions

The primary aim of this project was to improved the direct ray tracing code to the point that it's predictions are useful, and this has been achieved. While the paraxial approximation generally tends to gives more accurate results in any given situation, the ray diagrams produced do not show exactly what is happening whereas the direct ray tracing approach vividly illustrates the effect of spherical aberration on the focal properties of a device. For this reason I believe the approach is of great use in terms of device design by allowing the user to see roughly how significant the spherical aberration is and to get an idea of the general behaviour of the lens. Also, the finite element solution for the potential has been show to be good (although recent work suggests there may be advantages in using the more complex second order finite element method[6]), and the paraxial approximation code shown to work to consistently reasonable accuracy.

The most important development in this project has been the successful interpolation of a two-dimensional finite element mesh, which to my knowledge has not been achieved before (at least in the field of electro-optics). Previous direct ray tracing software has used a set of cartesian mesh points for which high accuracy interpolation is elementary, at the cost of restricting the types of device geometry that can be described. The new code uses the more flexible finite element approach, and the interpolation is flexible enough to cope with awkward arrangements of elements, for example when moving into the gap of a two-tube lens, where the size of the elements changes rapidly.

In my opinion, the integration code is the main cause of the remaining errors in the direct ray simulation, and can be improved in a number of ways. The inaccuracies in the Hamming predictor-corrector routine are mainly a consequence of the need for the code to be restarted with the Runge-Kutta routine every time the time step is halved or doubled, and this could possibly be rectified by using the more complex HPCD algorithm[1]. This version of the Hamming code gets around the problem of restarting the predictor-corrector by storing a greater number of the previous points, and then using this data to find the appropriate set of previous points for the halved/doubled time step by rearrangement and interpolation of the stored points.

For more accurate results still, there are a number of ways in which the interpolation might be improved. Firstly, only polynomials up to third order have been investigated, and quartic or quintic interpolation may well produce even better results (at the cost of slowing down the integration process, as a matrix of $2n \times n$ elements must be solved). Secondly, the accuracy of the current interpolation was improved by finding the scaling factor which most successfully conserved total energy, but the best scaling factor will change a little for different mesh geometries, and so higher accuracy might be obtained by making the code automatically search for the best scaling factor via the total energy condition (although this will slow the integration down).

This automated total energy check leads naturally to a new approach to the ray tracing problem. Instead of integrating the equations of motion of an electron whilst monitoring it's progress via a total energy check, the code could use the conservation of energy principle more directly via the principle of least action. The integration would be broken up into a set of small time steps (as before) and then the movement of the electron from one point to the next would be achieved by finding the speed and angle (and thus the kinetic and potential energy at the end of the interval) which gave the minimum action for that step, perhaps by Monte Carlo methods. This would naturally lead to the path of least action through the lens for an electron with a particular set of initial conditions, and has the advantage that the derivative of the potential is not required. This means that the accuracy of the simulation would be more directly defined by the quality of the finite element mesh. As this represents a significant departure from the technique used in the current program, there was not enough time to investigate this possibility. However, in my opinion this could form an interesting source of future research.

6. References

[1] Hawkes P.W. & Kasper E. (1989) Principles Of Electron Optics, Volume 1 (Academic Press).

[2] Press W.H. et al (1992) Numerical Recipes in FORTRAN, 2nd Ed (Cambridge).

[3] Natali S. Chio D.Di Kuyatt C.E. (1971) Accurate Calculations of Properties of the Two-Tube Electrostatic Lens. I. Improved Digital Methods for the Precise Calculation of Electric Field and Trajectories (Journal of research of the National Bureau of Standards, **76A**, no. 1, pp 27-35).

[4] Munro E. (1990) Numerical modelling of electron and ion optics on personal computers (Journal of Vacuum Science Technology, **8B,** no.6, pp 1657-1665).

[5] Liu Z. & Ximen J. (1993) Numerical analysis of higher-order geometrical aberrations for a two-tube electrostatic lens (Journal of Applied Physics, **74**, no. 10, pp 5946-5950).

[6] Zhu X. & Munro E. (1995) Second-order finite element method and its practical application in charged particle optics (Journal of Microscopy, **179**, pt. 2, August, pp 170-180).

[7] Hecht E. (1987) Optics, 2nd edition (Addison-Wesley).

[8] Sturrock P. A. (1955) Static & Dynamic Electron Optics (Cambridge).

[9] Munro E. (1975) A Set Of Computer Programs For Calculating The Properties Of Electron Lenses (Engineering Department, University of Cambridge ref: CUED/B- Elect TR45).

[10] Kikuchi M. (1986) Finite Element Methods In Mechanics (Cambridge).

[11] Liu Z. & Ximen J. (1992) - (Journal of Applied Physics E, **72**, pp 28).

Appendix A: The Finite Element Meshes

A.1 Finite Element Mesh Used For Comparison With Natali Et Al[3]:

A.2 Finite Element Mesh Used For Comparison With Liu & Ximen:

Appendix B: The Ray Tracing Code: TRAJEC.FOR:

```
$notstrict
$notruncate

        SUBROUTINE TRAJEC
C  Programme to read in data created by BUILD and calculate the paths,
C  Version 6.1  MP  24.10.96
C
C  The file courb.fon must be present in the environment set by DOS

$include: 'lens.def'
$include: 'gracol.def'
$include: 'trajec.def'

        INTEGER X,Y,STAT,TRAJCODE
        REAL EE,L,VP
        LOGICAL OVLY,PLOT
        REAL*8 SLOPE

        X = 1
        Y = 3
        STAT = 0
        DATA X00,Y00,NRAYS,THETA1,THETA2/0.05,0.0,5,0.2,1.0/
        TRAJED = .FALSE.
        TRAJCODE = 0

        EE = 0.0
        IF (EE.GT.0.0) THEN
          VP = SQRT(2*EE*EMR)
        ELSE
          VP = 0.0e0
        ENDIF
        EPS = 1.E-5

 5      CALL MNEW('TRAJECTORY','TRAJEC.HLP')
        CALL MROW('File: %8!15Start KE:%7 eV!35Start at Z:%7 cm'//
     +           '!58R:%7 cm !71Paths: %2')
        CALL MROW('Accuracy: %9 !44Angles: Start %5 !55Stop %5'//
     +            ' degs')
        CALL MROW('Calculate: Paths $Finite El.or!31$Paraxial' //
     +            ' !44$Aberrations !61$Magnification')
        CALL MROW('$Exit to main menu')

        CALL MENTS(1,1,NAME)
        CALL MREAL(1,2,'F%7.1',EE)
        CALL MREAL(1,3,'F%7.4',X00)
        CALL MREAL(1,4,'F%7.4',Y00)
        CALL MINT(1,5,'%2',NRAYS)
        CALL MREAL(2,1,'E%7.1',EPS)
        CALL MREAL(2,2,'F%5.2',THETA1)
        CALL MREAL(2,3,'F%5.2',THETA2)

 10     CALL GCMD('Your choice?',Y,X,STAT,*900)
        GOTO(100,200,300,400) Y

 100    GOTO(110,120,130,140,150) X

 110    Y = 2
        X = 1
        GOTO 5

C Enter the start kinetic energy

 120    CALL GREAL('Enter energy (eV) at start of paths',
     +             Y,X,EE,EE,0.,20000.,STAT,*900)
C Initial velocity of electrons VP
        VP = SQRT(2*EE*EMR)
        VAXED = .FALSE.
        GOTO 5

C Enter the x,y coordinates for the start of the paths

 130    CALL GREAL('Enter Z coord.for start of paths (cm)',
```

```
     +              Y,X,X00,X00,-100.,ETDE2X(TEDT),STAT,*900)
        GOTO 5

 140    CALL GREAL('Enter R coord.for start of paths (cm)',
     +              Y,X,Y00,Y00,-10.,10.,STAT,*900)
        GOTO 5

C Enter the number of paths to be traced

 150    CALL GINT('Enter no.of rays to be traced',
     +              Y,X,NRAYS,NRAYS,1,LIMNOR,STAT,*900)
        GOTO 5

 200    GOTO(210,220,230) X

 210    CALL GREAL('Enter accuracy required',
     +              Y,X,EPS,EPS,1.e-12,1.e-2,STAT,*900)
        GOTO 5

 220    CALL GREAL('Enter lowest angle for a ray',
     +              Y,X,THETA1,THETA1,-90.,90.,STAT,*900)
        GOTO 5

 230    CALL GREAL('Enter highest angle for a ray',
     +              Y,X,THETA2,THETA2,THETA1,90.,STAT,*900)
        GOTO 5

 300    GOTO(310,320,330,340) X

C Main calculation of paths using finite element modelling

 310    CALL MPUTS('$FINITE ELEMENT TRAJECTORIES:$')
        CALL PATHS(VP,EPS,OVLY)
        TRAJED = .TRUE.
        TRAJCODE = 1
        CALL GRAISE(3)
        CALL GRAISE(2)
        OVLY = .FALSE.
        Y = 3
        X = 3
        GOTO 5

C Calculation of paths using the paraxial ray equation and the axial
C potential computed from the finite element model.

 320    CALL FINDAXPOT(EE,OVLY)

C Now use the interpolated axial potential stored in the common array
C VZINT to find the paraxial paths using Picht's algorithm.

        PLOT = .TRUE.
        TRAJED = .TRUE.
        TRAJCODE = 2
        CALL MPUTS('$PARAXIAL TRAJECTORIES:$')
        CALL AXPATHS(EE,PLOT,OVLY,SLOPE)
        CALL GRAISE(3)
        CALL GRAISE(2)
        CALL PREAL('Source at F%5.2 cm.',X00)
        CALL PREAL(' with energy % eV',EE)
        CALL SPOTAT(L)
        CALL PREAL('!40Spot at F%6.2 cm$',L)
        IF (ETDPOT(12).NE.0.0) THEN
          CALL PREAL('Gun Lens % eV',ETDPOT(12))
        ENDIF
        IF (ETDPOT(23).NE.0.0) THEN
          CALL PREAL('!40Objective Lens % eV$',ETDPOT(23))
        ENDIF
        Y = 3
        X = 3
        GOTO 5

 330    IF (.NOT.TRAJED) THEN
          CALL WARN('Calculate trajectories first')
          Y = 3
```

```
          X = 2
        ELSE
         CALL ABBERS(TRAJCODE)
         Y = 3
         X = 3
        ENDIF
        GOTO 5

 340    CALL MAGNIF
        Y = 3
        X = 4
        GOTO 5

 400    RETURN

 900    STAT =  MAX(STAT,0)
        GOTO 5

        END

C ---------------------------------------------------------------------
        SUBROUTINE PARAXABBERS
$include: 'lens.def'
$include: 'trajec.def'
        LOGICAL PLOT,OVLY
        INTEGER I,J,K,NL,NSTART,NRAYSOLD
        REAL L,THETA1OLD,X00OLD
        REAL*8 ASUM,SLOPE

C Initialize all the variables used for calculation of abberrations

        PLOT = .FALSE.
        OVLY = .FALSE.
        NRAYSOLD  = NRAYS
        THETA1OLD = THETA1
        X00OLD    = X00
        NRAYS     = 1
        THETA1    = 5.0
        CS        = 0.0D0
        CC        = 0.0D0

C Find the trajectory RAD for the special ray arriving at the image at 45
C degrees to the axis.

        CALL AXPATHS(EE,PLOT,OVLY,SLOPE)
        THETA1 = THETA1/SLOPE
        CALL AXPATHS(EE,PLOT,OVLY,SLOPE)

C Calculate the differential RPRIME of RAD wrt Z

        NSTART = 1+INT(X00/DELZ)
        DO 10 I=NSTART+1,NAXPTS-1
          J = I-1
          K = I+1
          RPRIME(I) = (RAD(K)-RAD(J))/(2.0D0*DELZ)
 10     CONTINUE
        RPRIME(NSTART) = RPRIME(NSTART+1)
        RPRIME(NAXPTS) = RPRIME(NAXPTS-1)

C Form the integrand for the spherical abberration coeff.
C Uses Munro, p131. T and TPRIME were calculated in the
C subroutine AXPOT below.

        CALL SPOTAT(L)
        NL =1+INT(L/DELZ)

        DO 20 I = NSTART+1,NL
          IF (RAD(I).EQ.0.0) RAD(I) = 1.0D-6
          AINT(I) = 5.833333333D0*T(I)**4 + 1.0D1*T(I)*T(I)*TPRIME(I)
          AINT(I) = AINT(I) + 5.0D0*TPRIME(I)*TPRIME(I)
          AINT(I) = AINT(I)+1.866666667D1*(T(I)**3)*RPRIME(I)/RAD(I)
          AINT(I) = AINT(I)-6.0D0*(T(I)**2)*((RPRIME(I)/RAD(I))**2)
          AINT(I) = AINT(I)*SQRT(VZINT(I))*RAD(I)**4
 20     CONTINUE
```

```
        ASUM = 0.0D0
        DO 30, I=NSTART+2,NL
          ASUM = ASUM + (AINT(I-1)+AINT(I))*DELZ/2.0D0
 30     CONTINUE

C Now get Cs in mm

        CS = ASUM/(6.4D0*SQRT(ABS(ETDPOT(TEDT))))


C Now calculate the chromatic abberration coefficient

        DO 40 I=NSTART+1,NL
          IF (RAD(I).EQ.0.0) RAD(I) = 1.0D-6
          AINT(I) = 0.5D0*T(I)*RPRIME(I)
          AINT(I) = AINT(I) + 0.25D0*RAD(I)*TPRIME(I)
          AINT(I) = AINT(I) + 0.25D0*RAD(I)*T(I)**2
          AINT(I) = AINT(I)*RAD(I)/SQRT(VZINT(I))
 40     CONTINUE
        ASUM = 0.0D0
        DO 50 I=NSTART+2,NL
          ASUM = ASUM + (AINT(I-1)+AINT(I))*DELZ/2.0D0
 50     CONTINUE
        CC = 1.0D1*SQRT(ETDPOT(TEDT))*ASUM

C Reset the values of some of the variables

        THETA1 = THETA1OLD
        NRAYS  = NRAYSOLD

        RETURN
        END
C ---------------------------------------------------------------------

        SUBROUTINE BOUNDELS

$include: 'lens.def'
$include: 'trajec.def'

        XMAX = -1.E8
        YMAX = -1.E8
        XMIN = 1.E8
        YMIN = 1.E8
        DO 5 I = 1,TEDT
          IF (ETDE1X(I).LT.XMIN) XMIN = ETDE1X(I)
          IF (ETDE1Y(I).LT.YMIN) YMIN = ETDE1Y(I)
          IF (ETDE2X(I).GT.XMAX) XMAX = ETDE2X(I)
          IF (ETDE2Y(I).GT.YMAX) YMAX = ETDE2Y(I)
 5      CONTINUE

        RETURN
        END


C ---------------------------------------------------------------------

        SUBROUTINE FINDEL(ZP,RP,NEL,OUT)

C Finds the number NEL of the element containing point Zp,Rp. If it
C fails then it returns FALSE in OUT.

$include: 'lens.def'
$include: 'trajec.def'

        REAL*8 AREA,DET,ZP,RP
        REAL XE(3),YE(3),A(3)
        INTEGER I,IEL,J,NEL
        LOGICAL OUT


        OUT = .FALSE.

c  Search whole array, first setting coords for a given element J
```

```
        DO 10 J = 1,TEMT
          JINC = 2*J
          IF (J.EQ.1) JINC = 1
          J1 = CURREL + J
          J2 = CURREL - J
          DO 15 K = J1,J2,-JINC
            IF (K.GE.1 .AND. K.LE.TEMT) THEN
            DO 20 I = 1,3
              IEL   = ELCONN(I,K)
              XE(I) = POINTX(IEL)*SCALE
              YE(I) = POINTY(IEL)*SCALE
 20         CONTINUE

c  Compute area of element K

            DET = XE(2)*(YE(3)-YE(1))+XE(3)*(YE(1)-YE(2))
            DET = DET + XE(1)*(YE(2)-YE(3))
            AREA = 0.5*DET

c  Find areal coords of zp,rp relative to element J

            A(1) = (ZP*(YE(2)-YE(3)) + RP*(XE(3)-XE(2)) +
     +             (XE(2)*YE(3)-XE(3)*YE(2)))/DET
            A(2) = (ZP*(YE(3)-YE(1)) + RP*(XE(1)-XE(3)) +
     +             (XE(3)*YE(1)-XE(1)*YE(3)))/DET
            A(3) = (ZP*(YE(1)-YE(2))+RP*(XE(2)-XE(1)) +
     +             (XE(1)*YE(2)-XE(2)*YE(1)))/DET

c  Test whether zp,rp is inside element J. ie 0<=A(I)<=1

            IF(A(1).GE.0.0.AND.A(1).LE.1.0) THEN
              IF(A(2).GE.0.0.AND.A(2).LE.1.0) THEN
                IF(A(3).GE.0.0.AND.A(3).LE.1.0) THEN
                  NEL = K
                  CURREL = NEL
                  GOTO 30
                ENDIF
              END IF
            END IF
            ENDIF
 15       CONTINUE
 10     CONTINUE

C  If it is not in any of the elements, then say so:
        OUT = .TRUE.

 30     RETURN
        END

c  ____________________________________________________________________

        SUBROUTINE FINDNN(N,NN,POSN)

C This calculates the element numbers of the three neighbouring elements
C to the one numbered N. It passes back thheir numbers in NN(3) and
C signals via POSN if N is on the axis (POSN='A') or on electrode surface
C (POSN='E') or at lefthand end ('L') or righthand end ('R').
C If N is surrounded by other elements then POSN='F' (for free).

$include: 'lens.def'

        CHARACTER*1 POSN
        INTEGER NN(3),NELEM(17)
        INTEGER FN,BN,N,LENGTH,NAX,NLH,NRH,NRE,BAND
        INTEGER I,J,K,L

        BAND = 17
        POSN = 'F'

C Is base of element N on the axis?  Zero NN at same time.

        NAX = 0
        DO 10 I=1,3
          IF (POINTY(ELCONN(I,N)).EQ.0.0) NAX = NAX + 1
```

```
          NN(I) = 0
 10     CONTINUE
        IF (NAX.EQ.2) THEN
          POSN = 'A'
          BAND = 8
          GOTO 18
        ENDIF

C Is element such that one side is on lefthand end?

        NLH = 0
        DO 12 I=1,3
          IF (POINTX(ELCONN(I,N)).EQ.0.0) NLH = NLH + 1
          NN(I) = 0
 12     CONTINUE
        IF (NLH.EQ.2) THEN
          POSN = 'L'
          GOTO 18
        ENDIF

C Is element such that one side is on righthand end?

        NRH = 0
        LENGTH = ETDE2X(TEDT)
        DO 14 I=1,3
          IF (POINTX(ELCONN(I,N)).EQ.LENGTH) NRH = NRH + 1
          NN(I) = 0
 14     CONTINUE
        IF (NRH.EQ.2) THEN
          POSN = 'R'
          GOTO 18
        ENDIF

C Is element on an electrode?
        NRE = 0

C Put code to find extrema here when element has one side on an electrode
        IF (NRE.EQ.2) POSN = 'E'

C Identify the elements adjacent to N
  18    DO 20 I=1,BAND
          NELEM(I) = 0
 20     CONTINUE

        K = 0

        DO 200 I=1,TEMT/2

C NN(K) will contain 3 element numbers of the next nearest neighbour
C elements. Count forwards first
          FN = N + I
          IF (FN.GT.TEMT) GOTO 60
          DO 50 J=1,3
            DO 40 NE=1,3
              IF (ELCONN(NE,FN).EQ.ELCONN(J,N)) THEN
                K = K + 1
                NELEM(K) = FN
                IF (K.EQ.BAND) GOTO 300
              ENDIF
 40         CONTINUE
 50       CONTINUE

C Backward count
 60       BN = N - I
          IF (BN.LT.1) GOTO 200
          IF ((BN.LT.1).AND.(FN.GT.TEMT)) GOTO 300
          DO 80 J=1,3
            DO 70 NE=1,3
              IF (ELCONN(NE,BN).EQ.ELCONN(J,N)) THEN
                K = K + 1
                NELEM(K) = BN
                IF (K.EQ.BAND) GOTO 300
              ENDIF
 70         CONTINUE
```

```
 80       CONTINUE
 200    CONTINUE

C Find elements with two nodes common to current element:
 300    L = 0
        DO I=1,BAND
         K = 0
         BN=NELEM(I)
         DO NE=1,3
          DO J=1,3
           IF (ELCONN(NE,BN).EQ.ELCONN(J,N)) K=K+1
          ENDDO
         ENDDO
         IF (K.EQ.2 .AND. BN.NE.0) THEN
          L = L + 1
          NN(L) = BN
         ENDIF
        ENDDO
        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE GRAD(M,EZM,ERM,ZC,RC,POS)

C Finds the electric field components EZM,ERM at the centre ZC,RC of cell M

$include: 'lens.def'
$include: 'trajec.def'
        INTEGER M,I,J
        REAL*8 EZM,ERM,X(3),Y(3),Z(3),D(3,2),DET,ZC,RC
        CHARACTER*1 POS

C Set up coordinates of element M

        DO 10 I=1,3
          J = ELCONN(I,M)
          X(I) = POINTX(J)*SCALE
          Y(I) = POINTY(J)*SCALE
          Z(I) = F(J)

C Does the current element have its base situated on the axis or at either
C end of the lens? If so reverse the sign of X or Y for the non-zero apex.
C If at righthand end then extend apex beyond length of lens.

          IF (POS.EQ.'A') THEN
            IF (POINTY(J).EQ.0.0) GOTO 10
            Y(I) = -Y(I)
          ENDIF
          IF (POS.EQ.'L') THEN
            IF (POINTX(J).EQ.0.0) GOTO 10
            X(I) = -X(I)
          ENDIF
          IF (POS.EQ.'R') THEN
            IF (POINTX(J).EQ.ETDE2X(TEDT)) GOTO 10
            X(I) = 2*ETDE2X(TEDT) - POINTX(J)
          ENDIF
C NOTE TO MP - Add code to look after case when POS='E' - element has
C one side on an electrode.

 10     CONTINUE

C Find the determinant of element M coordinates

        DET = X(2)*(Y(3)-Y(1)) + X(3)*(Y(1)-Y(2)) + X(1)*(Y(2)-Y(3))

        D(1,1) = (Y(2)-Y(3))/DET
        D(2,1) = (Y(3)-Y(1))/DET
        D(3,1) = (Y(1)-Y(2))/DET
        D(1,2) = (X(3)-X(2))/DET
        D(2,2) = (X(1)-X(3))/DET
        D(3,2) = (X(2)-X(1))/DET

        EZM = 0.0
```

```
        ERM = 0.0

        DO 20 I=1,3
          EZM = EZM + Z(I)*D(I,1)
          ERM = ERM + Z(I)*D(I,2)
 20     CONTINUE

        ZC = (X(1) + X(2) + X(3))/3.0
        RC = (Y(1) + Y(2) + Y(3))/3.0

        RETURN
        END


C -------------------------------------------------------------------

        SUBROUTINE ISITOUT(ZP,RP,OUT)

$include: 'trajec.def'
        LOGICAL OUT
        REAL*8 ZP,RP

        IF ((ZP.GT.XMAX*SCALE).OR.
     +      (ZP.LT.XMIN*SCALE).OR.
     +      (RP.GT.YMAX*SCALE)) THEN
          OUT = .TRUE.
        ENDIF

        RETURN
        END

C -------------------------------------------------------------------

        SUBROUTINE MAGNIF

        CALL WARN('Code not ready yet')

        RETURN
        END
C -------------------------------------------------------------------

        SUBROUTINE PLANE(AX,AY,AZ,X,Y,Z)

C  Fits a plane to the 3 points (ax,ay,az) and returns the value of z
C  at the point (x,y)

        REAL*8 X,Y,AX(3),AY(3),AZ(3)
        REAL*8 Z,A,B,C,DEN
        REAL*8 BX1,BX2,BY1,BY2,BZ1,BZ2

          BZ1 = AZ(1) - AZ(2)
          BZ2 = AZ(2) - AZ(3)
          BY1 = AY(1) - AY(2)
          BY2 = AY(2) - AY(3)
          BX1 = AX(1) - AX(2)
          BX2 = AX(2) - AX(3)
          DEN = BX1*BY2 - BX2*BY1

          A = (BZ1*BY2 - BZ2*BY1)/DEN
          B = (BZ2*BX1 - BZ1*BX2)/DEN
          C = AZ(1) - A*AX(1) - B*AY(1)

          Z = A*X + B*Y + C

        RETURN
        END

C -------------------------------------------------------------------

        SUBROUTINE SORTIT

c  Routine to order the mesh used in POTGEN.
c  Re-order the arrays IJK,X,Y, and F.
```

```
c  Uses an index pointer method to change the array numbering.
c  The arrays from POTGEN are numbered to minimize the bandwidth
c  in the finite element analysis. In order to search the arrays
c  logically in TRAJ they need to be numbered in a spatially
c  sequence, (ie with increasing z).

$include: 'lens.def'
        INTEGER INDEX(LIMPTS),IJKT(3,LIMPTS)
        REAL XC(LIMPTS),XEE(3),XX,YY
C       CHARACTER*12 FNAME
        CHARACTER*60 MESS

        XX = 0.0
        YY = 0.0

c  Initialize INDEX to the range 1 to TEMT

        DO 10 I = 1,TEMT
          INDEX(I) = I
10      CONTINUE

c  Find the centroids (XC,YC) of each element.
        MESS = 'Mesh sorting...finding element centroids...'
        CALL MBUSY(0,0,MESS)
        DO 30 I = 1,TEMT
          DO 20 J = 1,3
            IEL = ELCONN(J,I)
            XEE(J) = POINTX(IEL)
20        CONTINUE
          XC(I) = (XEE(1)+XEE(2)+XEE(3))/3.0E0
30     CONTINUE

c  Sorting Routine.

          MESS = 'Mesh sorting...exchanging pointers...'
          CALL MBUSY(0,0,MESS)
          CALL QSORT(TEMT,XC,INDEX)

c  The code above has renumbered I as INDEX(I).
c  Rearrange other arrays with INDEX(I) for I.

        DO 60 J=1,TEMT
          DO 60 I=1,3
            IJKT(I,J) = ELCONN(I,INDEX(J))
60      CONTINUE

c  Write contents of IJKT back into ELCONN

        DO 70 J = 1,TEMT
          DO 70 I = 1,3
            ELCONN(I,J) = IJKT(I,J)
70      CONTINUE

C  Save Modified, Sorted, Mesh File.  Use later when required - declare FNAME
C       FNAME = NAME
C       CALL CONCAT(FNAME,'.MMH')
C       OPEN(UNIT=21,FILE=FNAME,STATUS='UNKNOWN')
C       WRITE(21,80) TEMT,TPOINT,MB
C80     FORMAT(3I5)
C       WRITE(21,90) (XX,YY,I=1,TPOINT)
C90     FORMAT(2(2F10.4,4X))
C       WRITE(21,100) ((ELCONN(I,J),I=1,3),J=1,TEMT)
C100    FORMAT(12I5)
C       CLOSE(UNIT=21)
C       CALL MPUTS('Sorted connectivities in file: ')
C       CALL MPUTS(FNAME)
C       CALL MPUTS(' $')
C       CALL MBUSY(0,0,' ')

        RETURN
        END


C ---------------------------------------------------------------------
```

```
        SUBROUTINE WRTRAJ(NR,NIR)

$include: 'lens.def'
$include: 'trajec.def'
        CHARACTER*12 FNAME

        FNAME = NAME
        CALL CONCAT(FNAME,'.TRJ')

        IF (NIR.EQ.1) THEN
          OPEN(UNIT=21,FILE=FNAME,STATUS='UNKNOWN')
          WRITE(21,*) '# Set of rays from file: ',NAME
          WRITE(21,*) '# Angular range: ',THETA1,' to ',THETA2
          WRITE(21,*) '# No of rays: ',NR
          WRITE(21,*) '# Ray origin at: ',X00,',',Y00
          WRITE(21,*) '#'
        ENDIF
        WRITE(21,*) '# Angle of ray: ',THETA1+DTHETAR*(NIR-1)
        WRITE(21,*) '# No of points in ray: ',KOUNT
        WRITE(21,*) ' '
        DO 18 I = 1,KOUNT
          WRITE(21,12) YP(3,I)/SCALE,YP(4,I)/SCALE
18      CONTINUE
        IF (NIR.EQ.NR) CLOSE(UNIT=21)
10      FORMAT(I5)
12      FORMAT(2E25.16)
        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE SPLINE(X,Y,N,YP1,YPN,Y2)

C Routine from Press for cubic spline interpolation using arrays X and Y.
C Given arrays X(1:n) and Y(1:n) containing a tabulated function i.e.
C yi= f(xi),with x1<x2<...xn, and given values yp1 and ypn for the first
C derivative of the interpolating function at points 1 and n respectively,
C this routine returns an array y2(1:n) of length n which contains the
C second derivative of the interpolating function at the tabulated points
C xi. If yp1 and/or ypn are equal to 1.E30 or larger, the routine is signaled
C to set the corresponding boundary condition for a natural spline, with
C zero 2nd derivative on that boundary.
C Parameter NMAX is the largest anticipated value for N.

        INTEGER I,K,NMAX,N
        PARAMETER(NMAX=2000)
        REAL*8 YP1,YPN,X(N),Y(N),Y2(N),U(NMAX),P,QN,SIG,UN

        IF (YP1.GT..99D30) THEN
          Y2(1) = 0.0
          U(1)  = 0.0
        ELSE
          Y2(1) = -0.5
          U(1) = (0.3D1/(X(2)-X(1)))*((Y(2)-Y(1))/(X(2)-X(1))-YP1)
        ENDIF

        DO 11 I=2,N-1
          SIG = (X(I)-X(I-1))/(X(I+1)-X(I-1))
          P = SIG*Y2(I-1)+2.
          Y2(I) = (SIG-1.)/P
          U(I) = (6.*((Y(I+1)-Y(I))/(X(I+1)-X(I))-(Y(I)-Y(I-1))
     +          /(X(I)-X(I-1)))/(X(I+1)-X(I-1))-SIG*U(I-1))/P
 11     CONTINUE

        IF (YPN.GT..99D30) THEN
          QN = 0.
          UN = 0.
        ELSE
          QN = 0.5
          UN = (3./(X(N)-X(N-1)))*(YPN-(Y(N)-Y(N-1))/(X(N)-X(N-1)))
        ENDIF
        Y2(N) = (UN-QN*U(N-1))/(QN*Y2(N-1)+1.)
        DO 12 K=N-1,1,-1
```

```
          Y2(K) = Y2(K)*Y2(K+1)+U(K)
 12     CONTINUE
        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE SPLINT(XA,YA,Y2A,N,X,Y)

C Given the arrays XA(1:n) and YA(1:n) of length N, which tabulate a function
C (with the XAi's in order) and given the array Y2A(1:n) which is the output
C from SPLINE above, and given a value of X, this routine returns a cubic-
C spline interpolated value Y.

        INTEGER N,K,KHI,KLO
        REAL*8 X,Y,XA(N),YA(N),Y2A(N),A,B,H

        KLO = 1
        KHI = N

 1      IF (KHI-KLO.GT.1) THEN
          K = (KHI+KLO)/2
          IF (XA(K).GT.X) THEN
            KHI = K
          ELSE
            KLO = K
          ENDIF
          GOTO 1
        ENDIF

        H = XA(KHI)-XA(KLO)
        IF (H.EQ.0.) PAUSE
        A = (XA(KHI)-X)/H
        B = (X-XA(KLO))/H
        Y = A*YA(KLO)+B*YA(KHI)+
     +         ((A**3-A)*Y2A(KLO)+(B**3-B)*Y2A(KHI))*(H**2)/6.
        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE FINDAXPOT(EE,OVLY)

$include: 'lens.def'
$include: 'trajec.def'
        REAL EE
        REAL*8 TZ,TV,LENGTH,Z
        INTEGER I,J,K
        LOGICAL OVLY

        OVLY = .FALSE.
        IF (.NOT.SORTED) THEN
          CALL SORTIT
          SORTED = .TRUE.
        ENDIF

C Collect all the axial coordinates and their potentials into ZAX and VAX
        IF (.NOT.VAXED) THEN
          CALL MBUSY(0,0,'Calculating axial potential...')
          K = 1
          DO 10, I = 1,TEMT
          DO 10, J = 1,3
            IF (POINTY(ELCONN(J,I)).EQ.0.0) THEN
              TZ = POINTX(ELCONN(J,I))
              TV = F(ELCONN(J,I))
              IF (K.EQ.1) THEN
                ZAX(K) = TZ
                VAX(K) = TV
                K = K+1
              ELSE
                IF (TZ.EQ.ZAX(K-1)) GOTO 10
                ZAX(K) = TZ
                VAX(K) = TV
C               CALL PREAL('%  ',REAL(TZ))
```

```
C               CALL PREAL('   %$',REAL(TV))
                K = K+1
              ENDIF
            ENDIF
            NPTS = K-1
 10       CONTINUE

C Interpolate the axial potential onto NAXPTS evenly spaced intervals

          CALL SPLINE(ZAX,VAX,NPTS,1.D30,1.D30,VAXINT)
          LENGTH = DBLE(NODEX(TNODE))
          DELZ   = LENGTH/(NAXPTS-1)
          DO 20 K = 1,NAXPTS
            Z = (K-1)*DELZ
            CALL SPLINT(ZAX,VAX,VAXINT,NPTS,Z,VZINT(K))

C Protect against subsequent divide by zero errors
            IF (VZINT(K).LE.0.0) VZINT(K)=1.0D-10
C           CALL PREAL('  %',REAL(Z))
C           CALL PREAL('    %$',REAL(VZINT(K)))
 20       CONTINUE

C Now calculate the lens strength function T
          T(1) = 0.0
          DO 30 K=1,NAXPTS-2
            I = K+1
            J = K+2
            T(I) = (VZINT(J)-VZINT(K))/(2.0D0*DELZ*(EE+VZINT(I)))
 30       CONTINUE
          T(NAXPTS-1) = T(NAXPTS-2)
          T(NAXPTS) = T(NAXPTS-1)

C Now differentiate T wrt Z for T prime to be used to find the
C aberration coefficients

          DO 40, K = 2,NAXPTS-1
            I = K-1
            J = K+1
            TPRIME(K) = (T(J)-T(I))/(2.0D0*DELZ)
 40       CONTINUE
          TPRIME(1) = TPRIME(2)
          TPRIME(NAXPTS) = TPRIME(NAXPTS-1)
        ENDIF
        VAXED = .TRUE.
        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE AXPATHS(EE,PLOT,OVLY,SLOPE)

$include: 'lens.def'
$include: 'trajec.def'
$include: 'gracol.def'

        REAL EE
        REAL*8 Z,THETAR1,THETAR2,SLOPE
        INTEGER NR,XS,YS,NSTART
        LOGICAL OVLY,PLOT

        CALL MBUSY(0,0,'Calculating paraxial trajectories...')
        OPEN(UNIT=41,FILE='PARAX.TRJ',STATUS='UNKNOWN')

C First set up the values RHO(1) and RHO(2) for the start of each path.

        THETAR1 = PI*DBLE(THETA1/1.8D2)
        THETAR2 = PI*DBLE(THETA2/1.8D2)
        IF (NRAYS.GT.1) THEN
          DTHETAR = (THETAR2-THETAR1)/(NRAYS-1)
        ENDIF

C Set up the graphics screen with an outline of lens surfaces
        IF (PLOT) THEN
          CALL GCLS
```

```
          CALL DTRAJ(OVLY,XSCA,YSCA,RMIN)
          CALL GPROMP(1,TITLE)
          CALL GPROMP(2,'Hit ESCAPE to exit plotting')
          XORG = 50
          YORG = 20
          CALL GSPAL(RAIPAL)
          CALL GSCOL(WHITE)
        ENDIF

        NSTART = 1+INT(X00/DELZ)
        DO 3300 NR=1,NRAYS
          RAD(NSTART)   = Y00
          RAD(NSTART+1) = Y00 + (THETAR1 + (NR-1)*DTHETAR)*DELZ
          RHO(NSTART)   = RAD(NSTART)*((EE+VZINT(NSTART))**0.25)
          RHO(NSTART+1) = RAD(NSTART+1)*((EE+VZINT(NSTART+1))**0.25)

C Now compute Picht's Rho and estimate the radius of the path at each Z

          T(1)   = 0.0
          DO 323 K = NSTART, NAXPTS-2
            I = K+1
            J = K+2
            RHO(J) = 2.0D0*RHO(I)-RHO(K)
            RHO(J) = RHO(J)-0.1875D0*DELZ*DELZ*T(I)*T(I)*RHO(I)
 323      CONTINUE
          DO 324 K=NSTART+2,NAXPTS
            RAD(K) = RHO(K)/((EE+VZINT(K))**0.25)
 324      CONTINUE

C Plot the calculated trajectory

          IF (PLOT) THEN
            XS = XORG + INT(X00*XSCA)
            YS = YORG + INT((Y00-RMIN)*YSCA)
            WRITE(41,*) X00,Y00
            CALL GMOVE(XS,YS)
            DO 325 K=NSTART,NAXPTS-1
              Z = X00 + (K-NSTART+1)*DELZ
              XS = XORG + INT(Z*XSCA)
              YS = YORG + INT((RAD(K)-RMIN)*YSCA)
              CALL GLINE(XS,YS)
              WRITE(41,*) Z,RAD(K)
 325        CONTINUE
            XS = XORG + INT(X00*XSCA)
            YS = YORG + INT((-Y00-RMIN)*YSCA)
            CALL GMOVE(XS,YS)
            DO 326 K=NSTART,NAXPTS-1
              Z = X00 + (K-NSTART+1)*DELZ
              XS = XORG + INT(Z*XSCA)
              YS = YORG + INT((-RAD(K)-RMIN)*YSCA)
              CALL GLINE(XS,YS)
 326        CONTINUE
          ENDIF
 3300   CONTINUE

C Now search one of paths for the crossovers

        XSOLD = 0
        ICROSS = 0
        DO 3310,K=NSTART+1,NAXPTS-1
          Z = X00 + (K-NSTART)*DELZ
          IF ((RAD(K-1).NE.0.0).AND.((RAD(K)/RAD(K-1)).LT.0.0)) THEN
            ICROSS = ICROSS + 1
            SLOPE  = (RAD(K-1)-RAD(K))/(Z-XSOLD)
            CROSSZ(ICROSS) = XSOLD+RAD(K-1)/SLOPE

            IF (PLOT) THEN
            CALL PREAL('Xover at z = F%8.4 cm ',REAL(CROSSZ(ICROSS)))
            CALL PREAL('!40Slope =F%6.4 $',REAL(SLOPE))
            ENDIF
          ENDIF
          XSOLD = Z
 3310   CONTINUE
```

```
        CLOSE(UNIT=41)
        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE SPOTAT(L)

C Finds the crossover most distant from the source. This is taken as the
C focal point for the gun

$include: 'lens.def'
$include: 'trajec.def'
        REAL L
        INTEGER I

        L = 0.0
        DO 10, I=1,ICROSS
          IF(CROSSZ(I).GT.L) L = CROSSZ(I)
 10     CONTINUE

        RETURN
        END

C ---------------------------------------------------------------------

        SUBROUTINE QSORT(N,ARR,IBRR)

C Subroutine SORT2 from Press et al. p326. Sorts an array ARR(1:N) into
C ascending order using Quicksort, whilst making the corresponding re -
C arrangement of the array IBRR(1:N). Here it is midified to sort the
C X coordinates XC of the finite element centroids into ascending order
C whilst also rearranging the integer array INDEX into the corresponding
C sequence.

        INTEGER N,M,NSTACK,IBRR(N)
        REAL ARR(N)
        PARAMETER(M=7,NSTACK=50)

        INTEGER I,IB,IR,J,JSTACK,K,L,ISTACK(NSTACK),ITEMP
        REAL A,TEMP

        JSTACK = 0
        L = 1
        IR = N
 1      IF (IR-L.LT.M) THEN
C         Insertion sort when subarray is small enough
          DO 12 J=L+1,IR
            A = ARR(J)
            IB = IBRR(J)
            DO 11 I=J-1,1,-1
              IF (ARR(I).LE.A) GOTO 2
              ARR(I+1) = ARR(I)
              IBRR(I+1) = IBRR(I)
 11         CONTINUE
            I = 0
 2          ARR(I+1) = A
            IBRR(I+1) = IB
 12       CONTINUE
          IF (JSTACK.EQ.0) RETURN
C         Pop stack and begin a new round of partitioning
          IR = ISTACK(JSTACK)
          L  = ISTACK(JSTACK-1)
          JSTACK = JSTACK - 2
        ELSE
C         Choose median of left, right and centre elements as partitioning
C         element A. Also re-arrange so that A(L+1)<= A(1) <= A(IR)
          K         = (L+IR)/2
          TEMP      = ARR(K)
          ARR(K)    = ARR(L+1)
          ARR(L+1)  = TEMP
          ITEMP     = IBRR(K)
          IBRR(K)   = IBRR(L+1)
          IBRR(L+1) = ITEMP
```

```
          IF (ARR(L+1).GT.ARR(IR)) THEN
            TEMP      = ARR(L+1)
            ARR(L+1)  = ARR(IR)
            ARR(IR)   = TEMP
            ITEMP     = IBRR(L+1)
            IBRR(L+1) = IBRR(IR)
            IBRR(IR)  = ITEMP
          ENDIF
          IF (ARR(L).GT.ARR(IR)) THEN
            TEMP      = ARR(L)
            ARR(L)    = ARR(IR)
            ARR(IR)   = TEMP
            ITEMP     = IBRR(L)
            IBRR(L)   = IBRR(IR)
            IBRR(IR)  = ITEMP
          ENDIF
          IF ( ARR(L+1).GT.ARR(L)) THEN
            TEMP      = ARR(L+1)
            ARR(L+1)  = ARR(L)
            ARR(L)    = TEMP
            ITEMP     = IBRR(L+1)
            IBRR(L+1) = IBRR(L)
            IBRR(L)   = ITEMP
          ENDIF
C         Initiate pointers for partitioning
          I  = L + 1
          J  = IR
          A  = ARR(L)
          IB = IBRR(L)

C  Beginning of innermost loop
C         Scan up to find element > A
 3        CONTINUE
            I = I + 1
            IF (ARR(I).LT.A) GOTO 3
C         Scan down to find element < A
 4        CONTINUE
            J = J - 1
            IF (ARR(J).GT.A) GOTO 4
C         If pointers crossed. Exit with partitioning complete.
          IF (J.LT.I) GOTO 5
C         Exchange elements of both arrays.
          TEMP    = ARR(I)
          ARR(I)  = ARR(J)
          ARR(J)  = TEMP
          ITEMP   = IBRR(I)
          IBRR(I) = IBRR(J)
          IBRR(J) = ITEMP
          GOTO 3
C  End of innermost loop.

 5        ARR(L)  = ARR(J)
          ARR(J)  = A
          IBRR(L) = IBRR(J)
          IBRR(J) = IB
          JSTACK  = JSTACK + 2
C  Push pointers to larger subarray on stack,process smaller subarray
C  immediately.
          IF (JSTACK.GT.NSTACK) THEN
            CALL PINT('JSTACK = %',JSTACK)
            CALL PINT('   N = %',N)
            CALL PINT('   NSTACK = %$',NSTACK)
            CALL WARN('Fatal Error -- NSTACK too small in QSORT')
            RETURN
          ENDIF
          IF (IR-I+1.GE.J-1) THEN
            ISTACK(JSTACK)   = IR
            ISTACK(JSTACK-1) = I
            IR               = J - 1
          ELSE
            ISTACK(JSTACK)   = J - 1
            ISTACK(JSTACK-1) = L
            L                = I
          ENDIF
```

```
        ENDIF
        GOTO 1
        END


C ---------------------------------------------------------------------
C New work by Andrew Jackson:
C (& alterations to original code)
C
C v1.23 28/2/97
C ---------------------------------------------------------------------
        SUBROUTINE WritePot
$include: 'lens.def'
$include: 'trajec.def'
        PARAMETER (NMAX=10)
        DIMENSION nCz(NMAX),nCr(NMAX)
        REAL*8 YiK(4),Z,R,Pot,Dif,nCz,nCr
        INTEGER nth,I,res,cycle
        LOGICAL URK
        COMMON  /SPLIN/ nth,nCz,nCr
        COMMON  /TEMP/  cycle
        cycle=1

        res=100

        OPEN(UNIT=21,FILE='AxNodes.dat',STATUS='UNKNOWN')
        WRITE(21,*) '# Axial node potential data:'
        DO I=TPOINT,1,-1
         IF (POINTY(I).EQ.0.0d0) THEN
          WRITE(21,112) POINTX(I)*SCALE*100.0,F(I)
         ENDIF
        ENDDO
        CLOSE(UNIT=21)

c       OPEN(UNIT=21,FILE='svd.dat',STATUS='UNKNOWN')
c       R=0.0d0
c       WRITE(21,*) '# Potential dist for rays'
c       DO I=1,res
c        Z=I*(0.101/res)
c        Yik(3)=Z
c        Yik(4)=R
c        CALL MeshSpline(Yik,URK)
c        CALL nthaeval(nth,nCz,Yik(3),Pot)
c        CALL nthadiff(nth,nCz,Yik(3),Dif)
c         WRITE(21,113) Z,Pot,Dif
c       ENDDO
c        CLOSE(UNIT=21)

 112    FORMAT(2F15.8)
 113    FORMAT(3F15.8)

        RETURN
        END

C ---------------------------------------------------------------------
        SUBROUTINE FINELABBERS
C  Take X-over points from FinEl Calc and find spherical abberation CS
$include: 'lens.def'
$include: 'trajec.def'
        PARAMETER (NMAX=20,MMAX=10)
        REAL*8 AAB(5,5),BAB(5,5),alpha,slope,sig(NMAX),AbCo(MMAX)
        REAL*8 xC(NMAX),yC(NMAX),chisq,covar(MMAX,MMAX)
        INTEGER ABNP,I,J,R,ABM,ABN

C Set number of coefficient to find (f+Cs3+Cs5+...):
        ABNP = 3

C  Least squares svd algorithm:
        DO I=1,ICROSS
         sig(I)=1.0d0
         xC(I)=DATAN(DBLE(CROSSLOPE(I)))
         yC(I)=DBLE(CROSSZ(I))
        ENDDO
        CALL SVDFIT(xC,yC,sig,ICROSS,AbCo,ABNP,chisq,2)
```

```
        CALL SVDVAR(ABNP,covar)

C  Tranfer results to an array in a shared common block:
        DO I=1,ABNP
         AbCoErr(I,1)=AbCo(I)
         AbCoErr(I,2)=SQRT(covar(I,I))
        ENDDO

        RETURN
        END
C ---------------------------------------------------------------------
        SUBROUTINE ABBERS(TRAJCODE)
$include: 'lens.def'
$include: 'trajec.def'
         INTEGER TRAJCODE

         IF (TRAJCODE.EQ.1 .AND. ICROSS.EQ.0) THEN
           CALL WARN('Need at least 1 X-over for this calculation.')
           RETURN
         ELSEIF (TRAJCODE.EQ.1) THEN
           CALL FINELABBERS
         ELSEIF (TRAJCODE.EQ.2) THEN
           CALL PARAXABBERS
         ENDIF
         IF (TRAJCODE.EQ.1) THEN
           CALL PREAL('Position of focus:F%10.4 cm$',AbCoErr(1,1))
           CALL PREAL('+/-:F%10.4 cm$',AbCoErr(1,2))
           CALL PREAL('3rd order:F%10.4 cm$',AbCoErr(2,1))
           CALL PREAL('+/-:F%8.2 cm$',AbCoErr(2,2))
           CALL PREAL('5th order:F%10.4 cm$',AbCoErr(3,1))
           CALL PREAL('+/-:F%8.2 cm$',AbCoErr(3,2))
         ELSEIF (TRAJCODE.EQ.2) THEN
           CALL PREAL('Aber.coeffs: Spherical:F%8.1 mm',REAL(CS))
           CALL PREAL('!40Chromatic:F%7.1 mm$',REAL(CC))
         ENDIF

         RETURN
         END

C ---------------------------------------------------------------------
       SUBROUTINE PATHS(VP,EPS,OVLY)
$include: 'lens.def'
$include: 'trajec.def'
$include: 'gracol.def'
        LOGICAL OVLY,OUT,PLOTEL,CROSSED
        INTEGER KEY,NP,I,J,IEL,sign1,sign2,INTK,OLDKOUNT
        REAL*8 TSTART,TEND,LINPOT,Zinit,Rinit,VPP,Hinit,Hmax
        REAL*8 Xpawet,Ypawet
        REAL*8 H1,YST(4),Y(4),HMIN,errs
        REAL Slope,Inter,reslutn
        EXTERNAL LINPOT

C Write out potential data etc for analysis:
        CALL WritePot
        OPEN(UNIT=23,FILE='Z.dat',STATUS='UNKNOWN')
        OPEN(UNIT=24,FILE='R.dat',STATUS='UNKNOWN')
        OPEN(UNIT=25,FILE='E.dat',STATUS='UNKNOWN')

        PLOTEL = .FALSE.
        ICROSS = 0
        NP=4
        reslutn=0.2d0

C  Define angular variation of rays:
        THETAR = PI*DBLE(THETA1/1.8D2)
        IF (NRAYS.GT.1) THEN
         DTHETAR = PI*(THETA2-THETA1)/((NRAYS-1)*180.0)
        ENDIF

C  Sort the data if necessary:
        IF (.NOT.SORTED) THEN
          CALL SORTIT
          SORTED = .TRUE.
        ENDIF
```

```
        OVLY = .TRUE.
        CALL GCLS
        CALL DTRAJ(OVLY,XSCA,YSCA,RMIN)
        CALL GPROMP(1,TITLE)
        CALL GPROMP(2,'Hit ESC to escape')
        XORG = 50
        YORG = 20
        CALL GSPAL(RAIPAL)
        CALL GSCOL(WHITE)

C  Main Loop around NRAYS
        TSTART = 0.0
        KMAX = LIMNNT
        TEND = 0.0
        errs=DBLE(EPS)

C  Make a rough estimate of transit time through structure
C  Then find timestep from that:
        DO 350 I = 1,TEDT
          VEL = SQRT(2*EMR*ETDPOT(I))
          IF (VEL.NE.0.0) THEN
            FLEN = (ETDE2X(I) - ETDE1X(I))*SCALE
            TEND = TEND + FLEN/VEL
          ENDIF
 350    CONTINUE
        TEND = TEND
        CALL PREAL('Transition time %10.4 secs$',REAL(TEND))
        Hinit = (TEND-TSTART)/200.0d0

C  Find the limiting values of the electrodes outside which the mesh
C  does not extend
        CALL BOUNDELS

C  Plot nodes for all the elements:
        IF (PLOTEL) THEN
         DO J = 1,TEMT
          DO I = 1,3
           IEL   = ELCONN(I,J)
           Xpawet=DBLE(POINTX(IEL)*SCALE)
           Ypawet=DBLE(POINTY(IEL)*SCALE)
           CALL SCLINE(Xpawet,Ypawet,Xpawet,Ypawet)
          ENDDO
         ENDDO
        ENDIF

C Main ray loop:
        DO 500 I = 1,NRAYS
          CALL TKHIT(KEY)
          CALL PINT('Path % ',I)
          CURREL = 1

C  Correct initial electron energy by potential at current point
          Zinit = X00*SCALE
          Rinit = Y00*SCALE
          VPP = VP + SQRT(2*LINPOT(Zinit,Rinit)*EMR)

C  Set up initial conditions
          YST(1) = VPP*COS(THETAR)
          YST(2) = VPP*SIN(THETAR)
          YST(3) = Zinit
          YST(4) = Rinit
          DXSAV = H1
          WRITE(23,*) YST(1)**2+YST(2)**2,LINPOT(Zinit,Rinit),EMR
          WRITE(23,*) VP,VPP

C Integration loop for each path.  Initialising:
        KOUNT = 1
        INTK = -1
        time = 0.0d0
        OUT = .FALSE.
        CROSSED = .FALSE.

C Store initial conditions and set up calc array:
```

```
        DO J=1,NP
         Y(J)=YST(J)
         YP(J,KOUNT)=Y(J)
        ENDDO
        Hmin = 0.0d0
c       Hmin = Hinit*errs
        Hmax = Hinit*reslutn
        H1   = Hmax/2.0d0

C Main loop over the integration for a single ray:
  400   OLDKOUNT=KOUNT

C Find the approximation to the potential surface at Z,R:
        CALL MeshSpline(Y,OUT)

C Hamming Predictor-Corrector:
        CALL HPCD(Y,NP,H1,Hmax,Hmin,errs,INTK)

C Increment the time counter:
        time=time+H1

C Do we have a new ray section?
        KOUNT=INT(time/Hmax)+2
        IF (KOUNT.NE.OLDKOUNT) THEN

C Store electron path in an array:
         DO J=1,NP
          YP(J,KOUNT)=Y(J)
         ENDDO

C Plot new line section:
        CALL SCLINE(YP(3,KOUNT-1),YP(4,KOUNT-1),YP(3,KOUNT),YP(4,KOUNT))

C Check for cross-over:
          sign1 = SIGN(1,YP(4,KOUNT))
          sign2 = SIGN(1,YP(4,KOUNT-1))
          IF (sign1.NE.sign2.AND.(THETAR.GT.0.0.OR.Rinit.NE.0.0)) THEN
            Slope  = (YP(4,KOUNT-1)-YP(4,KOUNT))
     +                /(YP(3,KOUNT-1)-YP(3,KOUNT))
            Inter  = YP(4,KOUNT)-Slope*YP(3,KOUNT)
            Inter  = -(Inter/Slope)/SCALE
            IF (.NOT.CROSSED) THEN
             ICROSS = ICROSS + 1
             CROSSZ(ICROSS) = Inter
             CROSSLOPE(ICROSS) = Slope
             CROSSED = .TRUE.
            ELSE
             IF (Inter.GT.CROSSZ(ICROSS)) CROSSZ(ICROSS)=Inter
             CROSSLOPE(ICROSS) = Slope
            ENDIF
C Output cross-over to user:
            CALL PREAL('Xover at z = F%6.2 cm ',CROSSZ(ICROSS))
c           CALL PREAL('!40Slope =F%6.4 $',REAL(SLOPE))
          ENDIF

C End of new-section IF construct:
        ENDIF

C If the electron has left the system, quit the loop:
c        IF (OUT) WRITE(*,*) 'PATHS:OUT'
        IF (OUT) GOTO 410

C Check for escape key:
        CALL TKHIT(KEY)
c       IF (KEY.EQ.27) WRITE(*,*) 'PATHS:ESC'
        IF (KEY.EQ.27) GOTO 410

C Loop back through integration...
        GOTO 400

C Plot the mirror image path
 410    KEY=0
        WRITE(23,*)
        WRITE(24,*)
```

```
        WRITE(25,*)
        WRITE(*,*) KOUNT
        DO 449 J=1,KOUNT
 449    CONTINUE
C Write mirrored path to file and plot it too:
        DO 450 J=2,KOUNT-1
          CALL SCLINE(YP(3,J-1),-YP(4,J-1),YP(3,J),-YP(4,J))
 450    CONTINUE

C Write trajectories to data file '<NAME>.TRJ':
        CALL WRTRAJ(NRAYS,I)

C Write the number of points in this ray to the screen:
        CALL PINT('  % pts$',KOUNT)

C If escape has been pressed, then quit from PATHS:
        CALL TKHIT(KEY)
        IF (KEY.EQ.27) GOTO 600

C Increase ray launch angle:
 490    THETAR = THETAR + DTHETAR
 500    CONTINUE

C End:
 600    CLOSE(UNIT=23)
        CLOSE(UNIT=24)
        CLOSE(UNIT=25)
        RETURN
        END

C ---------------------------------------------------------------------       -
        SUBROUTINE RungeKutta(Y,NP,h,hmin,hmax,errs,INTK)
$include: 'lens.def'
$include: 'trajec.def'

        REAL*8 Y(4),Y1(4),Y2a(4),Y2b(4),DYDX(4),h
        REAL*8 hmin,hmax,errs,Derr
        INTEGER I,INTK,FLAG

C  Once with step = h:
 17     FLAG=0
        CALL RK4(Y,DYDX,NP,0.0d0,h,Y1)
C  And twice with step = h/2:
        CALL RK4(Y,DYDX,NP,0.0d0,0.5d0*h,Y2a)
        CALL RK4(Y2a,DYDX,NP,0.0d0,0.5d0*h,Y2b)

        IF (INTK.EQ.-2) THEN
C  Adaptive extrapolative RK error level check:
         DO I=1,NP
          Derr=Y2b(I)-Y1(I)
          IF (Derr.GE.errs .AND. 0.5d0*h.GT.hmin) FLAG=1
          IF (Derr.LT.0.02d0*errs .AND. 2.0d0*h.LT.hmax) FLAG=2
         ENDDO
        ENDIF
        IF (FLAG.EQ.1) h=0.5d0*h
        IF (FLAG.EQ.2) h=2.0d0*h

C IF stepsize has changed, redo last step:
c       IF (FLAG.NE.0) GOTO 17

C  Extrapolate:
        DO I=1,NP
         Y(I)=Y2b(I)+(1.0d0/15.0d0)*(Y2b(I)-Y1(I))
C  Or normal Runge-Kutta:
c         Y(I)=Y1(I)
        ENDDO

        RETURN
        END

C ---------------------------------------------------------------------
        SUBROUTINE HPCD(Y,NP,h,hmax,hmin,errs,INTK)
$include: 'lens.def'
$include: 'trajec.def'
```

```
        REAL*8 Y(4),DYDX(4),h,Yold(4),DQDX(4)
        REAL*8 DH(4,-8:1),DYH(4,-8:1),Del(4),P(4),Q(4)
        REAL*8 DH2(4,-8:1),DYH2(4,-8:1),Yt(4)
        REAL*8 errs,hmax,hmin
        INTEGER I,J,INTK,RKSTPS,FLAG
        COMMON /HPCDBK/ DH,DYH,Del

C Initialising etcetera:
c        RKSTPS = 5
        RKSTPS = 3
        IF (INTK.EQ.-1) THEN
         DO I=1,NP
          DO J=-RKSTPS,1
           DH(I,J)=0.0d0
           DYH(I,J)=0.0d0
          ENDDO
         ENDDO
         INTK=0
        ENDIF

C  First shift the elements of the history arrays:
        DO I=1,NP
         DO J=-RKSTPS,0
          DH(I,J)=DH(I,J+1)
          DYH(I,J)=DYH(I,J+1)
         ENDDO
         Yold(I)=Y(I)
        ENDDO

C  Use this line for extrapolative RK:
        INTK=0
C  Use this line to use adaptive RK:
c        INTK=-2

C  Now use Runge-Kutta or Hamming depending on steps taken:
 10     FLAG = 0
        IF (INTK.LE.RKSTPS) THEN
         CALL RungeKutta(Y,NP,h,hmin,hmax,errs,INTK)
         CALL DERIVS(Y,DYDX)
         DO I=1,NP
          DH(I,1)=Y(I)-Yold(I)
          DYH(I,1)=DYDX(I)
          Del(I)=0.0d0
         ENDDO
         INTK=INTK+1
        ELSE
C  Hamming predictor corrector:
 20      DO I=1,NP
          P(I)=(h/3.0d0)*(7.0d0*(DYH(I,0)+DYH(I,-2))
     +         -8.0d0*DYH(I,-1))-DH(I,-2)
          Q(I)=Y(I)+P(I)+(116.0d0/125.0d0)*Del(I)
         ENDDO
         CALL DERIVS(Q,DQDX)
         DO I=1,NP
          DYH(I,1)=DQDX(I)
          DH(I,1)=(1.0d0/8.0d0)*(DH(I,0)+DH(I,-1)+3.0d0*h
     +           *(DYH(I,1)+2.0d0*DYH(I,0)-DYH(I,-1)))
          Del(I)=DH(I,1)-P(I)
C  Change integration step or carry on:
          IF (Del(I).GE.errs .AND. 0.5d0*h.GE.hmin) FLAG=1
          IF (Del(I).LT.0.02d0*errs .AND. 2.0d0*h.LE.hmax) FLAG=2
          IF (FLAG.EQ.0) THEN
           DH(I,1)=DH(I,1)-(9.0d0/125.0d0)*Del(I)
           Y(I)=Y(I)+DH(I,1)
          ENDIF
         ENDDO
        ENDIF

C Under stepsize change condition, rearrange data to repeat calc:
        IF (FLAG.EQ.1) THEN
c        DO I=1,NP
c         DH2(I,-1)=(176.0d0*DH(I,0)+41.0d0*DH(I,-1)+DH(I,-2))/256.0d0
c     +  -h*(-15.0d0*DYH(I,0)+90.0d0*DYH(I,-1)+15.0d0*DYH(I,-2))/256.0d0
```

```
c         DH2(I,-2)=DH(I,-1)
c        DH2(I,-3)=(-112.0d0*DH(I,0)+109.0d0*DH(I,-1)+DH(I,-2))/256.0d0
c     +  -h*(3.0d0*DYH(I,0)+54.0d0*DYH(I,-1)-27.0d0*DYH(I,-2))/256.0d0
c         DH2(I,-4)=DH(I,-2)
c         Del(I)=0.0d0
c        ENDDO
c        DO I=1,NP
c         DO J=0,-2,-1
c          DH(I,J)=DH2(I,J)
c          Yt(I)=DH2(I,J)
c          CALL DERIVS(Yt,DYDX)
c           DYH(I,J)=DYDX(I)
c         ENDDO
c        ENDDO
         h=0.5d0*h
c        INTK=RKSTPS+1
         INTK=0
        ENDIF
        IF (FLAG.EQ.2) THEN
c        DO I=1,NP
c         DO J=0,-2,-1
c          DH(I,J)=DH(I,J*2)
c           DYH(I,J)=DYH(I,J*2)
c         ENDDO
c         Del(I)=0.0d0
c        ENDDO
c        h=2.0d0*h
c        INTK=RKSTPS+1
         INTK=0
        ENDIF

C IF stepsize has changed, redo last step:
        IF (FLAG.NE.0) GOTO 10

        RETURN
        END

C ------------------------------------------------------------------------
        SUBROUTINE MeshSpline(Y,OUT)
$include: 'lens.def'
$include: 'trajec.def'
        PARAMETER (MMAX=10,NMAX=20)
        REAL*8 Z,R,Y(4),LINPOT
        REAL*8 ZC,RC,negZ,posZ,negR,posR
        REAL*8 nCz(MMAX),nCr(MMAX),nZ(NMAX),nR(NMAX)
        REAL*8 nPz(NMAX),nPr(NMAX),noksc
        REAL*8 nZm,pZm,nRm,pRm,Elk,Pot,chisq,sigz(NMAX),sigr(NMAX)
        INTEGER CEL,NN(3),nth,nok
        LOGICAL OUT,NEG,AXIS
        CHARACTER*1 POSN
        EXTERNAL LINPOT
        COMMON  /SPLIN/ nth,nCz,nCr

        OUT  = .FALSE.
        NEG  = .FALSE.
        ma   = 4

C  SVD point parameter:
        nok   = 2*ma
C  Spline point parameter
c        nok   = ma

        nth  = ma
        noksc = 1.75d0*(DBLE(nok)/2.0d0)
        Z = Y(3)
        R = Y(4)
        IF (R.LT.0.0) THEN
          NEG = .TRUE.
        ENDIF
        DO I=1,NMAX
         sigz(I)=1.0d0
         sigr(I)=1.0d0
        ENDDO
```

```
C  Find element number which electron occupies
         CALL FINDEL(Z,ABS(R),CEL,OUT)

C  Find nearest neighbour elements
         CALL FINDNN(CEL,NN,POSN)

C  Find Z and R ranges such that polyn. approx. covers more than one element
         posZ=+1.0d-20
         negZ=-1.0d-20
         posR=+1.0d-20
         negR=-1.0d-20
         AXIS=.FALSE.
         DO i=1,3
           IF (POSN.EQ.'F' .AND. NN(i).EQ.0) WRITE(*,*) POSN
           IF (POSN.EQ.'A' .AND. NN(i).EQ.0) THEN
             CALL FINDELCENT(CEL,ZC,RC)
             AXIS=.TRUE.
             RC=-RC
             NN(i)=CEL
           ELSE
             CALL FINDELCENT(NN(i),ZC,RC)
c            CALL SCLINE(ZC,RC,ZC,RC)
           ENDIF
           IF (NEG) RC=-RC
           ZC=ZC-Z
           RC=RC-R
           IF (ZC.GT.posZ) posZ = ZC
           IF (ZC.LT.negZ) negZ = ZC
           IF (RC.GT.posR) posR = RC
           IF (RC.LT.negR) negR = RC
         ENDDO

C  Asymmetric spline axial symmetry fix:
c        IF (AXIS) negR=-posR
C  Symmetric spline span averaging:
         posR=noksc*(ABS(posR)+ABS(negR))/2.0d0
         posZ=noksc*(ABS(posZ)+ABS(negZ))/2.0d0
         negR=-posR
         negZ=-posZ

C  Find the minimum span size at the current point and check that
C  the current splines fit within it:
         CALL MinSplin(Z,R,nZm,pZm,nRm,pRm)
         IF (negZ.LT.nZm) negZ=nZm
         IF (posZ.GT.pZm) posZ=pZm
         IF (posR.GT.pRm) posR=pRm

C  Construct the two polynomial appoximations to the potential:
         DO i=1,nok
C  Symmetric spline def:
         nZ(i)=Z+negZ+DBLE(i-1)*(ABS(posZ)+ABS(negZ))/DBLE(nok-1)
         nPz(i)=LINPOT(nZ(i),R)
         nR(i)=R+negR+(i-1)*(ABS(posR)+ABS(negR))/(nok-1)
         nPr(i)=LINPOT(Z,nR(i))
C  Asymmetric spline def:
c        nZ(i)=Z+noksc*negZ+noksc*(i-1)*(posZ-negZ)/(nok-1)
c        nPz(i)=LINPOT(nZ(i),R)
c        nR(i)=R+noksc*negR+noksc*(i-1)*(posR-negR)/(nok-1)
c        nPr(i)=LINPOT(Z,nR(i))
         ENDDO

C  Plot spans of splines on the screen:
cc       DO i=1,nok
cc        CALL SCLINE(nZ(i),R,nZ(i),R)
cc        CALL SCLINE(Z,nR(i),Z,nR(i))
cc       ENDDO
cc       CALL SCLINE(Z,R,Z,R)

C Find the approximation coefficients:
c         CALL nthapprox(nok,nZ,nPz,nCz)
c        CALL nthapprox(nok,nR,nPr,nCr)

C Find the polynomial coefficients via least-sqrs SVDecomp:
         CALL SVDFIT(nZ,nPz,sigz,nok,nCz,ma,chisq,1)
```

```
        CALL SVDFIT(nR,nPr,sigr,nok,nCr,ma,chisq,1)

        RETURN
        END

C ---------------------------------------------------------------------
        SUBROUTINE DERIVS(Y,DYDX)
$include: 'lens.def'
$include: 'trajec.def'
        PARAMETER (NMAX=10)
        DIMENSION nCz(NMAX),nCr(NMAX)
        REAL*8 Y(4),DYDX(4),EZP,ERP,nCz,nCr,ZC,RC,Pot,LINPOT,vsq
        INTEGER nth,CEL,NN(3),cycle
        LOGICAL OUT,DATOUT
        CHARACTER*1 POSN
        COMMON  /SPLIN/ nth,nCz,nCr
        COMMON  /TEMP/  cycle
        EXTERNAL LINPOT

C Flag for detailed trajectory output:
        DATOUT=.TRUE.

C  Find differential of electric field at Z,R from polyn. approx.
        CALL nthadiff(nth,nCz,Y(3),EZP)
        CALL nthadiff(nth,nCr,Y(4),ERP)
        CALL nthaeval(nth,nCz,Y(3),Pot)


C  Old style linear gradient code:
c       CALL FINDEL(Y(3),ABS(Y(4)),CEL,OUT)
c       CALL FINDNN(CEL,NN,POSN)
c       CALL GRAD(CEL,EZP,ERP,ZC,RC,POSN)
c       Pot=LINPOT(Y(3),Y(4))


C Check by pot output
        IF (DATOUT) THEN
         cycle=cycle+1
         IF (cycle.GT.25) THEN
         vsq=Y(1)**2+Y(2)**2
         WRITE(23,131) Y(3),Pot,EZP
         WRITE(24,131) Y(4),Pot,ERP
         WRITE(25,132) time,Pot-vsq/(2.0d0*EMR)
         cycle=1
         ENDIF
        ENDIF
 131    FORMAT(2F14.6,E13.5)
 132    FORMAT(2E14.6)


C  Return information to integration routines in the required format.
        DYDX(1) = EMR*EZP
        DYDX(2) = EMR*ERP
        DYDX(3) = Y(1)
        DYDX(4) = Y(2)

        RETURN
        END

C ---------------------------------------------------------------------------
        SUBROUTINE MinSplin(X,Y,nXm,pXm,nYm,pYm)
$include: 'lens.def'
$include: 'trajec.def'
        REAL*8 X,Y,nXm,pXm,nYm,pYm,Ex1,Ey1,Ex2,Ey2
        REAL*8 Xtst,Ytst,DeltX,DeltY,Grad
        INTEGER I,Sign

        nXm=XMIN*SCALE-X
        pXm=XMAX*SCALE-X
        nYm=YMIN*SCALE-Y
        pYm=YMAX*SCALE-Y

C  Loop over electrodes:
        DO I = 1,TEDT
```

```
         Ex1 = ETDE1X(I)*SCALE-X
         Ex2 = ETDE2X(I)*SCALE-X
         Ey1 = ETDE1Y(I)*SCALE-Y
         Ey2 = ETDE2Y(I)*SCALE-Y
         DeltX = Ex2-Ex1
         DeltY = Ey2-Ey1
C  Check for new y-min/y-max:
         Sign = DSIGN(1,Ex1) + DSIGN(1,Ex2)
         IF (Sign.EQ.0.AND.(DeltX.GT.0.0d0.OR.DeltX.LT.0.0d0)) THEN
          Grad = DeltY/DeltX
          Ytst = Ey1 - Grad*Ex1
          IF (Ytst.GT.0.0d0 .AND. Ytst.LT.pYm) pYm=Ytst
          IF (Ytst.LT.0.0d0 .AND. Ytst.GT.nYm) nYm=Ytst
         ENDIF
C  Check for new x-min/x-max:
         Sign = DSIGN(1,Ey1) + DSIGN(1,Ey2)
         IF (Sign.EQ.0.AND.(DeltY.GT.0.0d0.OR.DeltY.LT.0.0d0)) THEN
          Grad = DeltX/DeltY
          Xtst = Ex1 - Grad*Ey1
          IF (Xtst.GT.0.0d0 .AND. Xtst.LT.pXm) pXm=Xtst
          IF (Xtst.LT.0.0d0 .AND. Xtst.GT.nXm) nXm=Xtst
         ENDIF
        ENDDO

        RETURN
        END

C -------------------------------------------------------------------------
        SUBROUTINE SCLINE(z1,r1,z2,r2)
$include: 'lens.def'
$include: 'trajec.def'
        REAL*8 z1,r1,z2,r2
        INTEGER px,py

        px=XORG+INT(z1*XSCA/SCALE)
        py=YORG+INT((r1/SCALE-RMIN)*YSCA)
        CALL GMOVE(px,py)
        px=XORG+INT(z2*XSCA/SCALE)
        py=YORG+INT((r2/SCALE-RMIN)*YSCA)
        CALL GLINE(px,py)

        END

C -------------------------------------------------------------------------
        FUNCTION LINPOT(X,Y)

C  Code first finds the element that X and Y lie in.  Then
C  take the number N of an element and fit a plane to the
C  three corners.  Then it finds the value of z  at the point
C  x,y entered and outputs it for further work. Used to linearly
C  interpolate V for the element N.

$include: 'lens.def'
$include: 'trajec.def'
        INTEGER N,I,J
        REAL*8 X,Y,LINPOT
        REAL*8 AX(3),AY(3),AZ(3)
        LOGICAL ELOUT,YNEG

        IF (Y.LT.0.0) THEN
         Y = -Y
         YNEG=.TRUE.
        ELSE
         YNEG=.FALSE.
        ENDIF

        CALL FINDEL(X,Y,N,ELOUT)
        DO I = 1,3
          J = ELCONN(I,N)
          AX(I) = POINTX(J)*SCALE
          AY(I) = POINTY(J)*SCALE
          AZ(I) = F(J)
        ENDDO
```

```
        CALL PLANE(AX,AY,AZ,X,Y,LINPOT)

        IF (YNEG) Y = -Y

        RETURN
        END

C -------------------------------------------------------------
       SUBROUTINE FINDELCENT(N,ZC,RC)
$include: 'lens.def'
$include: 'trajec.def'
       INTEGER N
       REAL*8 ZC,RC
C
       ZC=0.0d0
       RC=0.0d0
       DO i=1,3
         j=ELCONN(i,N)
         ZC=ZC+POINTX(j)*SCALE
         RC=RC+POINTY(j)*SCALE
       ENDDO
       ZC=ZC/3.0d0
       RC=RC/3.0d0
C
       END

C -------------------------------------------------------------
      SUBROUTINE nthadiff(N,C,x,dy)
      PARAMETER (NMAX=10)
      DIMENSION C(NMAX)
      INTEGER N
      REAL*8 C,x,dy
C
      dy=0.0d0
      DO i=2,N
        dy=dy+C(i)*(i-1)*(x**(i-2))
      ENDDO
C
      END

C -------------------------------------------------------------
      SUBROUTINE nthaeval(N,C,x,y)
      PARAMETER (NMAX=10)
      DIMENSION C(NMAX)
      INTEGER N
      REAL*8 C,x,y
C
      y=C(1)
      DO i=2,N
        y=y+C(i)*(x**(i-1))
      ENDDO
C
      END

C -------------------------------------------------------------
      SUBROUTINE nthapprox(N,X,Y,C)
      PARAMETER (NMAX=10)
      DIMENSION X(NMAX),Y(NMAX),C(NMAX),A(NMAX,NMAX),B(NMAX,NMAX)
      INTEGER N
      REAL*8 X,Y,C,A,B
C
      DO j=1,N
       A(j,1)=1.0d0
       DO i=2,N
        A(j,i)=X(j)**(i-1)
       ENDDO
       B(j,1)=Y(j)
      ENDDO
C
      CALL GAUSSJ(A,N,NMAX,B,1,NMAX)
C
      DO j=1,N
       C(j)=B(j,1)
      ENDDO
```

```
C
      END

C ---------------------------------------------------------------------
      SUBROUTINE FUNCS(X,P,NP,I)
      IMPLICIT REAL*8 (a-h,o-z)
      PARAMETER(NMAX=20,MMAX=10)
      REAL*8 P(NMAX)
      INTEGER I,J

C  Standard nth order polynomial:
      IF (I.EQ.1) THEN
       P(1)=1.0d0
       DO J=2,NP
         P(J)=P(J-1)*X
       ENDDO
C  Abberation polynomial of form 1 - tan2(x) - tan4(x) - ...:
      ELSEIF (I.EQ.2) THEN
       P(1)=1.0d0
       DO J=2,NP
c         P(J)=-(DTAN(X))**(2*(J-1))
         P(J)=-(X**(2*(J-1)+1))/(TAN(X))
       ENDDO
      ENDIF

      RETURN
      END

C ---------------------------------------------------------------------
C  Routines adapted from Press et al:
C ---------------------------------------------------------------------
      SUBROUTINE GAUSSJ(A,N,NP,B,M,MP)
C ---------------------------------------------------------------------
      SUBROUTINE RK4(Y,DYDX,N,X,H,YOUT)
C ---------------------------------------------------------------------
      SUBROUTINE SVDFIT(X,Y,SIG,NDATA,A,MA,CHISQ,IFUNC)
C---------------------------------------------------------------------
      SUBROUTINE SVBKSB(M,N,B,X)
C---------------------------------------------------------------------
      SUBROUTINE SVDVAR(MA,CVM)
C---------------------------------------------------------------------
      SUBROUTINE SVDCMP(A,M,N)
C ---------------------------------------------------------------------
C End of TRAJEC.FOR
```