Computational Methods

Introduction

This appendix describes the methods used for computations within SIMION 3D Version 7.0. The discussion is somewhat generalized to avoid writing a tome on the subject.

SIMION 7.0 is an electrostatic and magnetic field modeling program. This means it is designed to model the electrostatic and magnetic fields and forces created by a collection of shaped electrodes given certain symmetry assumptions. Electrostatic and magnetic fields can be modeled as boundary value problem solutions of an elliptical partial differential equation called the Laplace equation.

The specific method used within SIMION is a finite difference technique called over-relaxation. This technique is applied to two or three dimensional potential arrays of points representing electrode (*pole*) and non-electrode (*non-pole*) regions. The objective is to obtain a best estimate of the potentials for those points within the array that depict non-electrode (*non-pole*) regions.

Relaxation techniques use iteration, a technique of successive approximation. Relaxation has the advantage that normal numerical computation errors are minimized, solutions are quite stable, and computer memory storage requirements are minimized. However, its disadvantage is that the exact number of iterations required for a given level of refining is quite variable and initially unknown for each specific solution.

In the discussions below references will be made mostly to electrostatic arrays. ***These discussions apply in the same manner to magnetic arrays unless otherwise noted.***

The Structure of the Potential array

In SIMION, the potential array is a one dimensional array of double precision points (*referencing methods make it appear two or three dimensional to the user*). Each array element represents a point within the two (*or three*) dimensional array. Each array element typically stores two pieces of information. The first is the current voltage at the point, and the second is whether or not the point is an electrode. Electrode points are marked by adding two times the maximum allowable array voltage to their actual voltage. ***SIMION has the ability to automatically re-scale a potential array with a larger offset voltage if the maximum allowable array voltage is exceeded.***

The array is considered packed when each point holds both voltage and electrode flag information. Certain functions like refining operate faster if the electrode flag information is held in a separate array. Each PA in RAM has its own separate buffer to support the unpacked potential array when needed.

The Relaxation Method

The relaxation method uses nearest neighbor points to obtain new estimates for each point. SIMION uses the nearest four points for 2D (*nearest six points for 3D*). The example below demonstrates the 2D array (*3D adds a point 6 above in z and a point 5 below in z for a total of six*):

$$\begin{array}{ccccc} & P_4 & & & \\ P_1 & P_0 & P_2 & 2D & P_{0new} = (P_1+P_2+P_3+P_4)/4 \\ & & & 3D & P_{0new} = (P_1+P_2+P_3+P_4+P_5+P_6)/6 \\ & P_3 & & & \end{array}$$

The equations above (*used to approximate Laplace Equation solutions*) estimate the new value of a point from the ***average*** value of its nearest four (*or six - 3D*) neighbors.

If each point within the potential array is estimated with this technique (*except electrode points*), we have performed a single iteration. Each time we scan through the array again (*another iteration*) changes made in previous iterations are propagated further throughout the potential array. ***Non-electrode points change less and less between successive iterations.*** At some point, when these changes are ***small enough***, the potential array can be said to be refined ***close enough*** for some purpose.

Over-Relaxation

If the equation above is used for refining potential arrays, a large number of iterations will be required. Over-relaxation speeds the refining process by increasing each voltage adjustment by some factor. Let's say that the new value of a point is computed to be 101 volts and its current value is 100 volts. The required adjustment would be +1 volt for simple relaxation. Over-relaxation would change this adjustment by multiplying it times a factor that ranges from 1 (*relaxation*) to 2 (*unstable over-relaxation*). ***SIMION's over-relaxation factor assumes the number one is added to it.*** Thus in SIMION, an over-relaxation factor of 0.9 applied to a +1 volt change results in a correction of:

1.9 volts = 1 * (1.0 + 0.9)

Dynamically Self-Adjusting Over-Relaxation

It turns out that for each different potential array there exists a unique over-relaxation factor that gives the fastest solution convergence. These ideal factors are generally in the range of 0.9 but they are different for each array. SIMION makes use of a dynamically self-adjusting factor to help avoid looking for the ideal over-relaxation factor. A normal SIMION refine uses the following factors:

ITERATION	OVER-RELAXATION FACTOR	
1	0.40	
2-10	0.67	
>10	F	= (F_{old} * Hist) + (1 - Hist) * (F_{max} - DF * Error)
Where:	F_{old}	= factor used in prior iteration
	Hist	= History factor (e.g. 0.70)
	F_{max}	= Max over-relaxation (e.g. 0.90)
	DF	= F_{max} - 0.40
and:	Error	= 1/(1 + (DEL/(2*DELM)))
Where:	DEL	= Max prior abs single point change
	DELM	= Convergence Goal (e.g. 0.005)

The self-adjusting characteristics use the maximum single point absolute voltage change to drive the factor. A maximum over-relaxation factor limits the maximum value, and a history factor limits the

dynamic rate of change. ***As the array approaches convergence the over-relaxation factor is automatically reduced to help smooth the final voltage estimates.***

Planar 2D Symmetry Refining Equations

The voltage change estimates for ***planar 2D non-mirroring*** potential arrays are calculated as follows:

Interior points	P_0	=	$(P_1+P_2+P_3+P_4)/4$
All corner points	P_0	=	$(P_2+P_4)/2$ {lower left corner}
All edge points	P_0	=	$(P_1+P_2+P_3)/3$ {top edge}

The voltage estimate changes for ***x mirroring*** are as follows:

LL corner point	P_0	=	$(P_2+P_2+P_4)/3$
Left edge points	P_0	=	$(P_2+P_2+P_3+P_4)/4$

The voltage estimate changes for ***y mirroring*** are as follows:

LL corner point	P_0	=	$(P_2+P_4+P_4)/3$
Bottom edge points	P_0	=	$(P_1+P_2+P_4+P_4)/4$

The voltage estimate changes for ***x & y mirroring*** are as follows:

LL corner point	P_0	=	$(P_2+P_4)/2$

Planar 3D Symmetry Refining Equations

Interior points	P_0	=	$(P_1+P_2+P_3+P_4+P_5+P_6)/6$
All corner points	P_0	=	$(P_2+P_4+P_6)/3$ {LLL corner}
All edge plane points	P_0	=	$(P_1+P_2+P_4+P_5+P_6)/5$ {Bottom xz)
All edge line points	P_0	=	$(P_2+P_4+P_5+P_6)/4$ {LL edge line}

The voltage estimate changes for the various x, y, and z mirroring follow from the 2D planar examples above.

Cylindrical 2D Symmetry Refining Equations

Cylindrical 2D symmetry equations reflect the fact that each of the four nearest neighbor points **do not** contribute equally to the best estimate of the central point. SIMION uses an effective area weighting function. In this approach, each of the four neighbor points contribute as a function of the area of view (*considering cylindrical symmetry*) it has with the central point.

Interior points	P_0	=	$(P_1+P_2+P_A+P_A)/4$
Lft/Rht edge pts	P_0	=	$(P_2+P_A+P_A)/3$ {Left edge points}
Interior axis pts	P_0	=	$(P_1+P_2)/6 + 2*(P_4)/3$ {y = 0}
Corner axis pts	P_0	=	$(P_2)/5 + 4*(P_4)/5$ {LL Corner}
Upper corner pts	P_0	=	$(2*y*P_2+(2*y-1)*P_3)/(4*y-1)$ {upr lft}
Top edge points	P0	=	(2*y*(P1+P2+P3) - P3)/(6*y-1)
Where:	PA	=	P3 + F*(P4-P3)
	F	=	(2+(1/y))/4
	y	=	y position of center point in grid units

The voltage estimate changes for x mirroring follow from the 2D planar examples above.

Skipped Point Refining

SIMION 7.0 (*6.0 too*) employs a skipped point refining technique (*developed by the author*) that greatly speeds up refining. With skipped point refining active, most arrays refine in times proportional to their number of points (*as opposed to n squared for normal finite difference techniques*).

Skipped refining makes use of a rather direct approach that becomes very complex in practice. This approach is to initially refine a smaller array by skipping points, estimate the values of intermediate points (*double the array density*), and refine again. The process continues until no points are being skipped (*the final refine*). ***The approach used is powers of two point skipping***.

SIMION begins by looking at the size of the array to be refined. The maximum initial skipping is limited by the smallest array dimension. SIMION wants at least 4 points as a minimum dimension in the starting skip level. This means that ***square*** or ***cubic*** arrays produce the greatest initial skip factors for their size (*and usually faster refines*).

Another benefit of skipped point refining is that SIMION automatically refines the initial and next skip to orders of magnitude lower error than the user specification. This quickly finds and establishes the linear gradient fields within the array (*improving accuracy and greatly speeding refining*). ***Thus there is no need to preset linear gradients (a previous speed up trick) as in pre-6.0 versions of SIMION***.

Unfortunately, the big problem is that of electrode visibility. When you are skipping points you are most likely skipping electrode points here and there. If you blithely ignore this issue, the skipped point refine solutions may be quite wrong. This error will persist until the skip length reduces to the point that the troublesome electrode points are visible. Now the program has to propagate these newly visible effects around the array. ***The invisible electrode point problem typically kills any speed advantage simple minded skipped point refining has***.

SIMION attacks this problem by scanning for and flagging any skipped electrodes at the beginning of each level of skipped point refining. Now SIMION knows which points have one or more invisible electrode points lurking near them and where they are. These special points are refined with a star weighting function that converges properly on linear gradients (*nature's preferred gradient*).

This approach combined with special compensations for irregular shifts in array boundaries as skipping changes makes for a very complex refining system. ***However, it also turns out to be dramatically faster for large arrays than the previous SIMION methods***.

Array Doubling

The **DOUBLE OPTION** works in the following way: Dummy points are inserted between each existing point. This means each initial square area of four points becomes a nine point square with the original four points acting as the corner points of the new square.

The problem is what to do with the new dummy points? Are they to be electrodes or non-electrodes? Points are typed in the following manner:

1. Scan alternate lines (*y=0, y=2, and etc.*) from left to right. The dummy point is converted to an electrode ***if and only if*** the point to the left and right are both electrode points ***of the same value***, or else the dummy point is considered to be a non-electrode.

2. Scan each column (*x=0, x=1, and etc.*) from bottom to top looking at the odd line dummy points (*y=1, y=3, and etc.*). The dummy point is converted to an electrode ***if and only if*** the points below and above are both electrode points ***of the same value***, or else the dummy point is considered to be a non-electrode.

Note: If Interpolated Electrodes Mode is active and any new point has neighboring electrode points of different values the new point will be an electrode of the average value of its neighboring electrode points.

SIMION handles 3D arrays in an equivalent manner.

Note: New non-electrode points are <u>always</u> set to zero. The doubled array must be refined after doubling to be usable.

Fast Voltage Adjustment Method

The fast voltage adjustment method used in SIMION makes use of the additive solution property of the Laplace equation. This property allows separate solutions for each boundary (*or collections of boundary elements*) to be combined in a simple scaling/additive process to obtain the potential at each point in the potential array.

For example, if a potential array has five electrodes (*e.g. 1, 2, 3, 4, and 5 volt points*) we could create and refine five potential arrays (*one for each electrode*). Each array would be the specific solution of the Laplace equation for a particular electrode. In potential array number one, all electrode points ***except*** those associated with electrode number one (e.g. *one volt*) would be set to zero volts. The electrode points representing electrode number one would be set to the desired voltage for electrode number one. When this array was refined it would represent the specific solution for the effects of electrode one with the selected voltage.

The remaining four electrode potential arrays are created in a similar fashion. The composite potential would be computed as follows:

Composite potential:

$$P_n = P_{n1} + P_{n2} + P_{n3} + P_{n4} + P_{n5}$$

Where:

n = **Any common array location**

P_{n1} = **Electrode one's solution array**

SIMION includes an additional potential array, the base array, to support the fast adjustment process. This array initially ***only*** contains the solutions for any electrode points that do not have their own individual potential arrays. **If all electrode points are associated with a particular individual solution potential array then all the points of the base array are initially at zero volts.**

However, any non-fast adjustable electrodes have non-zero potentials (*e.g. 835 v*) SIMION 7.0 will refine an additional array **.PA_** containing only their solutions. This array is then available for fast proportional potential scaling of these points.

Fast adjustment works by using each reference electrode potential file and a scaling factor to change the non-electrode point voltages in the base potential array:

Voltage Adjustment:

$$V_{new} = V_{old} + F * V_{ref}$$

Where:

V_{new}	=	**Adjusted point voltage - base array**
V_{old}	=	**Current point voltage - base array**
V_{ref}	=	**Current point voltage - Elect array**
F	=	$Ve_{new}/Ve_{ref} - Ve_{old}/Ve_{ref}$
Ve_{new}	=	**New electrode voltage**
Ve_{old}	=	**Current electrode voltage**
Ve_{ref}	=	**Reference electrode array voltage**

The voltage adjustment is accomplished by loading portions of a reference electrode array and using the portion's points to adjust the equivalent points in the base potential array. During this process all electrode points of the specific electrode are changed to their new value. This process continues one electrode file at a time until the required adjustments have been made.

Trajectory Algorithms

Ion trajectory calculations are a result of three interdependent computations. First, electrostatic, magnetic, and charge repulsion (*if active*) forces must be calculated based on the current position and velocity of the ion(s). These forces are then used to compute the current ion acceleration and used by numerical integration techniques to predict the position and velocity of the ion at the next time step. Moreover, the time step itself must be continuously adjusted to maximize trajectory accuracy while minimizing the number of integration steps per trajectory.

Instance Selection Rules

Unlike previous versions, SIMION 7.0 (*6.0 too*) can have up to 200 potential array instances in its workspace at one time (*instead of just one*), and it must decide which instance to apply at each trajectory calculation point. This problem is solved by keeping instances in an ordered list. SIMION searches down this list from the last instance on the list (*highest priority*) toward the first (*instance 1 - lowest priority*). SIMION will always use the first electrostatic (*and magnetic if defined*) instance(s) it encounters while search down the instance list that contains ion to compute the forces on the ion.

SIMION's Unit System

SIMION uses the following internal unit system:

Time	=	**microseconds**
Length	=	**mm**
Position	=	**mm**
Velocity	=	**mm/ microsecond**
Acceleration	=	**mm/ microsecond^2**
Elect Gradient	=	**volts/ mm**

Electrostatic Force Computation

Electrostatic forces are initially computed in terms of volts per grid unit. As the ion progresses through the potential array it moves from within one square of grid points into another. SIMION automatically generates a small 16 point array that represents the current four point grid and the 12 grid points around it (*64 points for 3D arrays*) . If the ion happens to be in a boundary grid location SIMION estimates the potential of any grid point that falls outside the current potential array. The values of these grid points are determined by symmetry assumptions and grid point location.

P_4

P_1 P_{ION} P_2 **One <u>half</u> grid unit separations**

P_3

Voltage gradients are calculated by first estimating the potentials at four points (*six points for 3D*) as pictured above. These points are exactly 0.5 grid units from the current ion position.

The potential at each point is normally calculated via linear interpolation using the four grid points bounding the grid square it falls in (*8 point cube for 3D*). However, electrode boundaries present a problem.

If one of the four points (*6 points 3D*) has an electrode boundary between it and the ion position, SIMION adjusts its calculations. In this case the value of the point in question is calculated by assuming a straight line extrapolation using the boundaries of the ion's grid square (*assuming continuation of the current gradients*). ***This correction improves SIMION's ability to model fields near electrodes and ideal grids***.

Electrostatic Forces:

Planar 2D

$F_x = V_1 - V_2$ {sign corrected}
$F_y = V_3 - V_4$ {sign corrected}
$F_z = 0$

Planar 3D

$F_x = V_1 - V_2$ {sign corrected}
$F_y = V_3 - V_4$ {sign corrected}
$F_z = V_5 - V_6$ {sign corrected}

Cylindrical 2D

$F_x = V_1 - V_2$ {sign corrected}
$F_r = V_3 - V_4$ {sign corrected}
$F_y = F_r * y/r$ {sign corrected}
$F_z = F_r * z/r$ {sign corrected}

Where:

V_n = potential (in volts) at P_n
r = sqrt(y*y + z*z)

Electrostatic Acceleration:

Now Transform E Forces into workbench orientation
(Forces initially calculated in PA orientation)

$A_x = F_x /(1.03642722x10^{-2} * M * S)$
$A_y = F_y /(1.03642722x10^{-2} * M * S)$
$A_z = F_z /(1.03642722x10^{-2} * M * S)$

Where:

M = Ion's rest mass (amu) / unit charge
S = Scale in millimeters/grid unit

Electrostatic Forces Outside Instances

When an ion is outside all electrostatic instances SIMION looks both directions along its current trajectory path for the closest electrostatic instance of intersection in both directions. If the current trajectory intersects electrostatic instances in both directions, SIMION will determine the potentials at the points of intersection and estimate the resulting electrostatic acceleration (*if any*) assuming a linear gradient. ***Note: Charge repulsion calculations are not effected by the ion's instance (or non-instance) status. Warning: User Programs cannot change the potentials the ion sees, because the ion is not in any potential array's instance.***

Magnetic Force Computation

Magnetic forces are calculated (*if magnetic instances are defined*) by searching the list of instances to see if the ion is presently within a magnetic instance (*searching last to first in instance list*).

Magnetic Forces:

Planar 2D

B_x = ng * (P_1 - P_2) {sign corrected}
B_y = ng * (P_3 - P_4) {sign corrected}
B_z = 0

Planar 3D

B_x = ng * (P_1 - P_2) {sign corrected}
B_y = ng * (P_3 - P_4) {sign corrected}
B_z = ng * (P_5 - P_6) {sign corrected}

Cylindrical 2D

B_x = ng * (P_1 - P_2) {sign corrected}
B_r = ng * (P_3 - P_4) {sign corrected}
B_y = F_r * y/r {sign corrected}
B_z = F_r * z/r {sign corrected}

Where:
ng = user adj magnetic scale factor
P_n = potential (in Mags) at P_n
r = sqrt(y*y + z*z)

Magnetic Acceleration:

Now Transform B field into workbench orientation
(Forces initially calculated in PA orientation)

A_x = C * (V_yB_z - V_zB_y) / M
A_y = C * (V_zB_x - V_xB_z) / M
A_x = C * (V_xB_y - V_yB_x) / M

Where:
C = $9.648453082 \times 10^{-3}$
V_X = Ion's velocity in workbench axis direction
M = Ion's rest mass (amu) / unit charge

Charge Repulsion Forces

SIMION supports three estimates of charge repulsion: Beam, coulombic, and factor. ***The basic approach is to let a few ions represent many***. In general, charge repulsion estimations involve determining the forces between the current ion in question and all other currently flying ions. These forces are then scaled by a charge scaling factor and accelerations are obtained due to each interaction. The sum of the accelerations represent the estimate of charge repulsion effects on the current ion.

Beam Repulsion

Beam repulsion is estimated via line charge assumptions. Each ion is assumed to represent a line charge. The line charge density coulombs/mm is determined by apportioning the beam current between the ions according to their charge adjusted by their charge weighting factor and dividing it by the ion's velocity:

coulombs /mm = (coulombs/usec) / (mm / usec)

Beam repulsion requires that all ions be flown via ***space coherent integration***. This is accomplished by using ion number ***one*** as leader of the pack. At each time step for ion number one a plane containing the ion and normal to its velocity is computed. This plane is then used to control the time steps of all other ions so that they will fall within the plane too. The following calculates the charge repulsion between two representative weighted ions.

Beam Repulsion Acceleration:

re	=	r/r_{avgmin}
rfactor	=	$re^2 / ((0.341995 + re^3) * r_{avgmin})$
scale	=	$-1.73433341x10^{+9}$ * Amps * chg /(m * total_chg)
accel	=	scale * tchg * tcwf * rfactor / tv

Acceleration components computed using unit vector for r between the two ions

Where:		
amps	=	total beam amps
chg	=	charge on current ion (real charge)
m	=	rest mass of current ion (amu)
total_charge	=	sum of all ion's cwf * their absolute charge
tchg	=	test ion's charge (real charge)
tcwf	=	test ion's charge weighting factor
r	=	distance between the two ions
r_{avgmin}	=	current average min distance between ions
tv	=	speed of the test ion

Coulombic Repulsion

Coulombic repulsion is estimated via point charge assumptions. Each ion is assumed to represent a point charge. The total coulombs of charge is apportioned between the ions according to their charge adjusted by their charge weighting factor. Coulombic repulsion requires that all ions be flown in ***time coherent integration***.

Coulombic Repulsion Acceleration:

re	=	r/r_{avgmin}
rfactor	=	$re^2 / ((1.0 + re^4) * r_{avgmin})$
scale	=	$-8.67166704x10^{+14}$ * C * chg /(m * total_chg)
accel	=	scale * tchg * tcwf * rfactor

Acceleration components computed using unit vector for r between the two ions

Where:		
C	=	total coulombs of charge
chg	=	charge on current ion (real charge)
m	=	rest mass of current ion (amu)
total_chg	=	sum of all ion's cwf * their absolute charge
tchg	=	test ion's charge (real charge)
tcwf	=	test ion's charge weighting factor
r	=	distance between the two ions
r_{avgmin}	=	current average min distance between ions

Factor Repulsion

Factor repulsion is estimated via point charge assumptions. Each ion is assumed to represent a point charge. Each point's effective charge is determined by multiplying its charge by the charge multiplication factor adjusted by its charge weighting factor. Factor repulsion requires that all ions be flown in ***time coherent integration***.

Factor Repulsion Acceleration:

re	=	r/r_{avgmin}
rfactor	=	$re^2 / ((1.0 + re^4) * r_{avgmin})$
scale	=	$-1.389354835 \times 10^{-4} * F * chg / (m * avg\_cwf)$
accel	=	scale * tchg * tcwf * rfactor

Acceleration components computed using unit vector for r between the two ions

Where:		
F	=	charge multiplication factor
chg	=	charge on current ion (real charge)
m	=	rest mass of current ion (amu)
avg_cwf	=	average all ion's charge weighting factors
tchg	=	test ion's charge (real charge)
tcwf	=	test ion's charge weighting factor
r	=	distance between the two ions
r_{avgmin}	=	current average min distance between ions

Corrections for Ion Clouds

In each of the three repulsion types above, each ion is actually representing a cloud of ions. If for some reason an ion found itself in the middle of the cloud of another ion it should have no forces on it. However, if the standard $1/r^2$ (*Factor and Coulombic repulsion*) or 1/r (*Beam repulsion*) distances were to apply then forces would be infinite and everything would blow up.

SIMION compensates for this problem by using radius factors (*rfactor above*) that are $1/r^2$ or 1/r if r is large and diminish to zero as r approaches zero. The method used closely models the effect of having

the ion near a cloud of ions (*spherical - Coulombic and Factor, or cylindrical - Beam*) with an effective radius of r_{minavg}.

The value of r_{minavg} is set to the average minimum distance between all currently flying ions. SIMION updates this value each time step. Thus the effective radius of the ion clouds change as the ions move about. ***The single exception is when Factor repulsion is set to 1.0.*** Then SIMION ***always*** uses 3.0e-11 mm (*10 times the classical electron radius*) for the effective cloud diameter.

Relativistic Corrections

The forces due to electrostatic fields, magnetic fields, and any active charge repulsion are combined to obtain the x, y, and z acceleration components assuming the ion's ***rest*** mass. At this point if the ratio of the ion's v^2 (*speed squared*) divided by c^2 (*velocity of light squared*) is greater than 10^{-10}, the following relativistic corrections are applied:

Relativity Correction Factor:

$$rcf = \text{sqrt}(1 - v^2/c^2)$$

Convert each acceleration component (a_x, a_y, and a_z) to its tangential and normal components relative to the ion's current velocity vector (e.g. a_x into a_{xt} and a_{xn}) and apply the following relativistic acceleration correction:

$$a_x = rcf^3 * a_{xt} + rcf * a_{xn}$$
$$a_y = rcf^3 * a_{yt} + rcf * a_{yn}$$
$$a_z = rcf^3 * a_{zt} + rcf * a_{zn}$$

Numerical Integration Method

A standard fourth order Runge-Kutta method is used for numerical integration of the ion's trajectory in three dimensions. This approach has the advantage of good accuracy and the ability to use continuously adjustable time steps. This, to some extent, compensates for the added time required to calculate accelerations at four points for each time step.

Adjustable Time Steps

A good self-adjusting time step method is just as important as the interpolation and integration methods. Each depends on the other if reasonably accurate trajectories are to be calculated in a minimum of computer time.

Large time steps work well in regions of the potential array where voltage gradients are highly linear and have a minor impact the ion's rate of change in kinetic energy. **However in other areas it is very critical to have a small time step to maintain accuracy in high gradient areas. If the time step is significantly too long an ion can overshoot its energy null point (*e.g. reflector fields*) and gain enough false kinetic energy to gradually make the subsequent trajectory meaningless.**

SIMION utilizes a two step process to estimate an appropriate time step for each integration step. Before entering the first step of Runge-Kutta integration, the total velocity and acceleration terms are calculated at the current ion location.

The user has previously entered a value for the largest distance step permitted. **The default (*and maximum*) value for this distance step is one integration step per grid unit of length.**

This distance step is used with the current ion velocity to estimate the time step assuming zero acceleration. Likewise the distance step is used with the current ion acceleration to estimate the time step assuming zero initial velocity. The composite estimate is obtained by using a parallel effect assumption (*to avoid iterative solution techniques*).

Longest time step computation:

$$
\begin{aligned}
t_v &= d / v \\
t_a &= \mathrm{sqrt}(2 * d / a) \\
t_{step} &= t_v * t_a / (t_v + t_a) \quad \{a!=0,\ v!=0\} \\
t_{step} &= t_v \quad \{a = 0\} \\
t_{step} &= t_a \quad \{v = 0\}
\end{aligned}
$$

Where:

d	=	**Maximum distance step**
v	=	**current ion velocity**
a	=	**current ion acceleration**

Notice that t_{step} is calculated three ways depending on the presence of velocity and acceleration.

If both acceleration and velocity are present a second computation is performed to see if the time step should be shortened. A quantity called stop length is computed. Although it seldom indicates the actual ion's stopping distance (*e.g. magnetic accelerations*) it is a very good indicator of the rate of trajectory curvature. The stop length if shorter than 10 grid units is used to compute a reduced time step.

Reduced time step:

$$
\begin{aligned}
S_{dist} &= v * v / (2 * a) \\
\text{If } S_{dist} > 10 \quad & t_{step} = t_{step} \\
\text{If } 1 < S_{dist} < 10 \quad & t_{step} = t_{step} * S_{dist} / 10 \\
\text{If } S_{dist} < 1 \quad & t_{step} = t_{step} / 10
\end{aligned}
$$

Thus the normal time step can be as much as 10 times shorter if the ion is in a region of high trajectory curvature. This method appears to be quite successful in maintaining accuracy while minimizing total integration steps.

Edge Detection and Boundary Approach

Two other problems must also be considered. First, SIMION's trajectory algorithms are quite blind. They can serenely propel an ion right over a sharp gradient edge (*electrostatic or magnetic*) and detect it only after the fact. This can create all sorts of problems with energy conservation and thus compromise the accuracy of ion trajectories. Second, once a boundary (*of whatever type*) has been detected (*by whatever method*) how do you efficiently approach it (*being basically blind*).

Binary Boundary Approaches

SIMION 7.0 makes use of a binary approach method. SIMION looks ahead one time step to see if a boundary has been crossed. If so, the integration time step is cut in half and the integration calculation step is repeated. This process continues until the boundary is no longer crossed. At the next integration calculation SIMION remembers that a boundary exists and starts off with the last successful time step and halves it as necessary to avoid the boundary. Of course this cannot continue indefinitely because the boundary would never be crossed. SIMION limits the smallest time step to 0.0001 of the stop length corrected time step. When this limit is reached the boundary is crossed whatever the eventuality.

SIMION's boundary detection involves the tentative calculation of the next ion position. If any of the four Runge-Kutta terms blow up (*hits an electrode or is outside of grid*) a warning flag is set and if the time step is above the lower limit the time step is halved and the process loops.

Field Curvature Detection

The next boundary test is the gradient of acceleration. The coefficient of variation squared is calculated for the four Runge-Kutta Ax, Ay, and Az acceleration terms. If the C_v for any acceleration term is above 1/(accuracy level) (*typically 1/2 or 0.5*) the binary approach is used until all values are less than the upper limit or the minimum time step size has been reached.

Likewise, when the C_v is less than 1/2 its upper limit, SIMION doubles the time step a step at-a-time until the stop length time is reached or a higher C_v forces another time step reduction. This reduces errors on sharp edges (*grids and magnet boundaries*) and in higher field curvature areas.

SIMION has additional looping triggers for electrodes, grid boundaries, velocity reversals, and other events. Together these algorithms provide SIMION with accuracy improvements while minimizing the number of integration steps.

How Trajectory Quality is Controlled

SIMION uses a **Trajectory Computational Quality** panel (*Normal Controls Screen - accessed via the* **Normal** *tab*) to control how trajectories are calculated. The default value for this parameter is 3.

Quality < Zero

Provides control of relative time steps. The step distance is set to 1/(1 + abs(quality)) of the normal stop length time step. C_v, boundary, and reversal checking are turned off.

Quality set to Zero

This provides the fastest calculations at the expense of accuracy. Time steps are adjusted to move the ion the normal stop length time step (*typically one grid unit*). C_v, boundary, and reversal checking are turned off.

Quality > Zero and < 100

Time steps are adjusted to move the ion the normal stop length time step (*typically one grid unit*). ***CV, boundary, and reversal checking are turned on.*** C_v limits are based on 1.0/quality.

Quality > 100

Provides control of relative time steps too. The step distance is set to 1/(1 + abs(quality -100)) of the normal stop length time step. ***C_v, boundary, and reversal checking are turned on.*** C_v limits are based on 1.0/quality.

Pseudo-Random Number Generation

Pseudo-random numbers are made available to user programs (*Appendix I*) via the RAND and SEED user program functions. SIMION 7.0 contains a new pseudo-random number generator, because Dr. Richard Morrison of Monash University, Australia discovered some fine structure striations (*patterns*) in the random numbers when used in combination with certain trigonometric functions (*this pseudo-random generator has been in use from versions 4.0 – 6.0*).

The new pseudo-random number generator is one the author had devised for other uses. It is a pseudo-random walk generator with a repeat length of greater than 256! Real numbers are created by direct substitution of random bytes from the core generator into a preset IEEE double 64 bit core to create all mantissa values from 0.0 to 1.0. Unfortunately this generator is somewhat slower, but it tests with the best random generators known and has a longer repeat length.

The random number generator makes use of a randomized 65k byte table which is automatically created each time the SIMION initializes. Each created randomized table produces an entirely different sequence of pseudo-random numbers of almost infinite repeat length. A copy of this random table is saved as:

C:\FILES.GUI\INITIAL.TBL

If you want SIMION to reuse a random table definition instead of automatically recreating a different one each time the program starts copy the **initial.tbl** file to the following file:

C:\FILES.GUI\ACTIVE.TBL

SIMION will now always load this file as its **initial.tbl** at startup. To stop this feature ***delete*** the **active.tbl** file, and SIMION will automatically create a new table at each program startup.

The SEED user program function now works in the following manner: A value of 0.0 restores the generator to the initial random table values created or loaded at program startup. Any other number will be used to create a deterministically re-randomized ***working*** table starting with the initial random table settings (*takes some time*). Thus a seed value other than zero will create an entirely new random number sequence (*more than 256! are possible – as opposed to accessible – because of the small 64 bit seed used*).

Note: SIMION periodically makes pseudo-random number generator calls from within several of its GUI service loops (*e.g. Main Menu screen and View function* ***only*** *when the* **Fly'm** *button is* ***NOT*** *depressed*) to further randomize the numbers. To obtain repeatable sequences of random numbers from one fly'm to another, use a SEED function call in an initialize user program segment to force the same starting point (*a seed of 0.0 is recommended - fast*).

THIS PAGE INTENTIONALLY LEFT BLANK