Advanced Strategies and Tactics

Introduction

This chapter provides a few tricks and advanced concepts for getting the most out of SIMION 7.0. SIMION has enough power and flexibility that no one person will probably ever discover all the tricks or use all of its power. The objective of this chapter is to get you well on the road to discovering your own SIMION tricks. SIMION is the tool, ***you*** are the creator. ***You are mostly limited by your imagination***.

Doing More With Array Instances

The following suggestions provide interesting ways to make use of instances:

Instance Scaling Can Cause Instance Skipping

SIMION normally attempts to move an ion one grid unit per time step (*unless the absolute value of the Trajectory Computational Quality is high*). This is normally not a problem. However, if you have a small array instance with an image size that is smaller than one grid unit of the instance it is within, SIMION may accidentally skip over it (*nearsighted as it is*).

You can verify that an array instance is being skipped by using data recording (*Chapter 7*). Create a data record with the ion number (*and location if desired*). Select **Entering an Instance** as the recording event and send verbose output to the Data Monitoring Screen. If the array instance's entering record is missing then the array instance was accidentally skipped during ion flying.

The easiest way to avoid instance skipping is to have the size of the smallest instance a least as big as two or more grid units of the largest instance. If this is not possible, you have the option of turning up the computational quality or making use of user programming tricks (*discussed below*).

Instance Sizing and Positioning Issues

There is a rule with floating point numbers: ***Never depend on one floating point number to exactly equal another for it seldom if ever will!*** This same rule applies to instances. The computer uses base 2 numbers while you use of base 10 numbers. When you position one instance to exactly abut with another instance it is almost certain that they won't. If the two instances don't quite touch and their potential fields are slightly different SIMION will automatically accelerate (*or decelerate*) any ion that just happens to land in the very small region between them (*undesirable*). On the other hand if the instances are made to slightly overlap (*even by as little as a micron*) then all ions will transition between them without gaining or losing KE even if the interface potentials are not exactly equal (*much safer*).

This is particularly important if you use field free instances to jump ions about as described in the user devised trick describe at the end of this chapter. If you make use of jumping instances, make sure you always jump clearly into as opposed to right onto the edge of the destination instance. This insures that no ion will accidentally hit the edge or slightly miss the target instance possibly causing unexpected errors in the trajectory calculations for a few ions. These types of subtle errors have occurred with versions of SIMION built with different compilers and optimization levels. The FPU registers on Intel chips are 80 bits while the double precision word size is 64 bits. If one compiler holds a number's intermediate values in the FPU stack while another normalizes intermediate results at odd times to 64 bits here and there, different versions will not locate instances in ***exactly*** the same place and any modeling that depends on ***exact*** positioning can get into trouble. It is normally not a problem. ***However, the more clever you are the more wary you must be***. Remember: Overlap (*don't abut*) your instances and/or cleanly jump to a location well within an instance (*at least 0.1 micron from the edge*).

Speeding Up Ion Flying Between Array Instances

SIMION 7.0 automatically fast flies ions between array instances unless ions are being flown as dots or some form of charge repulsion is active. When fast flying between instances is ***not*** active, SIMION uses time steps based on the largest scaling factor used for ***any*** array instance. Thus if the largest array instance scale factor defined is 100 mm per grid unit, SIMION will assume 100 mm grid steps when flying outside of array instances.

Using Instances as Portable Beam Stops

It is sometimes useful to be able to insert a beam stop at any location along a beam line. Beam stops help you examine beam shapes, times of arrival, and lots of other things.

The easy way to do this is to create a potential array that is a solid electrode (***no*** *non-electrode points*). A good starting point is a 2D cylindrical array that is 3 grid units thick in x and perhaps 50 units thick in y. Fill the entire array with zero volt electrode points. Save the array with a name like **BEAMSTOP.PA.** You don't need to refine the array because there are no non-electrode points.

Now use the **Add** function within **View** to insert this array into the current workbench. Make sure that the instance has the highest priority (*e.g. highest number in instance list*). Now move, size, and orient the beam stop instance to the desired location within the workbench. ***That's all there is to it.***

Since the presence of one instance doesn't change the fields in another instance, the ions will splat into the beam stop without having their trajectories changed in any way by its presence. You should use a positive value (*e.g. 3*) for trajectory computational quality so that binary boundary approach will be active and the impact points of the ions will be calculated accurately.

This trick works best when you project the beam stop ***within*** other instances. If the beam stop is ***outside*** all other instances you must be sure its potential is the ***same*** as that of the last instance the ions flew through. Otherwise the ions will accelerate or decelerate into the beam stop. ***The way to avoid this problem is to make the beam stop array a magnetic array of zero Mag pole points.***

Modeling Field Interactions Between Instances

SIMION does not compute field interaction effects between instances. Each instance is a island unto itself. It is your responsibility to model these interactions by setting the appropriate array boundary conditions. The following is a collection of methods and tricks you can use. These ***will not*** solve ***all*** your field interaction problems but they should be helpful.

Fields Between Abutted Instances

There are times when the inner portion of a lens component has a simple 2D symmetry, while its ends require 3D asymmetrical modeling. One could use a large 3D array for the entire component. However, the 3D array may turn out to be impossibly large or not model certain aspects of the problem particularly well. In these cases you should consider abutting (*slightly overlapping*) instances together to form the component. An example of this is the quadrupole demo in the **_QUAD** directory.

The problem is how to match the fields properly in regions where instances abut. The most obvious solution would be to break the component at grid boundaries (*have instances abut at grid boundaries*). This assumes the component has plane shaped grids that allow this trick. If it does not have grids, you might well ask yourself why not? Even if there can be no grids, there may be portions with good linear gradients. In this case you could model the problem with grids (*placed well into the linear gradient*) to isolate instances without incurring significant errors when simulating a real-world grid-less system.

Another approach is to recognize the depth of your asymmetries. In a quadrupole, the end regions of the rods are quite asymmetrical. However the central portion of the rods do not see the end regions at all and thus can be modeled by a 2D array. The problem is where to transition between instances. The question is how far do the external fields penetrate. ***You can use SIMION to determine this for your problem.*** However, as a rule of thumb, it is normally safe to assume that external fields do not penetrate a gap to a depth of more than three to five times the gap's width. The quadrupole demo abuts the instances in this region.

Transferring Fields from Outer to Inner Instances

The first question you should ask yourself is why? It is normally good design practice to isolate components as much as possible from each other. This usually involves placing a can around the inner components. When you do this, the problem normally reduces to that of the abutting problem discussed above.

However, there are cases where isolation will not work. SIMION has only limited tools to support these cases. Whether you can make use of them depends on the nature and symmetry of your problem.

If you want to model an emission point at higher resolution within a larger volume, you could create a high resolution array for the point and superimpose it's instance upon the larger volume's instance. The problem is how to set the boundary conditions properly. The suggested approach would be to create the outer volume's array first. Model the emission point within it as best you can. **Refine** the outer volume's array and save it. Now use the **Modify** function to change the points along what will be the interface boundary for the inner instance into electrode points without changing their potentials. This is done by turning **Find** on, setting a large delta find potential, marking an actual boundary line, clicking the **Repl** button, and only changing point type to electrode. Now reduce the size of the array to this bounded portion (*by moving and sizing as required*). **Keep** the changes. Use the **Double** function (*with electrode interpolation active*) to increase the array density one or more times, **Refine** the array, and **Save** it with a new name. This method is discussed in Chapter 5 (**Double** *function*).

The approach will work only in certain cases. One: Both arrays are 2D cylindrical and the point is centered on the x-axis. Two: Both arrays are 3D asymmetrical (*less constraints - more RAM*).

Transferring Fields from Inner to Outer Instances

For example, let's say we have a collection of instances inside a larger 3D potential array instance (*the can*). Let's also assume that the instances interact electrostatically (*e.g. it is not field-free between them*). What we need is a way to model the effects of the interior instances on the fields of the containing 3D potential array instance. We could do this if we could project the appropriate electrode points from the interior instances into their equivalent 3D array instance points. We would then refine the 3D array to estimate the fields outside the interior instances.

The **Cpy** button (*PAs Control Screen in* **View**) provides a method to copy overlapping electrode (*or pole*) points from instances (*of the same type - electrostatic or magnetic*) to the equivalent locations on the ***currently selected 3D (<u>non-mirrored</u>) potential array instance***.

For this problem let's also assume that the 3D array is a **.PA0** fast adjust and that we want to be able to vary the effective voltages of the interior instance electrodes too. The first step would be to create the externally equivalent potential arrays for the internal instances. In most cases (*good design*) cylindrical potential arrays are enclosed by a tube. This means that the array looks like a solid tube to a outside observer. Thus one would use **Modify** to create a solid tube (*bar*) starting with the original array. If the tube is to be fast adjustable it should be given a potential appropriate to mesh properly with the 3D array's **.PA#** file and the file saved with a different name. Now load the desired **.IOB** file and use the **Rpl** button (*PAs Control Screen*) to load the 3D **.PA#** file in its instance as well as the solid cylinder in its instance. Now select the 3D instance, use the **Cpy** button to copy the points from the solid cylinder. Now exit **View** and **Refine** the **.PA#** file. You can now get back into **View** and use **Rpl** to restore the original potential arrays to their proper instances (*be sure to save the* **.IOB** *file afterwards*).

Note: Because the destination 3D array is generally at much lower resolution than the source array you may have problems with ions splating on the crude array (*instance below*) as they exit the higher resolution array above. This problem can be avoided by slightly expanding the size of the interior potential arrays with a boundary area (*volume*) of non-electrode points. This helps ions transition beyond any copied points (*and thus no splat*).

User Programs

If you have played with the demos you have probably seen user programs in action (*e.g. the trap demo*). It is recommended that you read (*or at least scan*) Appendix I before continuing with this section.

Static Variable Issues

Static variables are initialized just before the actual ion flying begins. SIMION 6.0 improperly allowed access to static variables in Initialize and Terminate program segments. SIMION 7.0 properly traps these accesses as errors. See Appendix I for more details.

User Programs and Potential Arrays

User programs are associated with potential arrays. For example the array **TEST.PA**'s user program array would be named **TEST.PRG**. Unless an ion is ***actually within*** the volume of an array instance (*using instance selection rules*) that ***<u>projects</u>*** an array ***with*** user programs, the ion will ***not*** be impacted by user programs.

This means that when an ion is flying outside of all instances, ***no*** user programming control can be exerted on it. This is very important to understand. ***Most of the it-doesn't-seem-to-work-at-all problems relate to ions being out of the span of control of the defined user programs***.

Examples include:

- An initialize program segment will ***<u>never</u>*** see ions that start outside its associated array's projected volume.
- A terminate program segment will ***<u>never</u>*** see ions that die outside its associated array's projected volume.

- Ions flying between array instances that have potentials modified by user programs will not see these effects at all (*while ions are outside the instances*).

Extending the Span of User Program Control

One method to insure that ions can be controlled anywhere within the workbench volume is to create an instance that fills the current workbench volume. This instance ***must be*** instance number ***one*** (*lowest priority*) and should use a low density 3D array (*conserves RAM*). If this array has user programs associated with it, then ions flying outside of higher priority instances will automatically be within its span of control.

Easy Voltage Control With User Programs

When flying ions, it would be nice to be able to fast adjust voltages without having to use the **Fast Adjust** function (*accessible from within* **View**). The ideal would be a voltage on a panel object that could be adjusted directly while ions are flying. User programs will allow us to do just that.

The following user program can be used with **TEST.PA0** file in the **MYSTUFF** directory (*example in Chapter 2*). Create the user program file with an editor and save it in the **MYSTUFF** directory as **TEST.PRG**:

```
DEFA    Lens_Voltage          1000    ;Adjustable variable and value
DEFS    Prior_Voltage         1000    ;Static variable for change detection
SEG     Fast_Adjust                   ;voltage adjustment segment
        RCL Lens_Voltage              ;get current voltage requested
        STO Adj_Elect02               ;set electrode two to desired voltage
        Exit                          ;exit program segment
SEG     Other_Actions                 ;used to update PE surface
        RCL Prior_Voltage             ;get last applied voltage
        RCL Lens_Voltage              ;get current requested voltage
        x=y Exit                      ;exit segment if the same
        STO Pior_Voltage              ;update prior to current
        1 STO Update_PE_Surface       ;request PE surface update
        Exit                          ;exit program segment
```

Use **View** to load the **TEST.IOB**. Fly the Ions **Grouped** in **Rerun** mode. Click the **AdjV** tab, change the voltage on the **Lens_Voltage** variable, and watch the ions fly. Switch to a potential energy view. Notice that the potential energy surface changes each time you change the **Lens_Voltage**. ***This might be a handy user program to modify and use with your simulations.***

Randomizing Ions Via User Programs

It is often desirable to use randomized ions. The **Initialize** program segment can be used to randomize ions. The **_RANDOM** and **_TRAP** demo user programs contain various random ion generators. The basic approach is to randomize the desired parameters of currently defined ions. Thus the randomizing routines will randomize whatever ions you define. Take the time to study each demo randomizer program segment.

The randomizer in the **_RANDOM** demo varies an ion's energy as a random factor of its starting energy and orientation within a random cone angle about its starting angle. The **GROUP.PRG** file in the **_TRAP** demo also randomizes starting location and time-of-birth.

These demos should serve to get you started. Remember, you can apply any probability distribution (*as defined by you*) to the randomization processes.

Data Recording and User Programs

User programs can be used to record data. There are basically two approaches you can use: The **Mark** and the **Message** commands:

The Mark Command

The **Mark** command (*only legal in* **Other_Actions** *program segments*) is considered a legal data recording event. Thus you can define the parameters you want to record and the format to use. Depress the **All Markers** event button. Now whenever an **Other_Actions** segment executes a **Mark** command, a data record will be created (*assuming that the* **Record** *button is depressed*).

The Message Command

The **Message** command (*legal in* **Initialize**, **Other_Actions**, *and* **Terminate** *segments*) can be used to output data records directly. You control the record format by what you include in the message. All numbers will be output using the currently defined data recording number format (*e.g.* **e**, **f**, *or* **g** *plus width and precision*). When **Message** commands are defined in user programs SIMION will ***automatically*** enable the Data Monitoring Screen.

The demos have some good examples of using the **Message** command. The **_BUNCHER** demo computes and displays ion hit time spreads (*in* **TARGET.PRG**). The **_TUNE** demo computes and displays focus quality and informs the user of the tuning process status.

Controlling Time Steps With User Programs

User programs can control time steps. The following examples demonstrate a couple of important uses:

To Approach a Time Boundary

The **_BUNCHER** demo makes use of a time boundary. Before the time boundary, the deceleration voltage is on. After the time boundary, the deceleration voltage is off. The trick is to switch the deceleration voltage at exactly the right time ***and*** have the ion time steps synchronized exactly with this time switch. The **BUNCHER.PRG** has code that accomplishes this. Take the time to see how this works. It may be useful for you too!

Limiting the Maximum Time Step

There are times when automatic time steps can be a problem. In the **_TRAP** demo, when ions are being flown in groups to form crystals, ions slow down and the time step dilates. This can create a stability problem if the time step starts jumping RF cycles. Thus the **TRAP.PRG** has a **TSTEP_ADJUST** segment that limits the maximum time step to 0.1 RF cycle. The protection is important for zero computational quality level trajectories.

Modifying Ion Motions and Accelerations

User programs allow you complete control of ion motions and accelerations. You can even use SIMION to model any arbitrary acceleration problem (*not even ion related*). This capability is quite useful for modeling viscous or collisional cooling.

Viscous Cooling

Appendix I shows listings of how to model viscous cooling. The **_TRAP** (**GROUP.PRG**) and **_DRAG** demos make use of viscous cooling. Study the user programs and try it for yourself.

Collisional Cooling

Two **_TRAP** demos (**INJECT.PRG** and **TICKLE.PRG**) use a simple collisional cooling model. It uses mean-free-path and the mass of the cooling gas as cooling control parameters. All collisions are assumed to be elastic. Further the model assumes cooling collisions don't change direction of ion trajectories (*a somewhat dubious assumption*). The model gives reasonable results and could serve as starting point for your efforts in this area.

Changing the Ions' Colors

The **_TRAP** (**TICKLE.PRG**) and **_BUNCHER** (**BUNCHER.PRG**) demo change the ions' colors as they fly. This is useful if you want to display a change. In the case of the trap demo the color tracks the polarity of the end-cap tickle voltages (*color indicates direction of tickle force - useful*). ***Changing ion colors can be very useful.***

Controlling the Rerun Button With User Programs

The **_TUNE** demo controls the **Rerun** button via the **Rerun_Flym** reserved variable. This allows the user program to keep re-flying the ions until the desired tuning objective has been obtained. It then turns off the **Rerun** button and makes one last flight at the tuned voltage so that the ions trajectories will be recorded. This is an interesting example of user programs used as controllers.

A Clever User Developed Trick

Creative users have developed many tricks with user programs. One of the most impressive tricks developed to date is the ion transporter trick. The problem that led to the trick was how to use realistic grids (*non-ideal – meshes and wires modeled accurately*) to model ions passing through a lens assembly without using all the RAM in the world (*e.g. accurately modeling a gridded reflectron*). Although the lens itself might be modeled with a simple 2D array, the effects of a real grid require a 3D array of high density.

The trick involved modeling the lens with a 2D array and ideal grid (*SIMION's normal grid*). A high density 3D array that modeled a couple of repeating sections of the grid was created (*y and z symmetry to reduce points*). Boundary electrode plates were used to induce the desired driving gradients in the far field. The lens and grid section were located separately in the workspace. Each array had an associated user program file, and each user program file had an other_actions program segment. As an ion approached (*ions flown singly*) the region of the ideal grid in the lens the lens's other_actions program segment would automatically record the location of the ion and jump (*instantly transport*) the ion into the 3D grid's volume at a location that was indexed to the assumption that the grid was mapped as a repeating pattern in the lens itself. The ion's insertion point was saved, and the ion flown through the grid region. As the ion passed through a defined location near the edge of the instance an Other_Actions program segment attached to the 3D grid jumped the ion back into the lens at the proper offset to represent its passage through the grid. The more accurate refractive effects of the 3D grid were introduced into the 2D model. Clever!

This kind of approach can be used to compensate for electrode orientations. Let's assume that we have an instrument where ions enter a region at one angle and exit it at another. If the change of angle is not an integral of 90 degrees it is impossible to align an array's grid with ***both*** the entry ***and*** exit regions. One will have the jags. However, we can make use of a geometry file and create two arrays from different orientations of the geometry file. Each array has a region aligned with the array points. We can now fly the ions into a instance of the first array (*entry region aligned*) and then use a user program (*Other_Actions*) to jump the ion into the second array's instance (*exit region aligned*) in some central region of the array.

Geometry Files

Appendix J gives an ***intense*** discussion of geometry files. These are quite useful if you have complex geometry definitions. Geometry files have ***three*** very real benefits:

1. Array geometry of arbitrary complexity can be defined with geometry files. Geometry files are much more powerful than normal **Modify** methods.
2. You fix an error by editing a file. With normal **Modify** methods you ***might*** have to re-enter all the geometry if you make a really bad error.
3. Once defined, a geometry file can be easily scaled to fit different array sizes. Thus an array can be doubled without introducing the jags.