More Fun With Fields: Simplification by Rebounding

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Back in **GuruGram #39**, we looked at **Fun With Fields** and found out how the **PostScript** computer language greatly simplified the understanding, analysis, and visualization of field problems. This was followed up by **GuruGram #41** on **Array to Image Conversion** that gave us further speedups and simplifications.

We saw that there was a property of fields called the **Laplacian** that determines exactly how your field is going to behave. A **zero Laplacian** is the simplest form and applies to much of electrostatics, electrical currents, temperature gradients, magnetics and noncompressible fluids.

One way to solve a zero Laplacian is to make some guesses and then set your **boundary conditions**. Followed by simply **averaging** each field point value with its four nearest neighbors. After repeating the process many hundreds to many thousands of times, an accurate field image should result.

The only little problem with the Laplacian averaging scheme is that it gets ugly for real world geometries. Especially when rectangular, irregular, and circular boundaries are all involved. I'd like to introduce a possibly new (and definitely heretical) method of Laplacian solution that avoids **all** of the exotic math.

I call this my **rebounding method**.

With rebounding, you **always** do your Laplacian averaging in a simple square or rectangular array. You then **reenter your boundary conditions BEFORE each repeated averaging!**

After a few hundred to a few thousand passes, your field will be a mix of useful and useless solution areas. You then do an image conversion and throw away all of the useless stuff with a proper choice of **clipping paths**. The big advantage is that simple **PostScript** graphic procs can replace extremely complicated and subtle coordinate transformations, complex trig, or otherwise messy math.

Like so…

The REBOUNDING METHOD of zero Laplacian solution:

1. **Do your repeated field averaging over a simple square.**
2. **Re-enter boundary conditions BEFORE each averaging.**
3. **Throw away unwanted areas by image clipping.**

Yeah, that's going to slow us down a little. But you can still get useful results in tens of seconds and precision results in a few minutes. And completely eliminate nearly all of the messy field math in the process.

There are at least two usable methods of reentering your boundary conditions into your field data array. These are the **direct method** and the **infill method**.

The **direct method** is useful for horizontal or vertical boundaries. In which you simply replace any possibly changed array values with their boundary defaults.

The **infill** method is named after the enigmatic and obscure **PostScript infill** operator. You create a **PostScript** graphic proc **the same size as your array**. You then use the **infill** (or possibly the **ineofill**) operator to see which array elements are to be reset to their boundary conditions.

For instance, you could create a fixed boundary circle and then "implant" that circle into your array. And automatically determine which field array points are inside or outside of that circle. **Using ZERO math!**

Now, the inside points of our circle will be dead wrong, but it will be **only the bounding edge** that we are concerned with. We'll later throw away all of the bad stuff with clipping paths on the final image.

The **infill** method is much slower than the direct method, so you should reserve it for times when the field points to be rebound aren't in single rows or columns.

Some Code

Before we look at three real-world rebounding examples, let's pick up some of our needed **PostScript** code. Here is a field array generator…

```
/initfield {/mtval exch store     % save empty value
            /ccc exch store       % save number of columns
            /rrr exch store       % save number of rows
            mark rrr {mark        % start for rrr arrays
            ccc {mtval} repeat    % place ccc data values each
            ] } repeat ]
            /field exch store } store
```

A 40,000 point array field can be set up as 200 row subarrays of 200 column data points each. This seems about right for many uses. **initfield** can be called with the number of rows, the number of columns, and the empty data value on the stack.

We have been using data value reals from 0 to 1000 with 500 being a mid value. They become **blue=0**, **green=500**, and **red=1000** in the final images.

Here's a **quadaverage** routine to replace every **internal** point in a square or rectangular field with the average of its nearest four neighbors…

```
/quadaverage {

 1 1 field length 2 sub {                % for each column
    /hhh exch store

  1 1 field 0 get length 2 sub {         % for each row
    /vvv exch store

    field hhh get vvv 1 add get          % two vertical points
    field hhh get vvv 1 sub get add

    field hhh 1 sub get vvv get          % two horizontal pts
    field hhh 1 add get vvv get add

    add 4 div                            % new average
    field hhh get exch vvv exch put      % and replace

  } for } for } bind def                 % complete loops
```

Note that the field edges do **not** change, so you should only need to set their boundary values **once**. It is only the **internal** values that may need rebounding.

We'll shortly see several examples of direct method rebounding. Here is one possible **infill** rebounder…

```
/resetbound {gsave /boundval exch      % save boundary value
        store newpath exec             % execute 1:1 fill proc
        0 1 field 0 get length 1 sub { % for each element
        0.5 add /curvpos exch store    % center samples
        0 1 field length 1 sub {       % for each column
        0.5 add /curhpos exch store    % center samples

        curvpos curhpos infill         % is rebound needed?
          { field curvpos get          % yes, reset value
           curhpos boundval put
           } if
        } for} for grestore} def
```

Note that the **PostScript** proc must have the same number of horizontal and vertical pixels as there are array-of-arrays elements. Note further that each field value is offset slightly by **0.5, 0.5** to **center** each point on its respective pixel.

Depending on the field complexity, several **resetbound** passes may be needed. The **infill** and **direct** methods can be combined to cover different array points.

These routines can be found ready-to-use in **REBOUND1.PSL**, **BUSONLY.PSL**, and elsewhere on **my website**.

Several Earlier Procs

Before continuing, let's reshow and update some utilities from the two earlier **GuruGrams**. Here's a hue to RGB converter and its supporting procs…

```
/plotsat 0.8 store              % color saturation 1=full
/plotbrt 1.0 store              % color brightness 1=full

/bkg {1 plotsat sub} store
/upset { 1 bkg sub              % available total sat shift
       &cwt mul                 % the desired shift
       bkg add} store           % plus the background
/downset { 1 bkg sub            % available total sat shift
       1 &cwt sub mul           % the desired shift
       bkg add} store           % plus the background

/huetorgb {dup /currenttint % save currenttint
        exch store 5.999
        mul dup floor cvi
        /&cbar exch store       % save case 0-5
        &cbar floor sub         % save posn into case
        /&cwt exch store

 [                              % array of case cases
  { 1 upset bkg}                %   red dominant 0 to .166
  { downset 1 bkg}              %   green dominant .166 to .333
  { bkg 1 upset}                %   green dominant .333 to .500
  { bkg downset 1}              %   blue dominant .500 to .666
  { upset bkg 1}                %   blue dominant .666 to .833
  { 1 bkg downset}              %   red dominant.833 to .999
] &cbar get exec                % exec selected case

255 mul plotbrt mul cvi /curblue exch store
255 mul plotbrt mul cvi /curgreen exch store
255 mul plotbrt mul cvi /curred exch store

curred curgreen curblue} def
```

And here is our disgustingly elegant array-to-string converter…

```
/makestring {dup length string dup /NullEncode filter
3 -1 roll {1 index exch write} forall pop} def
```

And our **field2image** converter that converts a by-columns array of arrays into a series of row strings as requested by an image proc…

```
/field2image { mark              % start row array
   0 1 field length 1 sub        % get row voltage values
   {/fcol exch store
   field fcol get
   arraycount get} for
            ]                    % complete voltage row array

   /arraycount arraycount        % advance pointer
   1 add store

   mark                          % start scaled hue array
   exch {1000 sub abs 1667       % 1000=red 0.000 0=blue 0.667
   div huetorgb  } forall ]      % and complete

   makestring } store            % and convert to string
```

And **fieldasimage** that converts a by-columns array of arrays into an image for compact storage and fast display…

```
/fieldasimage {
  <<                             % begin RGB image dictionary
    /arraycount 0 store          % new row string counter

    /ImageType 1                 % usual image stuff
    /Width imgwide
    /Height imghi
    /BitsPerComponent 8
    /MultipleDataSources false
    /Interpolate false
    /Decode [ 0 1 0 1 0 1]
    /ImageMatrix                 % work UP from bottom
      [imghi 0 0 imghi 0 0]
    /DataSource                  % MUST be deferred!!! proc
      {field2image}
>>
    image } store
```

Variable **/imghi** is the array height as found by **/imgwide {field length} store**. Similarly, variable **imghi** is the array width defined by **/imghi {field 0 get length} store**.

Finally, here is **fieldplot** that positions and draws a field image at a specified page location…

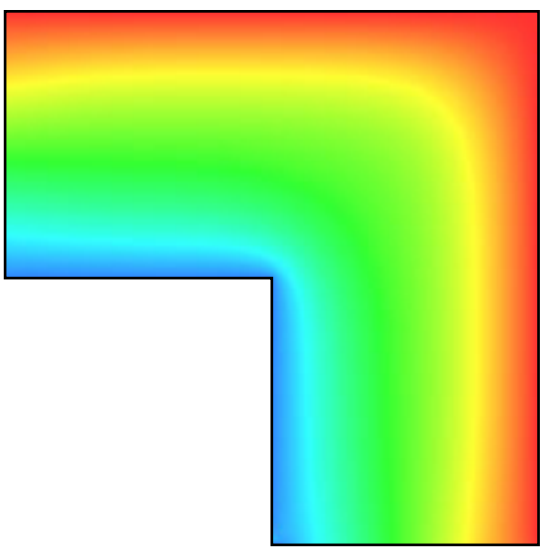
```
/fieldplot {
    /DeviceRGB setcolorspace% set color mode

    gsave translate          % position on page
    imghi dup scale          % scale to size
    fieldasimage             % place image
    grestore} store
```

So what good is all this stuff? Lets look at three real-world examples…

Example #1 — A Printed Circuit Trace Sharp Corner

What kind of trouble can you get into by using pretty square corners on higher power printed circuit traces? Let's start with our answer and work our way back to how we can use rebounding to get there…



We can immediately see that the current density at the inside corner is **three or more times higher than normal!** And that this can cause excessive heating at higher current levels. Thus, all of those pretty square edges can easily cause you bunches of grief. To make your current density more uniform, do a 45 degree break at each corner, or use rounded edges instead.

We start our field plot off with an array of 200 column arrays of 200 data points each. Our outside edges will not change during averaging, so we can enter them just **once** as…

```
/settop {0 1 199 {field exch get 199 1000 put} for} store

/setright {field 199 mark 200 {1000} repeat ] put} store

/setleft {/val 0 store 100 1 199 {field 0 get exch val put
  /val val 10 add store} for

/setbot {/val 0 store 100 1 199 {field exch get 0 val put
  /val val 10 add store} for
```

Since everything is horizontal or vertical, we can use the faster and simpler direct rebounding. The left-of-center row and the bottom-of-center edge boundaries **will** change with averaging and **must** be reentered before **each** averaging…

```
/setwest {0 1 100 {field exch get 100 0 put} for} store

/setsouth {0 1 100 {field 100 get exch 0 put} for} store
```

Here is the code that does the averaging and rebounding…

```
/pccornerfield {200 200 500 % create 200 x 200 green field
  initfield
  settop setright setleft      % set initial boundary conditions
  setbot setwest setsouth

  numrebounds {setwest         % repeat rebounds & averages
  setsouth quadaverage
  field 120 get 120 get ==     % optional progress reporter
          } repeat

  gsave pccornerclip           % clip to useful portion
  closepath gsave eoclip
  gsave 0 0 fieldplot          % plot the useful field
  grestore grestore stroke     % stroke outline
  grestore} store
```

numrebounds determines the number of averages taken. About 2500 gives useful results in this example. You can watch the optional **progress reporter** till nothing changes much. Curiously, in this example, the values first go down and then slightly back up again as the more distant precincts eventually report in.

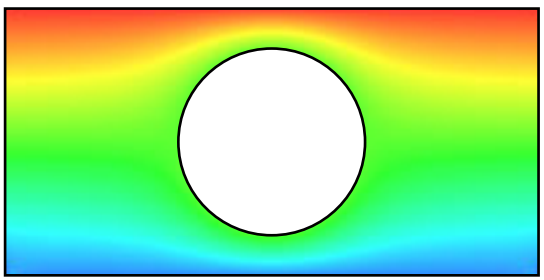
And here is the sneaky clipping that uses my **Gonzo Utilities** to chop out the unneeded (and wrong) southwest corner…

```
/pccornerclip {0 100 mt 100 pu 200 pr 200 pd
                    100 pl 100 pu 100 pl} store
```

Equipotential and gradient lines can easily be added to these plots by using the techniques of **FUNFIELD.PDF**. I've left them off here for clarity.

Example #2 — A Busbar Mounting Hole

Traditional field solutions quickly get messy if you have rectangular and circular areas involved at the same time. Such as this busbar with a mounting hole in it…



Here are the direct boundaries…

```
/setleft {/val 0 store  50 1 150 {field 0 get exch val put
          /val val 10 add store} for} store

/setright {/val 0 store  50 1 150 {field 199 get exch val put
          /val val 10 add store} for} store

/settop {0 1 199 {field exch get 150 1000 put} for } store
/setbot {0 1 199 {field exch get 50 0 put} for } store
```

setleft and **setright** need entered only once, while **settop** and **setbot** will have to be rebounded before each repeat averaging.

Here is our circular **infill** boundary…

```
/cencirc {newpath 100 100 35 0 360 arc} store
```

Here is the code that does the averaging and rebounding…

```
/busbarfield {200 200 500          % create 200 x 200 green field
  initfield
  setleft setright settop          % set initial boundary conditions
  setbot {cencirc}                 % using both direct and infill
  500 resetbound

  numrebounds {settop              % repeat rebounds & averages
  setbot {cencirc} 500
  resetbound quadaverage
  field 100 get 140 get ==         % optional progress reporter
          } repeat

  gsave busclip                    % clip to useful portion
  closepath gsave eoclip
  gsave 0 0 fieldplot              % plot the useful field
  grestore grestore stroke         % stroke outline
  grestore} store
```

Note that **cencirc** is called as **{cencirc} 500 resetbound**. As a **deferred** proc.

Finally, here is our **busbarclip** clipping path that chops off the upper and lower quarters of the square and the center of the hole circle…

```
/busclip {newpath 100 100 35 0 360 arc 0 50 mt 200 pr
          100 pu 200 pl 100 pd} store
```

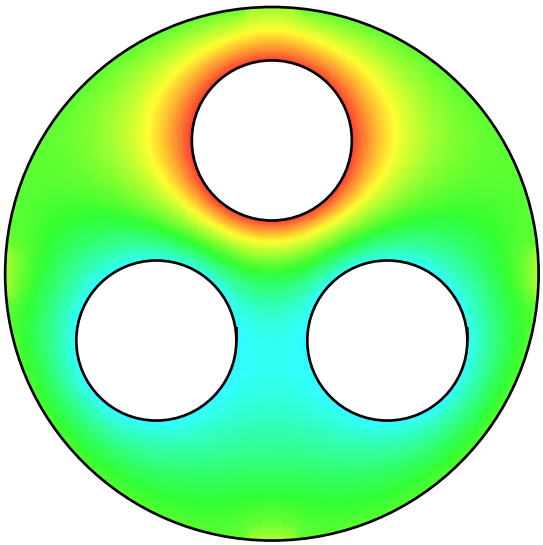
A reminder that we have **completely** characterized the field by the final numbers in our array. Intermediate points are easily and accurately found by interpolation. There is a slight error introduced by the finite number of data points working with differentials rather than true point derivatives. If more precision is needed, the number of data points can be quadrupled and then quadrupled again.

While equipotential lines and gradient lines are easily derived from the **fields** data array, there is often little point in doing so. The whole point of equipotentials and gradients was to **solve** the field problem in the first place. Which we have replaced with repeated rebounding and averaging instead.

Example #3 — Three Phase Power in a Conduit

The beauty of the rebounding method is that your averaging solution takes place over a plain old square. No cylindrical, spherical, or fancier transformations are needed. There are also no hyperbolic trig functions or similar nasties.

This field is really ugly when done the old way in traditional rectangular coordinates…



What we have here is a conduit with a triplet of three phase power lines in it. The top phase is shown at peak positive amplitude, while the bottom two will be at **one-half** their peak negative value. The outside shield is at ground potential.

Four infill boundaries are needed…

```
/threephaseoutside {newpath 100 100 100 0 360 arc
            0 0 mt 200 pu 200 pr 200 pd 200 pl} store

/onehole {newpath 100 150 30 0 360 arc} store

/twohole {newpath 143.3 75 30 0 360 arc} store

/threehole {newpath 143.5 125 30 0 360 arc} store
```

Here is the code that does the averaging and rebounding…

```
/threephasefield {200 200     % create 200 x 200 green field
  500 initfield
  {threephaseoutside}         % set four infill boundaries
  500 resetbound
  {onehole} 1000
  resetbound
  {twohole} 150
  resetbound
  {threehole} 150 resetbound
```

```
numrebounds {
500 resetbound                  % reset four infill boundaries
{onehole} 1000
resetbound
{twohole} 150
resetbound
{threehole} 150
resetbound quadaverage          % and do repeating averages
field 100 get 110 get ==        % optional progress reporter
          } repeat

gsave threephaseclip            % clip to useful portion
closepath gsave eoclip
gsave 0 0 fieldplot             % plot the useful field
grestore grestore stroke        % stroke outline
grestore} store
```

And here is our **threephaseclip** clipping code…

```
/threephaseclip {newpath 100 100 100 0 360 arc 130 150
moveto 100 150 30 0 360 arc 143 30 add 80 moveto 143 75
30 0 360 arc 57 30 add  80 moveto 57 75 30 0 360 arc
} store
```

The equipotentials and gradients for this particular field are quite impressive. I've left them off as an exercise for the student.

It might be very interesting to redo our three phase rebounding every thirty degrees or less. And then make an animated and looped sequence of the results. This can greatly aide the visualization and understanding of three phase power.

For More Help

Consulting services are available per our **Infopack** services and on a contract or an hourly basis. Additional GuruGrams are found **here**, PostScript topics **here**, and math items **here**. Really advanced **PostScript** math problems are found in our **Magic Sinewave** library as well.

Further **GuruGrams** await your ongoing support as a **Synergetics Partner**.