Animating the Evolution of a Field

John B. Schneider, Patrick J. Flynn*, and Kurt L. Shlager†

September 9, 1996

Abstract

Many problems in electromagnetics involve modeling a changing multidimensional field and one can often gain insight into the underlying physical problem by animating the changes. The ability to view directly the evolving field may also provide a useful debugging tool during modeling and simulation. In the past, a great deal of programming effort or the purchase of expensive commercial software was required to produce such animations. Here we present a relatively simple scheme to animate a changing two-dimensional field (which can be a "slice" through a computational domain of higher dimensions). The scheme permits various mappings of field values to colors so that the color of each pixel in an image indicates the field found at the corresponding location within the computational domain. Alternatively, a grayscale mapping can be used. A program used to construct individual frames of the animation is presented in full. Sufficient detail is given so that the customization of the code is straightforward. Public-domain software is used to view the frames as an animated sequence or to generate an MPEG file. Some aspects of the scheme described here are tailored for use in an X-windows or UNIX environment, but most of the important steps are independent of the operating system. Frame generation routines are presented in Fortran (C versions of the code are similar and can be obtained "on-line"). The scheme presented

*J. Schneider and P. Flynn are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164-2752, E-mail: `schneidj@eecs.wsu.edu` and `flynn@eecs.wsu.edu`

†K. Shlager is with the Antenna Systems Department, TRW Electronics Systems and Technology Division, Space and Electronics Group, MS 66/1284-G, Redondo Beach, CA 90278, E-mail: `kshlager@ant35.sp.trw.com`

here is relatively fast, efficient, and flexible and should serve well as a starting point for those wanting to "roll their own" graphics. In addition, for those needing more sophisticated renderings, pointers are given to several powerful commercial and public domain graphics packages.

1 Introduction

Animations permit data to be viewed from a unique perspective and can serve as an important analysis tool in many research projects. In a scheme where field values within a computational domain are mapped to color-coded pixels, one can readily observe the evolution of the field by animating the images obtained from the mapping. An animation might be used to show the propagation of a field as a function of time, or it might be used to show the changes in the distribution of a field as a function of some problem parameter (in a scattering problem this parameter might be the permittivity of the scatterer or a characteristic dimension). In many instances, animations can provide insights into the governing physical mechanisms; such insights may not be yielded from "static" results (e.g., results such as radiation patterns, resonant frequencies, or scattering cross sections). Unfortunately, the development of "in-house" animation capabilities has historically been beyond the reach of smaller research operations. However, the continued decline in the cost of computing and the easy access the Internet has provided to powerful and robust public-domain software allows animations to be easily produced by small laboratories with modest computational resources.

In this paper we present a scheme to obtain animations of an "evolving" field. For the sake of concreteness, the scheme will be presented in the context of the Finite-Difference Time-Domain (FDTD) technique applied to electromagnetic wave propagation. However, the same scheme can be used for many other applications—the underlying premise is that the values in a two-dimensional grid are changing as a function of some parameter and the goal is to observe those changes directly as a function of the parameter. For example, the scheme presented here might be used to make an animation that shows how the absorption of energy within a material changes as the conductivity is varied (and the calculation of the absorption might be done using any one of a number of techniques besides FDTD). Although the displayed values are taken from a two-dimensional grid, this grid can be a slice through a data space of three or more dimensions; hence, the scheme can be meaningfully applied to problems of arbitrary dimension.

To use the scheme presented here, the modifications that must be made to existing code are likely to be minimal since most of the work is done in "post-processing." Our implementation of the scheme relies upon publicly-available software and, in that sense, is free. However, in the sense that the scheme requires the modification of existing code, downloading software from different sites, and sorting through some programming details (all relatively simple), the scheme may be thought of as "low cost." Due to the widespread use of Fortran in the electromagnetics community, code will be presented in Fortran, but translation to C is straightforward. In fact, since C provides a more natural environment for byte-wise operations, the C code is somewhat simpler and significantly faster than the Fortran version. To obtain the C routines, the interested reader is directed to the on-line site mentioned below.

The approach described in this paper, which is depicted in Fig. 1, is one of many ways to construct animations of two-dimensional field data. Many commercial and public-domain software suites (examples include MATLAB, IDL, PV-WAVE, AVS, Mathematica, VTK, Explorer, Khoros) contain the basic procedures needed to build individual animation frames. The elements of the approach described here are:

1. A sampling procedure to subsample (reduce in size) or interpolate (increase in size) the measured field data producing a pseudo-measurement array of the same size as the desired output image.

2. An imaging procedure to map the field quantities in the pseudo-measurement array to a pixel color or shade of gray.

3. An output procedure to write the image data to a file in an appropriate format (e.g., GIF, PPM, TIFF, JPEG).

4. (optional) A procedure to assemble the images produced using the previous steps into a single animation file in an appropriate format (e.g., MPEG). Few visualization packages support this step, but most will provide an animation capability available within the application. Standalone animation files like MPEG files can be constructed by specialized applications such as the public-domain program ***mpeg_encode***. Specialized hardware exists for producing MPEG video files from digitized analog video input.

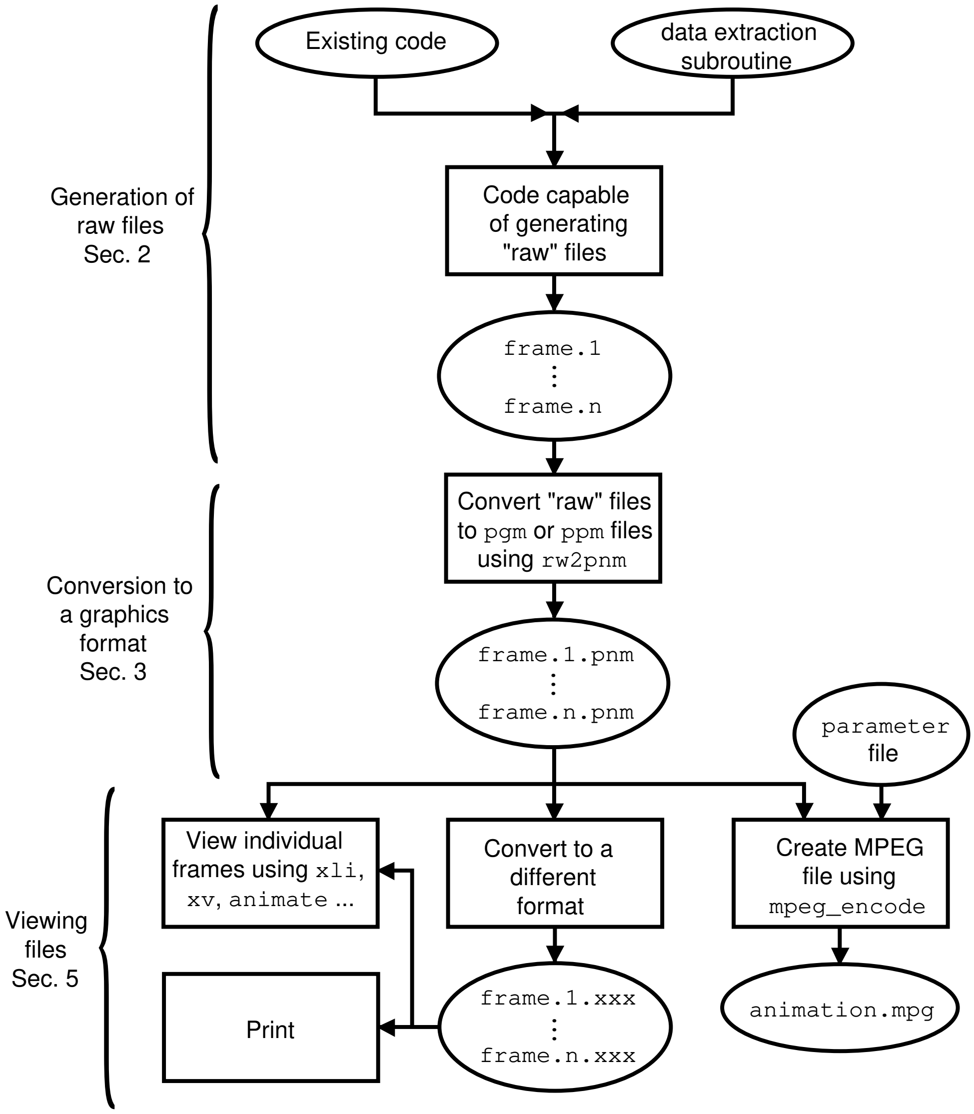

Figure 1: Flow chart describing the creation of animations or still images. Ovals are used to indicate stored data files or program listings; rectangles indicate executable routines.

Depending on the desired audience for animations produced using these techniques, some video production steps might be needed (e.g., titling, recording onto video tape, editing into productions complete with soundtracks, multiple animation segments, dissolves and fades, etc.). These production steps are beyond the scope of this article; specialized publications, such as *Digital Media*, contain articles and advertisements for companies which offer solutions and services for computer-generated video productions.

Because the scheme presented here is so simple, it may lack features that some could consider essential for certain applications. Those interested in obtaining more powerful software are directed to the software listed in Appendix C. Nevertheless, the scheme presented here does have the attraction that, due to its simplicity, it is easy to tailor to different purposes or incorporate directly into existing code. Also, it is relatively fast and efficient and can be used to generate animations "on the fly."

The paper is organized as follows: Section 2 describes the the assumed format for the "raw" frame (or data) files. The necessary modification of existing code to generate the raw data is discussed. Section 3 describes the conversion of the raw frame files into an acceptable graphics file format. Section 4 presents some examples of typical output. The final section discusses ways in which the frames may be viewed (either individually or as an animated sequence). The appendices provide code listings and Internet sites where relevant software can be obtained.

To save the interested reader from typing code, complete code is available on-line via the URL

http://www.eecs.wsu.edu/~schneidj/Animations/

This Web page provides example output as both still shots and MPEG animations. Additionally, the FDTD code that generated the examples is presented. This URL also provides links to all the relevant Internet sites mentioned in this paper.

2 Writing "Raw" Frame Files

The first step is the generation of the raw data that are the source of each frame of the animation. This involves combining a "data extraction" subroutine with existing code so that the resulting code produces the necessary data files. This is depicted in the top third of the flow chart of Fig. 1. The raw data is written to output files so that the actual rendering of an image can be done either in a separate post-processing stage or by a separate process that runs concurrently with the data-generating code. In this way, the amount of effort needed to "instrument" existing code is minimal—only the data extraction subroutine need be spliced into existing code. We have found it most convenient to place the raw data for each frame in separate output files, i.e., one output file per frame. The raw data are stored directly in binary format (instead of formatted ASCII values) to save space—each single-precision field value is stored using only four bytes. These raw files have a common "base name" in addition to a postfixed unique numeric designator that is set off from the base by a period. For example, a collection of one hundred raw data output files with the base name ***frame*** would consist of the files ***frame.1***, ***frame.2***, . . . ***frame.100***.

Although the raw data files must be consecutively numbered, one does not necessarily want to generate an output file at each distinct value of the "driving parameter" of the process. For example, in some instances one would not want to generate an output file at each time step in an FDTD simulation since there may be too little change from one frame to the next. A more reasonable approach might be to generate a frame after every two (or more) time steps. The offset between times at which output is produced is something that is best left to the individual user. Furthermore, it may also be desirable to subsample the spatial grid. For example, it may be best to only display the field associated with every second grid point if the grid is very dense. On the other hand, for grids that are very sparse, to avoid excessive graininess in the output image, it may be desirable to interpolate additional output (or image) points.

Since the details of any data extraction subroutine are so closely linked to the particular application, we will merely describe the minimal set of tasks it should fulfill. The data extraction subroutine should:

1. Open an output file for each frame of data. The file names have a unique "base" name and a postfixed frame number. Although not absolutely necessary, for the sake of efficiency, the output files should be opened for binary output (as opposed to formatted ASCII data).

2. Write the number of columns and rows of output (to be consistent with other data, these data are written as single precision real numbers).

3. Write the data that corresponds to each pixel of the image. Data is written as binary single precision real numbers. The order is such that the first point of data will correspond to the pixel in the upper-left corner of the image, the next point will appear to the right of that, and so on, until the first row is complete. The next data point corresponds to the first pixel in the second row, etc., proceeding in the direction in which one reads English text (left to right, top to bottom).

4. Finally, in many instances it is useful to have the data extraction routine record (and later report) the largest output value. Although this value is not strictly required, it can be used to normalize the images so that the colors in each image are rendered with respect to the maximum value found in the entire collection of images. Subsequently, this value will be referred to as $x_{\max}$.

This set of tasks is fairly straightforward and should be easy to implement for most programmers. However, two aspects of this list that may be somewhat challenging, especially to Fortran programmers, are the handling of binary data and the creation of a sequentially numbered set of output files with a common base name. Those who would benefit from an example data extraction routine are directed to Appendix A where a routine is given that, in addition to accomplishing the above tasks, permits the subsampling of any rectangular subsection of a two-dimensional grid using arbitrary offsets (i.e., arbitrary strides).

3 Conversion of Raw Data Files to a Standard Graphics Format

After the raw output files have been constructed, they must be processed into suitable graphics files. This may be done either concurrently while the data is being generated or in a post-processing step. This stage is depicted in the middle of the flow chart of Fig. 1. Here we will discuss conversion only in the context of post-processing. (A discussion of concurrent processing can be found at the Web site.)

The graphics formats used here are part of the suite of "portable anymap" (or PNM) formats that describe raster images. We will be using either grayscale mappings or color mappings. The grayscale output will be designated PGM output (corresponding to "portable graymap") while color output will be designated PPM output (corresponding to "portable pixmap"). Files with these formats can be manipulated in a number of ways using a freely available collection of routines, generally referred to as the ***pbmplus*** routines (Appendix C lists sites where these routines can be obtained). Furthermore, these files can easily be converted to a format suitable for printing (such as PostScript) by either using one of the filters supplied in the ***pbmplus*** distribution or by using the ***ghostscript*** utility found on many systems.

A PNM file starts with a two character "magic number" that specifies the format of the data comprising the image. For example, the magic number used for color images is "*P6*" indicating byte (binary) storage of pixel values (*P3* would be used if pixel values were represented in ASCII). The magic number is followed by the width, the height, and the maximum gray or color value (all this "header" information is written in ASCII decimal even when the subsequent data is binary). The width and height are the number of horizontal and vertical pixels in the image, respectively. The maximum gray or color value can be used to expand or contract the dynamic range of an image. However, this is not important for the task at hand, and hence this value will be fixed at 255. Following this header information is either the grayscale or RGB (red, green, blue) information for each pixel in the image. Thus, for a color image three bytes, representing red, green, and blue, each between 0 and the specified maximum value (255), are given "width $\times$ height" times. The first three bytes specify the color of the pixel in the top-left corner of the image and from there proceed left to right in normal English reading order. If all three bytes are 0, the pixel will be black; if all are 255, the pixel will be white. Different colors are obtained by varying the relative values of the R, G, and B bytes.

The production of grayscale images is relatively straightforward while the steps involved in creating a color image are slightly more involved. Hence, we will focus the discussion on the generation of color output. As the raw files are translated into PPM files, it is necessary to map field values into RGB values and there are several ways this can be done. It is quite common to quantize the fields so that all values that are between certain limits map to a specified color. Here, instead, we will present an approach where the mapping of fields to color is virtually continuous, i.e., the color changes continuously with continuous changes in field. Unfortunately, it is rather difficult to translate from a field value directly to a desired color using RGB values. In contrast, it is relatively simple to specify desired colors as points in a color space characterized by hue, saturation, and brightness. Assuming for the moment

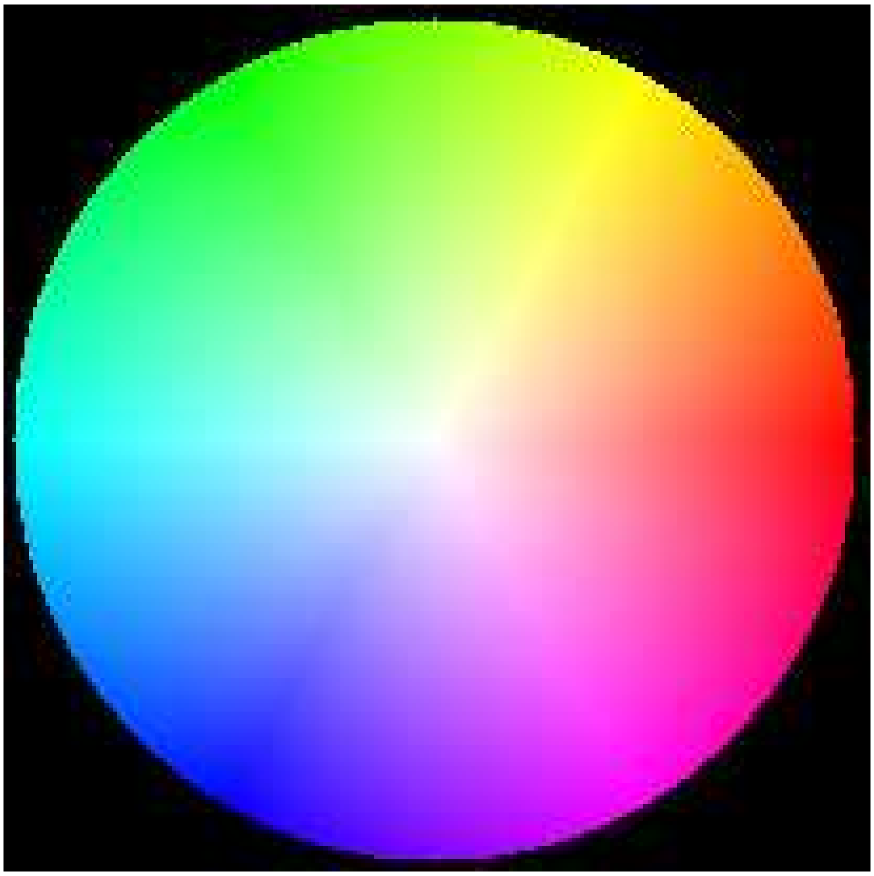

Figure 2: The hue-saturation color wheel (brightness set to unity).

that the brightness is set to unity, we can define a hue-saturation "color wheel" as shown in Fig. 2. The center of this wheel corresponds to equal parts of red, green, and blue and is thus white. The distance from the center is the saturation s, while the angle from the center to a given point is the hue h. The maximum saturation is unity and the hue is assumed to be between $0°$ and $360°$. Pure red is $(h = 0°, s = 1)$, green corresponds to $(h = 120°, s = 1)$, blue is $(h = 240°, s = 1)$, and white is (h =anything, $s = 0$). Besides hue and saturation, there is a brightness parameter, usually referred to as the *value* or v, that varies between zero and one. If the brightness is zero, the pixel is black regardless of the hue or saturation. If the brightness is a small value, the pixel appears dark. For example, if the saturation is zero (i.e., equal parts of red, green, and blue) but the brightness is low, the pixel will appear gray rather than white.

In the conversion of the raw data files to PPM files, a path within the HSV color space is defined. (The HSV space is defined as a hexcone, or six-sided pyramid, which can be

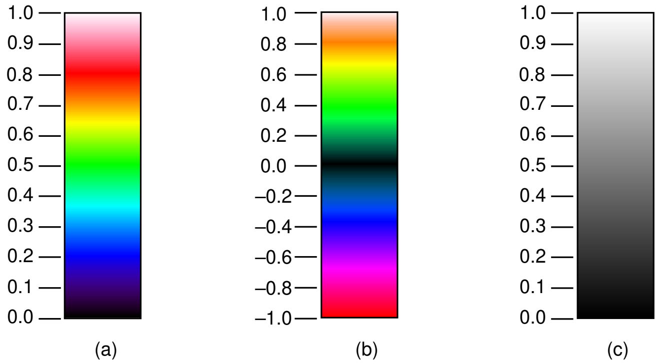

Figure 3: Mapping of fields to pixels. (a) One-sided color. (b) Two-sided color. (c) Grayscale.

described naturally in cylindrical coordinates. However, for the sake of simple visualization, it may be useful to think of the HSV color space in terms of a (ρ, ϕ, z) cylindrical coordinate system where s corresponds to ρ $(0 \leq \rho \leq 1)$, h to ϕ $(0^\circ \leq \phi \leq 360^\circ)$, and v to z $(0 \leq z \leq 1)$. A complete discussion of color spaces is beyond the scope of this article and the interested reader is directed elsewhere for a more detailed discussion of the HSV space. For example, see *Computer Graphics: Principles and Practice*, (2nd ed.), by J. Foley, A. van Dam, S. Feiner, and J. Hughes, Addison-Wesley, 1990.) Each field value is mapped onto this path through the HSV space, thereby identifying the HSV coordinates of the pixel. Finally, the coordinates in HSV space are converted to RGB (using a standard conversion algorithm) and written to the PPM file. Because it is important in some applications to distinguish between positive and negative values of the field while in others this distinction is unimportant, we will define two different color mappings (more can be constructed if desired). One mapping, shown in Fig. 3(a), is used for "one-sided" output, where it is assumed that the normalized absolute value of the field is between zero and unity. The other color mapping, shown in Fig. 3(b), is used for "two-sided" output, where it is assumed the normalized field is between -1 and 1. Grayscale mapping, shown in Fig. 3(c), is inherently "one-sided" so that that large fields, whether negative or positive, tend to white while small fields tend to black. Please note that once one has a basic understanding of the HSV color space it is simple to construct other color mappings. Therefore, one can easily design color maps that are tailored to a given application. For example, in our modeling of acoustic propagation in water, we chose to make zero pressure correspond to blue. This mapping of blue to "unperturbed water" provided a visual context that made the subsequent animations much easier to decipher for those viewing them for the first time then they might have been otherwise.

The main routine that converts the raw files to PGM or PPM files is ***rw2pnm*** which is presented in Appendix B. This routine calls the subroutine ***cmap1*** if the one-sided color map is desired, or ***cmap2*** if the two-sided color map is desired. The subroutine ***hsvrgb*** is used to convert HSV values to RGB values. (The subroutine listings are also given in Appendix B.) As discussed in more detail below, ***rw2pnm*** prompts the user for the base name of the raw files, a normalization value, a scaling parameter, and the desired mapping (i.e., grayscale, one-sided color, or two-sided color). After these are entered, the program uses all of the raw files in sequence to generate the output PNM files. Grayscale files have "***.pgm***" appended to the original file name while color output files are postfixed with "***.ppm***." (The ***rw2pnm*** program was written in such a way that it can be invoked more than once while working on the same set of files. This permits parallel processing and may be useful when more than one processor can access the disk where the files are stored.)

As the program ***rw2pnm*** generates its output, it normalizes each field value in each frame by the same amount. Typically, the user-specified normalization would be the $x_{\max}$ value that was found during the creation of the raw files (i.e., the largest value written by the data extraction subroutine). By using this value, the data is guaranteed to be within the limits assumed by the color mappings. If, for some reason, a normalized value falls outside of these limits, the value is mapped to the color corresponding to the appropriate limiting color and hence it is not imperative that the normalization actually be greater than the maximum absolute value of the field. In this way, if the actual $x_{\max}$ value is not know for a particular set of files, one need not sift through the entire set seeking the true maximum value provided that a reasonable guess can be made for a suitable value.

In many simulations the fields vary over a large range, and the mappings shown in Fig. 3 do not provide sufficient dynamic range. For example, in a small portion of the computational domain the fields may be large, but over the vast majority of space the fields may be orders of magnitude smaller. Or, in an animation, initially the fields may be be large, but as the animation progresses, the fields may become small (either through absorption or radiation). These "small" values, all close to zero relative to the largest fields, would appear virtually identical in terms of color. To provide greater dynamic range, the ***rw2pnm*** programs allows the user to specify a scaling parameter N_d. If N_d is zero, ordinary linear scaling is used.

Assuming the user entered the normalization value $x_{\max}$, the scaling parameter N_d is used to map logarithmically from the range of field values $x_{\max} \times 10^{-N_d} \leq x \leq x_{\max}$ to the range $0 \leq x \leq 1$. If the field is negative and two-sided output is required, the sign is preserved, i.e., the range $-x_{\max} \leq x \leq -x_{\max} \times 10^{-N_d}$ maps to $-1 \leq x \leq 0$. Using this scaling, all fields with an absolute value less than $x_{\max} \times 10^{-N_d}$ are treated as zero. This type of scaling can be used to emphasize the variation in small field values. For example, if the user specifies an N_d of 4, the first decade of values, those between $x_{\max}/10$ and $x_{\max}$, would be assigned colors corresponding to those between 0.75 and 1.00 on a linear scale. The second decade of values, those between $x_{\max}/100$ and $x_{\max}/10$, would be assigned colors corresponding to those between 0.50 and 0.75 on a linear scale, and so on. The same type of scaling is available if grayscale output is desired. The output from ***rw2pnm*** shows the grayscale or color map used, under the selected scaling, in a ten-pixel wide column along the left side of each frame.

The following section provides some examples using these mappings and the selectable scaling. In addition, the ***Animations*** Web site mentioned in Section 1 provides example MPEG and "still" output that was generated using each of the three mappings. The examples are based upon FDTD code that is also available from the Web site.

4 Example Output

To illustrate the type of output that is possible from this scheme, we will apply it to results obtained from an FDTD solution to a simple scattering problem. In this example, a planar TE pulse (the non-zero fields are H_z, E_x, and E_y) illuminates a rectangular perfect electric conductor (PEC). The pulse is a Ricker wavelet and is discretized so that there are 48 points per wavelength at the peak of its spectral energy. The "peak" frequency, or the frequency with peak spectral energy, has a corresponding wavelength λ_p. As shown in Fig. 4, the rectangular scatterer is $\lambda_p \times 3\lambda_p$ and has five slots cut in it which are separated (center-to-center) by $\lambda_p/2$. Each slot is $\lambda_p/6$ high and has a length of either $\lambda_p/3$ or $\lambda_p/2$.

To demonstrate the utility of animations, one merely has to view simulations of this scattering problem both with and without dielectric loading of the slots. Such animations can clearly show the effect that slot resonances, or the lack thereof, have on the way in which the fields are scattered. Since animations cannot be presented here and the display of a long sequence of still shots would be wasteful, results will be presented only for the case where

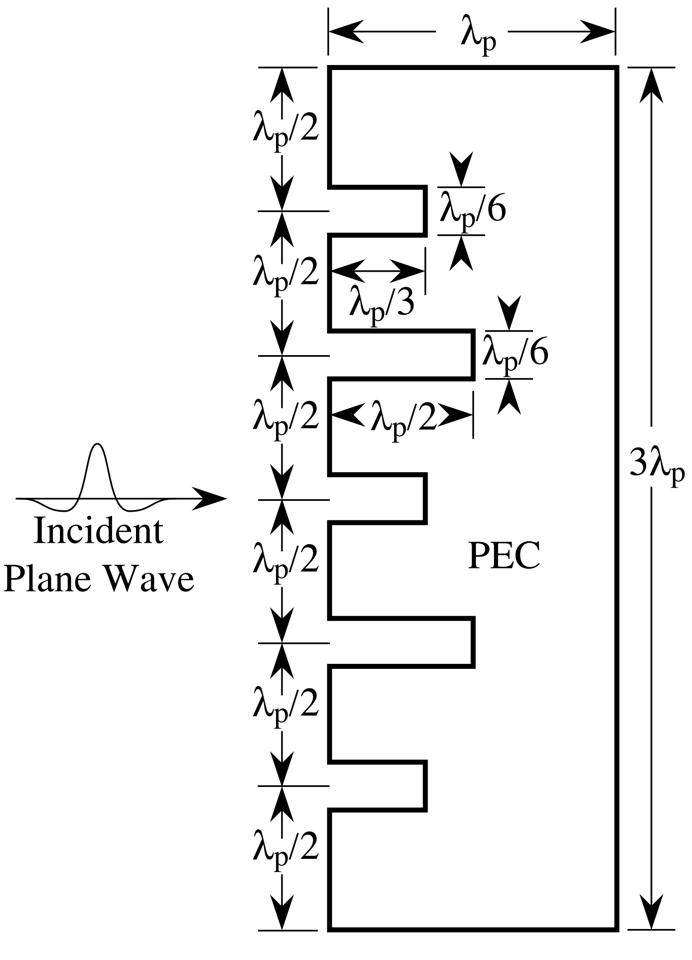

Figure 4: Sketch of scatterer in terms of free-space wavelengths at the peak frequency of the illuminating pulse. For the subsequent results, the slots have been loaded with a dielectric with a relative permittivity of 9.0. Thus, the electrical dimensions of the slots are effectively multiple by a factor of 3.0.

the slots are loaded with a dielectric that has a relative permittivity of 9.0. For this loading, the slots have an electrical height of $\lambda_p/2$ and a width of either λ_p or $1.5\lambda_p$, and hence have a strong resonance. Images will only be shown at a single point in time. Complete MPEG animations corresponding to these images are available from the Web site.

Figures 5-8 show the total H_z field at a point in time slightly after the incident pulse has past the scatterer. The incident pulse, which travels from left to right across the image plane, has a maximum value of unity whereas the total field, measured over the duration of the animation, has a maximum value of approximately 2.88 (A/m). This maximum value occurred at a point earlier in the animations than the time at which these images were recorded. However, to be consistent throughout the entire set of images and to illustrate the usefulness of the log scaling, a normalization of 2.88 (i.e., $x_{\max} = 2.88$) was used for each of these images. Figures 5-7 each used three decades of logarithmic scaling ($N_d = 3$) whereas Fig. 8 used a linear scale. Figure 5 is the gray scale image of the absolute value of the field. Figure 6 is the one-sided color image of the absolute value of the field and Fig. 7 shows the two-sided color image. In Fig. 8 the one-sided mapping is also used. For each of these images a narrow color map is shown along the left side of the image. Note that by using a linear scale, the image in Fig. 8 appears quite faint even along the maximum value of the incident pulse. This is a consequence of the fact that the image has been normalized by 2.88 whereas the incident pulse has a maximum value of unity. Thus, the values in the incident pulse only span one-third of the range of colors in the incident pulse. In the images that incorporate log scaling, field values between 2.88×10^{-3} and 2.88×10^{-2} span the first third of the color map, values between 2.88×10^{-2} and 2.88×10^{-1} span the next third, and value between 2.88×10^{-1} and 2.88 span the final third.

5 Viewing the Output and Animations

Once the PGM or PPM files have been created, there are, as depicted in the bottom third of the flow chart of Fig. 1, several ways in which they can be viewed. First, the files can be viewed as individual frames (stills). Many graphics viewers directly support the PNM formats and any of these could be used to view the individual frames. Alternatively, the files can be "strung together" to make animations of the field. Some graphics packages that are typically used to view still images also provide rudimentary animation capability. In

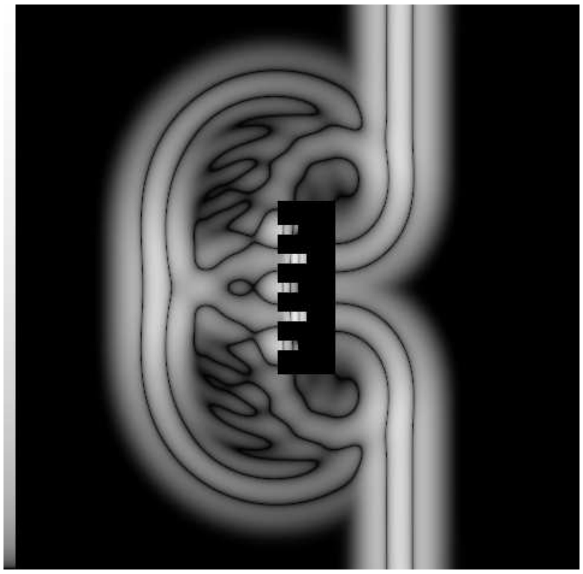

Figure 5: Gray scale plot the absolute value of the field using three decades of scaling.

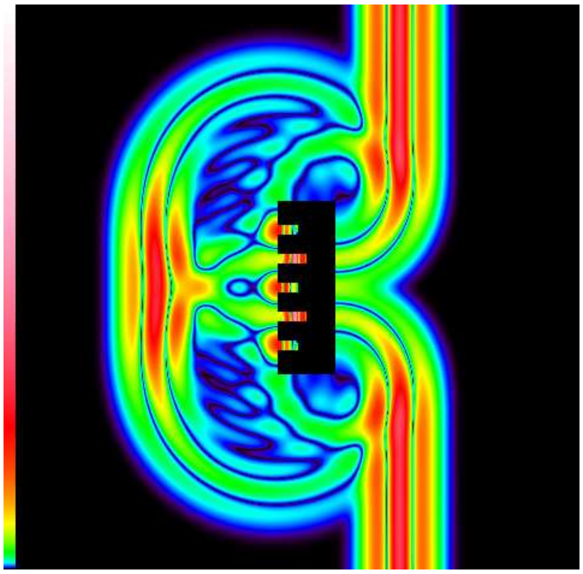

Figure 6: One-sided color image of the absolute value of the field using three decades of scaling.

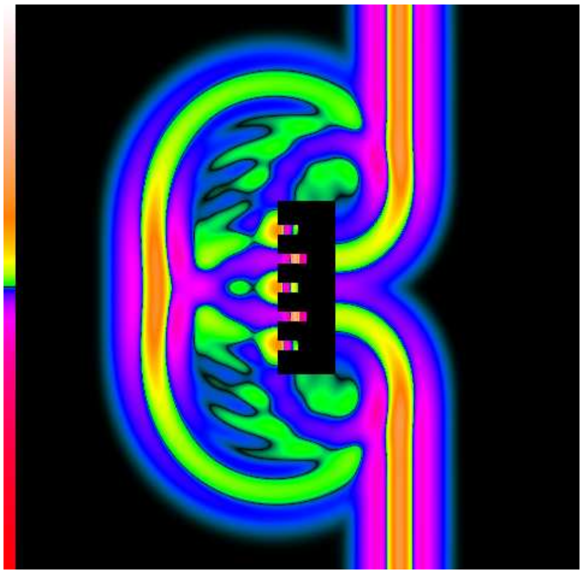

Figure 7: Two-sided color image of the field using three decades of scaling.

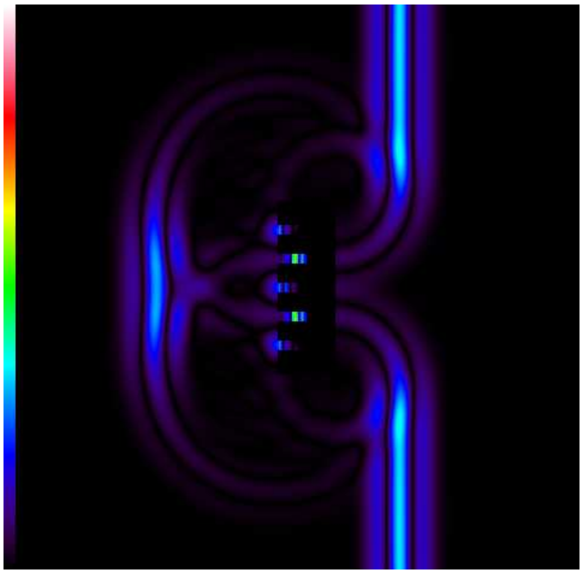

Figure 8: One-sided color image of the absolute value of the field using a linear scale.

addition, there is public domain software that will convert a sequence of frames into an MPEG file.

As an example of how single frames can be viewed on X Windows systems, the ***xli*** program can be used to view frame 39 with a command such as ***xli frame.39.ppm*** (assuming the base name is ***frame*** and color output was generated). A series of frames can be viewed by using the built-in filename expansion provided by Unix shells. For example, to view frames 30 to 49, the following command would be used

```
xli frame.[34]?.ppm
```

(Please refer to the appropriate shell documentation for the specifics concerning command-line file name substitution.) In ***xli***, the ***n*** or ***f*** keys are used to cycle through the frames. A rudimentary animation can be constructed using ***xli*** by specifying that displayed image should change after a given delay. For example,

```
xli -delay 1 frame.?.ppm  frame.??.ppm  frame.???.ppm
```

will cycle through the files (starting with files with a single digit, then two digit files, then three digit files) with a one second delay between each.

Besides ***xli***, there are many utilities that can be used to view images on X Windows systems. For instance, the popular image display utility ***xv*** directly supports PNM files and can be used for simple animations. The viewer ***display*** is available as part of the public-domain ImageMagick routines and can be used to automatically cycle through all the frames of the animation.

As a last resort, if one can display GIF files, the PGM and PPM files can be converted to GIF by using the ***ppmtogif*** routine provided in the ***pbmplus*** package mentioned earlier. GIF files have the advantage of being compressed and hence are much smaller than the corresponding PNM file. However, they have the disadvantage of being precompressed to eight-bit color and thus could obscure important details in certain images.

As an alternative to the low-level animation capability provided by routines such as ***display***, ***xli***, and ***xv***, a better animation utility is distributed with the ImageMagick routines. This animation utility, called ***animate***, works directly with a wide range of file formats, including PGM, PPM, and GIF. Some advantages of using this routine over the others is that it is less likely to cause color map "flashing" as it cycles through the frames, and the speed and direction of the animation are much easier to control.

The ***animate*** utility is relatively powerful and easy to use and thus is quite convenient to use on an X Windows workstation. However, in many instances it is desirable to share an animation with colleagues or to make it available via the Web. In these instances, one cannot rely upon a one-size-fits-all utility and it may not be practical to make all the underlying frames of an animation available. Instead, MPEG files provide a concise and portable means to distribute and playback animations.

MPEG (an acronym for Moving Pictures Expert Group) is a specification for the architecture and file formats of an encoder and a decoder for image sequences. The MPEG format has emerged as a standard format for image sequences transmitted over computer networks or retrieved from CD-ROMs or other media. Among its attractive features is the ability to trade off file size against the quality of the compression. For the purposes of this discussion the MPEG encoder can be viewed as a black box into which the PNM files are fed, compressed, and written to the output file which can then be transmitted and/or replayed on demand. It is not difficult to obtain compression rates of 90% or better on image sequences with substantial coherence between adjacent frames (such as those in our application), without sacrificing visual quality of the display.

In order to construct an MPEG file from the individual frame files, one can use the ***mpeg_encode*** utility that is available via anonymous ftp from ***mm-ftp.cs.berkeley.edu***. This utility comes with documentation and examples so that it is relatively easy to use. The basic procedure is to write a "parameter" file that tells the encoder the type of compression desired, the name and number of files, etc. The name of the parameter file is given as an argument to the ***mpeg_encode*** utility. For example, if the parameter file were named ***params***, the command ***mpeg_encode params*** would generate an MPEG file in accordance with the instructions contained within the file ***params***. The following is a typical parameter file that would be used to construct an output MPEG file named ***animation.mpg*** from the 100 image files ***frame.1.ppm***, ***frame.2.ppm***, ... ***frame.100.ppm***.

```
IQSCALE 4
PQSCALE 7
BQSCALE 7
PSEARCH_ALG TWOLEVEL
BSEARCH_ALG SIMPLE
GOP_SIZE 16
SLICES_PER_FRAME 1
PIXEL HALF
RANGE 12
```

```
PATTERN IBBPBBPBBPBBPBBP
FORCE_ENCODE_LAST_FRAME
INPUT_DIR .
INPUT
frame.*.ppm [1-100]
END_INPUT
BASE_FILE_FORMAT PPM
INPUT_CONVERT *
REFERENCE_FRAME DECODED
OUTPUT animation.mpg
```

Please refer to the documentation that accompanies ***mpeg_encode*** for more information about the items in the parameter file. Additional examples of the use of ***mpeg_encode*** can be found at the previously mentioned ***Animations*** Web site. If one does not have access to an MPEG viewer, one can also be obtained from the Berkeley ftp site. (It should be noted that several options are also available when running the viewer. These options can also affect the speed and quality of the playback of an MPEG file!)

6 Conclusions

The animation scheme presented here is relatively simply to understand and easy to implement even for those with no previous experience generating such graphics. Despite this simplicity, the resulting animations can often provide information about the underlying physical aspects of a problem that would be difficult to obtain using traditional plots or "still" images. The scheme is not inherently tied to any single computational method and can be used in conjunction with virtually all standard problem solving techniques (e.g., the finite element method, FDTD, or the method of moments). Finally, for those interested in having total control of their own graphics/animation package, the scheme presented here can serve as a solid foundation on which to build.

A Data Extraction Routine

The tasks which a data extraction routine must accomplish were discussed in Sec. 2. A sample routine that completes these tasks is the ***wrtraw*** routine which follows:

```
C.....Subroutine to write raw output files.
      subroutine wrtraw(nframe,istart,iend,ihop,jstart,jend,jhop,
     &     base,field,ilead,xmax)
```

```
      implicit real (a-h,o-z)
      real field(ilead,*)
      character filnam*1024, base*(*), frmt*30
C.....
C Local variables:
C     frmt   = internal file used to construct output file name
C     ilen   = number of non-blank characters in the base name
C     idum,jdum = dummy loop counters
C     ipnt   = record pointer for output file
C     ndigit = number of digits in current frame number
C Input variables:
C     field,ilead = two dimensional array with leading dimension ilead
C     base   = base name of output files
C     nframe = frame number (assumed set by calling program)
C     istart,jstart = starting column and row for output
C     iend,jend = ending column and row for output
C     ihop,jhop = column and row strides
C Output variable:
C     xmax   = largest absolute value of output
C.....
C.....Determine the location of last character in the base name.
      ilen = index(base,' ') - 1
C.....Calculate the number of digits in the current frame number.
      ndigit = log10(real(nframe)+0.01) + 1
C.....Construct the proper format for the desired output file name.
      write(frmt,'(a2,i4,a5,i9,a1)')'(a',ilen,',a1,i',ndigit,')'
C.....Put this together to obtain the proper output file name.
      write(filnam,frmt)base,'.',nframe
C.....Open the output file and write header info.
      open(68,file=filnam,status='new',form='unformatted',
     &     access='direct',recl=4,err=2000)
      write(68,rec=1)real((iend-istart)/ihop + 1)
      write(68,rec=2)real((jend-jstart)/jhop + 1)
C.....
C     Loop through writing of data.  The first entry is for the upper
C     left corner, the next is for the next column over, etc.
C.....
      ipnt=2
      do 20 jdum=jend,jstart,-jhop
         do 10 idum=istart,iend,ihop
            ipnt=ipnt + 1
            if(abs(field(idum,jdum)).gt.xmax)xmax=abs(field(idum,jdum))
            write(68,rec=ipnt)field(idum,jdum)
 10      continue
 20   continue
      close(68)
      return
 2000 write(6,*)'Error opening output file in wrtraw.'
      stop
      end
```

First, ***wrtraw*** opens an output file whose name is the base name ***base*** with the frame number ***nframe*** appended. We do not want to unreasonably restrict the length of the base name and hence assume it can have an arbitrary length (in practice, the code requires the length to be slightly less that 1024 characters). This fact necessitates the use of code which is somewhat obscure (ref. lines 25-31) to construct the output file name. The key is to understand the use of Fortran internal files and to recognize the role the internal file ***frmt*** plays in the construction of a format statement. Despite this bit of arcane code, the rest should be fairly self-explanatory.

Note that the output file is opened as an unformatted file in order to store values in binary form. Although binary storage is not essential, it can greatly reduce the size of the output file. The record length here is set to four which typically means four bytes, i.e., the usual number of bytes in a single precision floating point number. Unfortunately, some compilers interpret the record length differently. For example, the Digital Equipment Corporation Fortran compiler assumes the record length specifies the number of four-byte blocks. Hence, for these compilers the appropriate record length would be one. Machines, such as Crays, that use 64 bit words may require modification of this code. As a last resort, ASCII storage can be used, but this greatly increases the space needed to store results and would require the modification of the ***rw2pnm*** routine to permit the reading of ASCII input. (Since one typically deals with ASCII data, these simple changes are not presented here.)

B Routines for Converting to PNM Format

The main routine used to convert the raw files to PGM or PPM files is ***rw2pnm***. (As mention previously, it is important to note that the C version of this code, which is available from the Web site, is much simpler and significantly faster than this Fortran code.) This program relies upon the subroutines ***cmap1*** and ***cmap2*** to generate the one-sided and two-sided color maps, respectively. The subroutine ***hsvrgb*** is used to convert from values in HSV space to RGB values. The complete program listing for ***rw2pnm*** and its subroutines follows:

```
      program rw2pnm
C.....
C     This program converts "raw" binary files to pgm or ppm files.
C     Log compression can be used.
C.....
      implicit real (a-h,o-z)
      character base*1024, filein*1024, filout*1024,
     &     frmt*20, ccol*4, crow*4, tail(2)*4
      logical input, output
      data tail /'.pgm', '.ppm'/
```

```
C.....
C     h,s,v    = hue, saturation, and brightness (or "value") for pixel
C     ir,ig,ib = corresponding red, green, and blue values for pixel
C     eps      = "small" number added to data to prevent underflows
C     xnorm    = number used to normalize all input
C     xpnt     = the current field value obtained from the input file
C     i,j      = dummy counters
C     iin,iout = record pointers for the input and output files
C     irow,icol= number of rows and columns of data (integers)
C     crow,ccol= number of rows and columns of data (characters)
C     nframe   = current frame number
C     ndigit   = number of decimal digits in the current frame number
C     ndec     = number of decades of compression
C     input,output = true if input/output file exists (logical)
C     base     = base input and output file names
C     filein,filout = complete input and output file names
C     frmt     = internal file used to construct file names
C     iside    = indicates two-sided or one-sided data
C     icolor   = indicates color or grayscale
C     tail     = contains possible extensions for output file names
C     xtmp     = temporary variable for holding result
C.....
      write(6,*)'Enter the base name of the raw files.'
      read(5,'(a)')base
C.....Determine the location of last character in the base name.
      ilen = index(base,' ') - 1
 10   write(6,*)'Enter the normalization.'
      read(5,*)xnorm
      if (xnorm .le. 0.) goto 10
 20   write(6,*)'Enter the number of decades (0 = linear scale).'
      read(5,*)ndec
      if (ndec .lt. 0) goto 20
 30   write(6,*)'Do you want color or grayscale output?  (1 = ',
     &     'grayscale; 2 = color)'
      read(5,*)icolor
      if (icolor .eq. 2) then
 40      write(6,*)'Do you want to obtain "one-sided" output by',
     &        ' taking the absolute value'
         write(6,*)'of the input data?  (1 = one-sided; 2 = two-sided)'
         read(5,*)iside
         if (iside .ne. 1 .and. iside .ne. 2) goto 40
      elseif (icolor .ne. 1) then
         goto 30
      endif
      nframe = 0
 1000 nframe = nframe + 1
C.......Calculate the number of digits in the current frame number.
         ndigit = log10(real(nframe)+0.01) + 1
C.......Construct the proper format field for the input file name.
         write(frmt,'(a2,i2,a5,i9,a1)')'(a',ilen,',a1,i',ndigit,')'
C.......Put this together to obtain the input file name.
         write(filein,frmt)base,'.',nframe
C.......Construct the output file name.
         filout = filein
```

```
        filout(ilen+ndigit+2:ilen+ndigit+6) = tail(icolor)
C.......Check if input file is available.  Terminate if not.
        inquire(file=filein,exist=input)
        if (.not. input) then
           write(6,*)'Input not found: ',filein(1:index(filein,' '))
           stop
        endif
C.......If output file already exists, skip to next file.
        inquire(file=filout,exist=output)
        if (output) then
           write(6,*)'Skipping:',filout(1:index(filout,' '))
           goto 1000
        endif
        write(6,*)'Processing: ',filein(1:index(filein,' '))
C.......Open the input file and read the size of the image.
        open(55,file=filein,form='unformatted',status='old',
    &        access='direct',recl=4)
        read(55,rec=1)xtmp
        icol = int(xtmp)+10
        read(55,rec=2)xtmp
        irow = int(xtmp)
        write(ccol,'(i4)')icol
        write(crow,'(i4)')irow
C.......Open the output file and write the header information.
        open(66,file=filout,form='unformatted',status='new',
    &        access='direct',recl=1)
        write(66,rec=1)'P'
        write(66,rec=2)char(52+icolor)
        write(66,rec=3)char(10)
        write(66,rec=4)ccol(1:1)
        write(66,rec=5)ccol(2:2)
        write(66,rec=6)ccol(3:3)
        write(66,rec=7)ccol(4:4)
        write(66,rec=8)' '
        write(66,rec=9)crow(1:1)
        write(66,rec=10)crow(2:2)
        write(66,rec=11)crow(3:3)
        write(66,rec=12)crow(4:4)
        write(66,rec=13)char(10)
        write(66,rec=14)'2'
        write(66,rec=15)'5'
        write(66,rec=16)'5'
        write(66,rec=17)char(10)
C.......Initialize record pointers
        iout=18
        iin =3
C.......Calculate "epsilon" that is added to values if compressing.
        if (ndec.ne.0) eps = 10.0**(-ndec)*xnorm/(1.0-10.0**(-ndec))
C.......Loop through all pixels in the image.
        do 200 j=1,irow
           do 100 i=1,icol
              if (i .le. 10) then
C.............Draw the color map if within the first 10 columns.
                 if (icolor .eq. 1 .or. iside .eq. 1) then
```

```
                     xpnt = real(irow-j)/real(irow-1)*xnorm
                  else
                     xpnt = (-2./real(irow-1)*real(j)
     &                    + real(irow+1)/real(irow-1))*xnorm
                  endif
               else
C..............If not in first 10 columns, get data from file.
                  read(55,rec=iin,err=2000)xpnt
                  iin = iin+1
               endif
C..............If the point is out of range, whack it down to size.
               if (abs(xpnt) .gt. xnorm) xpnt = xpnt/abs(xpnt)*xnorm
C..............Use log scaling if requested.
               if (ndec .ne. 0) then
                  xtmp = log10((abs(xpnt)+eps)/(xnorm+eps))/
     &                 real(ndec)+1.
               else
                  xtmp = abs(xpnt/xnorm)
               endif
               if (icolor .eq. 1) then
C.................Grayscale output.
                  write(66,rec=iout)char(int(255*xtmp))
                  iout = iout+1
               else
                  if (iside .eq. 1) then
C.................Use color map #1 for one-sided data.
                     call cmap1(h,s,v,xtmp)
                  else
C.................Reconstruct sign & use color map #2 for two-sided.
                     if (xpnt .lt. 0.) xtmp = -xtmp
                     call cmap2(h,s,v,xtmp)
                  endif
C.................Convert (h,s,v) to (r,g,b) and write output.
                  call hsvrgb(h,s,v,ir,ig,ib)
                  write(66,rec=iout)char(ir)
                  write(66,rec=iout+1)char(ig)
                  write(66,rec=iout+2)char(ib)
                  iout = iout+3
               endif
 100           continue
 200        continue
 2000    close(55)
         close(66)
      goto 1000
      end
C***********************************************************************
C.....Color mapping appropriate for one-sided data.
      subroutine cmap1(h,s,v,xpnt)
      real h, s, v, xpnt, pi
      parameter (pi=3.141593)
      if (xpnt .gt. 0.8) then
         h = 330. + 30*(1.-xpnt)/0.2
         s = (1.-xpnt)/0.2
         v = 1.0
```

```
      elseif (xpnt .gt. 0.2) then
         t = 2.0*pi*((xpnt-0.2)/.6)
         h = 240.0*(0.8-xpnt)/0.6 + 5.0*sin(t)
         s = 1.0
         v = 1.0
      else
         h = 270.0*(.2-xpnt)+240.0
         s = 1.0
         v = xpnt/0.2
      endif
      return
      end
C***********************************************************************
C.....Color mapping appropriate for two-sided data.  Zero = black.
      subroutine cmap2(h,s,v,xpnt)
      real h, s, v, xpnt
      if (xpnt .gt. 0.8) then
         h = 30. - 30.*(xpnt-0.8)*(xpnt-0.8)/0.04
         s = (1.0-xpnt)/0.2
         v = 1.0
      else if (xpnt .gt. .3) then
         h = 135.0 - 105.*(xpnt-.3)/.5
         s = 1.0
         v = 1.0
      else if (xpnt .gt. 0.) then
         h = 180.0 - 45.*xpnt/.3
         s = 1.0
         v = xpnt/.3
      else if (xpnt .gt. -.3) then
         h = (180. - 45.*xpnt/.3)
         s = 1.0
         v = abs(xpnt)/.3
      else
         h = 225. - 135.*(xpnt+.3)/.7
         s = 1.0
         v = 1.0
      endif
      return
      end
C***********************************************************************
C.....Subroutine to convert (h,s,v) values to (r,g,b) values.
      subroutine hsvrgb(h,s,v,ir,ig,ib)
      implicit real (a-h,o-z)
      real pqvt(4)
      integer irpnt(6),  igpnt(6),  ibpnt(6)
      data irpnt/3,2,1,1,4,3/,igpnt/4,3,3,2,1,1/,ibpnt/1,1,4,3,3,2/
C.....
C     input:   0 <= h <= 360    (hue)
C              0 <= v <= 1.0    (value or brightness)
C              0 <= s <= 1.0    (saturation)
C     output:  0 <= ir,ig,ib <= maxval
C.....
      if (s .eq. 0.0) then
         ir = 255.0*v
```

```
227         ig = ir
228         ib = ir
229         return
230       endif
231       if (h .eq. 360.0) h=0.0
232       h = h / 60.0
233       i = int(h)
234       f = h-i
235       pqvt(3) = 255.0*v
236       pqvt(1) = pqvt(3)*(1.0-s)
237       pqvt(2) = pqvt(3)*(1.0-s*f)
238       pqvt(4) = pqvt(3)*(1.0-s*(1.0-f))
239       i  = i + 1
240       ir = pqvt(irpnt(i))
241       ig = pqvt(igpnt(i))
242       ib = pqvt(ibpnt(i))
243       return
244       end
245 C***********************************************************************
```

After the input and output files names have been constructed based on the current frame number and the given base name, the status of these files is checked. If the input file is not found, the program terminates (lines 67 to 71). If the input file exists and the output file already exists, another output file is *not* created. Instead, the frame number is advanced and the next set of input and output files is checked (lines 73 to 77). Only if the input file exists and the output file does not exist is a new output file created. This is done in order to permit parallel processing of the data—this program may be invoked more than once to work on the same set of files and the file handling should insure that separate invocations do not interfere with each other.

The majority of code should be easy to understand by experienced Fortran programmers. However, the use of the Fortran intrinsic function ***char*** in lines 140, 153, 154, and 155 to convert an integer value (which is between 0 and 255) to the corresponding single byte value (characters are inherently one byte long) may be unfamiliar.

The subroutine ***cmap1*** (ref. lines 165-185) maps a one-sided field to a color. The fields are assumed to be between zero and unity and the subroutine returns the corresponding location in the HSV color space.

The subroutine ***cmap2*** (ref. lines 186-211) maps a two-sided field to a color. The fields are assumed to be between -1 and 1 and the subroutine returns the corresponding location in the HSV color space.

The subroutine ***hsvrgb*** (ref. lines 213-244) converts from HSV values to RGB values.

C Software Sites

As mentioned previously, the URL *http://www.eecs.wsu.edu/~schneidj/Animations/* provides all the code discussed in this paper. Additionally, FDTD code is provided to illustrate the use of these routines. The Internet sites listed below are also linked to this page.

The entire suite of *pbmplus* routines is available from a number of sites (via anonymous ftp) including *ftp.x.org* from the directory *R5contrib*. The *netpbm* routines provides a superset of the functionality provided by the *pbmplus* routines and can also be obtained from the same location.

The ImageMagick routines can be obtained from
http://www.wizards.dupont.com/cristy/ImageMagick.html.

The *mpeg_encode* utility is available via anonymous ftp from *mm-ftp.cs.berkeley.edu* in the directory *pub/multimedia/mpeg/encode* while the *mpeg_play* utility is found in the directory *pub/multimedia/mpeg/play*. Additional MPEG information can be obtained from
http://www.arc.umn.edu/GVL/Software/mpeg.html and
http://www.eit.com/techinfo/mpeg/mpeg.html.

The commercial packages MATLAB and Mathematica, though not strictly graphics packages, provide a broad range of plotting tools. More information can be obtained from: *http://www.mathworks.com/* and *http://www.wri.com/*.

The commercial packages IDL and PV-WAVE share a common ancestry but have, for some time now, been developed independently. More information about these graphics packages can be obtained from *http://www.rsinc.com/idl/* and
http://doc.boulder.vni.com/pvwave.dir/wavehome.html. AVS produces another powerful commercial package that can be found described at *http://www.avs.com/*. Additionally, NAG produces the Explorer graphics package that is described at
http://www.nag.co.uk:70/1h/Welcome_IEC.

VTK and Khoros are also powerful graphics packages. But, unlike the previously mentioned packages, these can be obtained for free by individuals (although some restriction may apply—please see the appropriate licensing agreements). The relevant URL's are *http://www.cs.rpi.edu/~martink/* and *http://www.khoral.com/Technology/home.html*.

Acknowledgments

J.B.S. and P.J.F. acknowledge the Office of Naval Research, Code 321OA, for support of this work. K.L.S. is grateful to Dr. Ron Chan for his assistance in downloading and setting up the various public-domain utilities.