Defibrillation tutorial

SCIRun 4.3 Documentation

Center for Integrative Biomedical Computing
Scientific Computing & Imaging Institute
University of Utah

SCIRun software download:

http://software.sci.utah.edu

Center for Integrative Biomedical Computing:

http://www.sci.utah.edu/cibc

Supported by:

NIH grant P41- RR12553-07

Author(s):
Michael Steffen, Jess Tate, Jeroen Stinstra

Contents

Chapter 1

Overview

This tutorial demonstrates several tools within SCIRun for building models and solving simulations using imaging data. It describes a pipeline using both preprocessed images and user generated geometric fields to create a computational mesh and adapt the mesh for our computational requirements. It then continues to setup a finite element simulation and demonstrate the visualization of results.

This tutorial assumes basic knowledge of SCIRun: placing modules into a SCIRun network, connecting modules, visualizing data, etc. If the reader is not familiar with these operations, consult the Basic Tutorial, also distributed in the SCIRun documentation.

1.1 Defibrillation Model

This tutorial describes the tools and steps required to solve quasi-static volume conductor problems with the inclusion of electrodes and their known potentials. The problem will be solved on two separate domains. The first domain, a homogeneous cube, is used to demonstrate the basic techniques required to solve this type of problem within SCIRun. No external data is required and the plate electrodes are sized and placed interactively in the view window. These techniques are then extended to an inhomogeneous model of the human torso using a can, wire, and plate electrodes. Again, these electrodes are interactively placed within the torso.

The torso model is based on a series of cross-sectional MRI scans that has been hand-segmented into regions with various conductivities. These regions include the heart (ventricles and atria), blood, bone, lung, liver, kidney, fat, muscle, bowel gas, connective tissues and other.

The steady state electrical potential in an inhomogeneous volume conductor is described by the equation

$$\nabla \cdot (\boldsymbol{\sigma} \nabla \Phi) = 0, \tag{1.1}$$

where $\boldsymbol{\sigma}$ is a conductivity tensor field and Φ is the electric potential. Our goal is to solve the above equation, given a mesh, a set of known conductivities, and a set of known potentials corresponding to electrode locations.

We will impart Dirichlet boundary conditions anywhere the electric potential is known and Neumann boundary conditions on the surface of the object being simulated. Dirichlet boundary conditions imply

$$\Phi(x, y, z)|_{\bar{\Omega}_k} = V_k \tag{1.2}$$

where V_k is the known potential of electrode k, and $\bar{\Omega}_k$ specifies the domain coincident with electrode k. Neumann boundary conditions imply that

$$\left.\frac{\partial \Phi}{\partial n}\right|_{\bar{\Omega}} = 0 \tag{1.3}$$

on areas of the boundary not coincident with any $\bar{\Omega}_i$.

The finite element method will be employed to solve the above equation on meshes corresponding to our simulation domains. A brief explanation of the finite element method is provided in Chapter 2.

The remainder of the tutorial is split into three main sections:

1. Generating and solving a defibrillation like simulation on a cube domain.
2. Generating a SCIRun network which will aid the user in placement of various electrodes within a torso.
3. Solving the defibrillation simulation on the full human torso.

As a final note, the exact placements of electrodes within the following tutorial are not meant to represent realistic scenarios. The major consideration for placement here was to create simulations with attractive visualizations which help with a clearer understanding of the simulation results.

1.2 Software requirements

1.2.1 SCIRun Compatibility

The modules demonstrated in this tutorial are available in SCIRun version 4.2 and higher and this tutorial is not compatible with any older version of SCIRun. Also be sure to update your SCIRun version to the latest built available from the SCI software portal (http://software.sci.utah.edu), which will include the latest bug fixes and will make sure that the capabilities demonstrated in this tutorial are up to date.

1.2.2 Required Datasets

This tutorial relies on several datasets that are part of the SCIRunData bundle. To obtain these datasets, please go to the SCI software portal at http://software.sci.utah.edu, then hit **Download SCIRun** and instead of the SCIRun source or binary files, download the SCIRunData zip files. Note the latter is available as a Windows zip file or as a Linux gzip file. Both however contain the same datasets and only one of them has to be downloaded.

Chapter 2

Finite Element Modeling

As was previously stated, the steady state electrical potential in an inhomogeneous volume conductor is described by the equation

$$\nabla \cdot (\boldsymbol{\sigma} \nabla \Phi) = 0, . \tag{2.1}$$

and again, the boundary conditions are given by:

$$\Phi(x, y, z)|_{\Omega_k} = V_k \tag{2.2}$$

$$\left.\frac{\partial \Phi}{\partial n}\right|_{\Omega} = 0. \tag{2.3}$$

The finite element approximation beings by assuming our approximate solution takes the form

$$\bar{\Phi}(x, y, z) = \sum_i \Phi_i N_i(x, y, z), \tag{2.4}$$

where $\{N_i\}$ are a set of basis functions and $\{\Phi_i\}$ are a set of unknown coefficients. For the typical linear shape functions, N_i and Φ_i can be thought of as the shape function and coefficient associated with each grid node i.

The Galerkin method for solving the above equation begins by substituting our approximate solution $\bar{\Phi}$ for Φ in (2.1). This gives us

$$\nabla \cdot (\boldsymbol{\sigma} \nabla \sum_i \Phi_i N_i) = 0. \tag{2.5}$$

This is the so called "strong form" of the equation. The weak form comes by integrating both sides of (2.5) against a "trial function" (in this case we will use N_j as a trial function, chosen from the same set of basis functions as above. This leaves us with

$$\int_\Omega \nabla \cdot (\boldsymbol{\sigma} \nabla \sum_i \Phi_i N_i) N_j \, d\mathbf{V} = \int_\Omega 0 \cdot N_j \, d\mathbf{V}. \tag{2.6}$$

Integration by parts and further simplification yields, which satisfies

$$\sum_i \Phi_i \int_{\Omega - \bar{\Omega} - \bar{\Omega}_k} \boldsymbol{\sigma} \nabla N_i \nabla N_j \, d\mathbf{V} = 0. \tag{2.7}$$

This equation automatically satisfies the Neumann boundary conditions above. The solution to (2.7) can be written as the matrix equation:

$$\mathbf{K}\mathbf{\Phi} = 0 \tag{2.8}$$

where $\mathbf{K}_{ij} = \int \boldsymbol{\sigma} \nabla N_i \nabla N_j \, d\mathbf{V}$ is the stiffness matrix, and $\mathbf{\Phi} = [\Phi_1, \ldots, \Phi_n]^T$ is the vector of unknown coefficients. After solving, (2.8), our approximate solution is found by substituting our solution for $\{\Phi_i\}$ into (2.4).

Given a mesh and a set of conductivities, SCIRun has the capability to automatically generate the stiffness matrix $\mathbf{K}$ in (2.8). Adding the known potential values corresponding to the various electrodes is also a simple procedure. The remainder of this tutorial will show how this type of simulation is performed within SCIRun.

Chapter 3

Finite Element Simulation on a Cube

3.1 Building a Hexahedral Mesh

We start by building a hexahedral mesh of cube which will serve as our solution domain. To accomplish this, create a SCIRun network like the one shown in Figure 3.1. The network will consist of the **CreateLatVol** module connected to the **CalculateFieldData** module, followed by the **ShowField** module, and lastly the **ViewScene** module.

Open the user interface to the **CreateLatVol** module and specify the number of nodes to be 32x32x32 by typing those values into the "X Size", "Y Size", and "Z Size" fields. We will be solving this problem with data located in the cells, rather than at the nodes, so chose "Cells (constant basis)" in the "Data at Location" panel. Open the user interface to the **CalculateFieldData** module, and set the expression to `RESULT = 1.0;` to set the entire field to a value of 1.0. Click the "Execute All" button to run the networks and click the "VIEW" button on the **ViewScene** module to view the results. You should see something similar to the results shown in Figure 3.2. The images shown in this tutorial may not be the default rendering view. Manual rotation using the center mouse button may be required for your image to match those here.

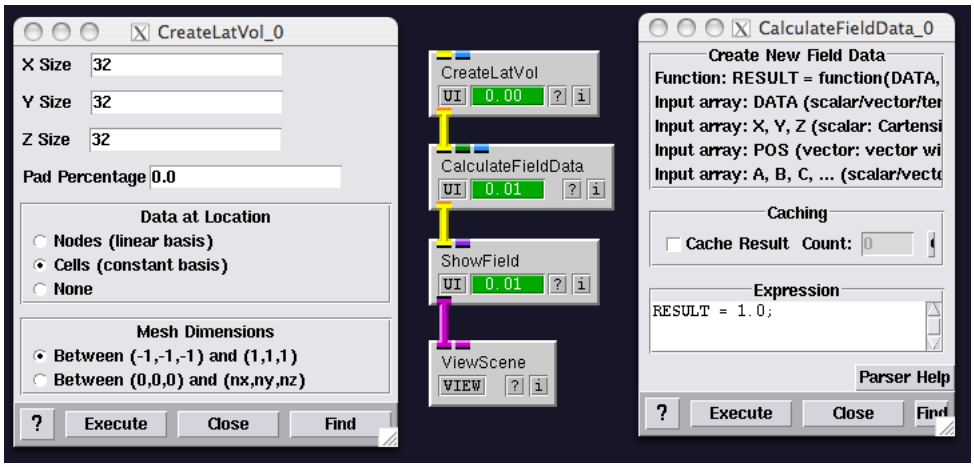

Figure 3.1. Network to generate hexahedral mesh.

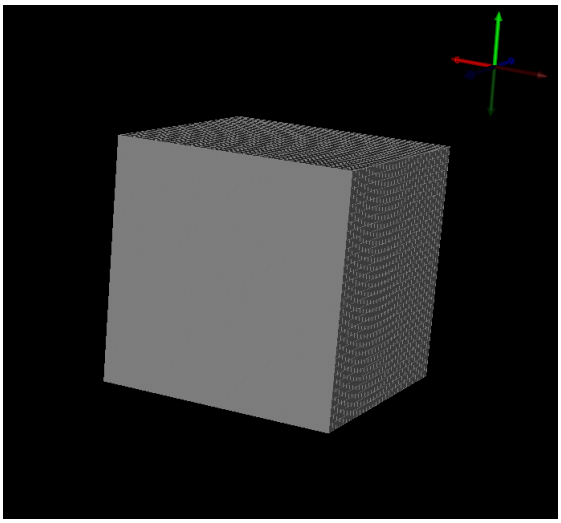

Figure 3.2. Generated hexahedral mesh

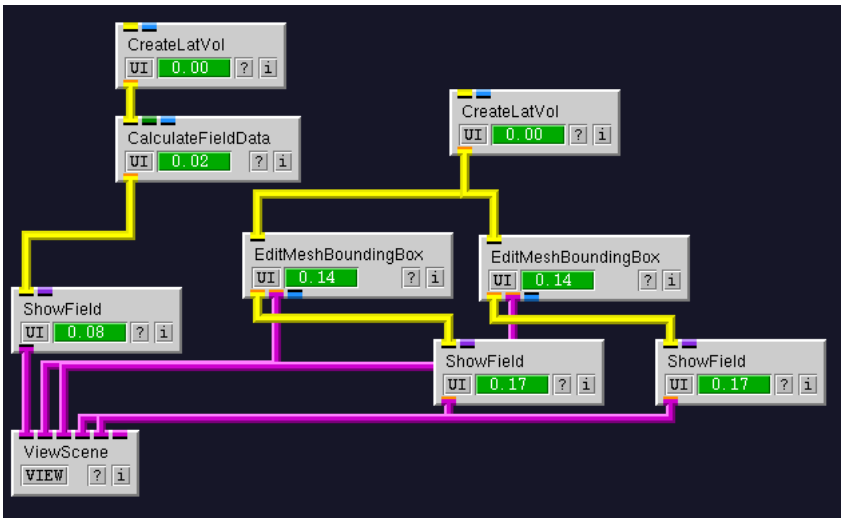

Figure 3.3. Adding plate electrodes to the network.

3.2 Creating Plate Electrode Geometry

The next step in building our network is to add plate anodes and cathodes to the simulation. The electrodes will be modeled as boxes, but this time we will use **EditMeshBoundingBox** modules to allow the user to resize and reposition the electrodes interactively. Add another **CreateLatVol** to the network, this time setting the "X Size", "Y Size", and "Z Size" fields to 2. Connect this new module to two separate **EditMeshBoundingBox** modules, one for the anode and one for the cathode. The **EditMeshBoundingBox** module will output a new transformed field and an editable visualization widget. Connect the yellow field output ports to **ShowField** modules, and connect the pink output ports of both **EditMeshBoundingBox** modules and the new **ShowField** modules to the **ViewScene** module. The modified network should look like that in Figure 3.3.

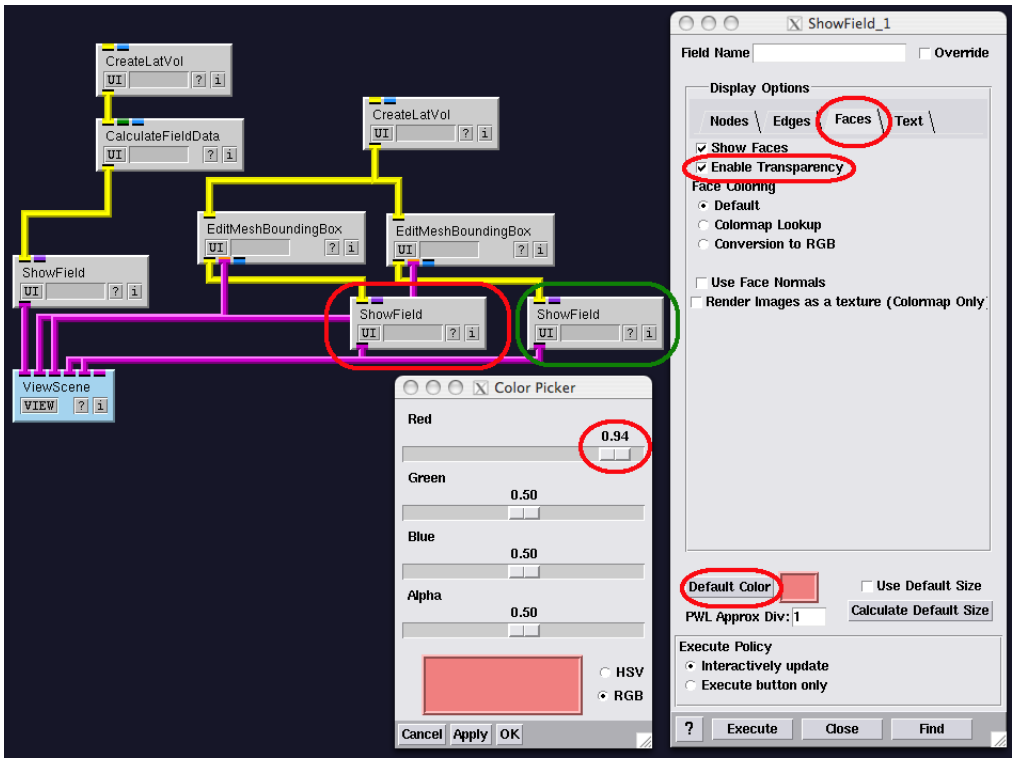

Figure 3.4. Editing the **ShowField** modules for the electrodes.

Before viewing this new network, we want to edit the new **ShowField** modules to distinguish the two electrodes. Open the first **ShowField** module, click on the "Faces" tab, and check the "Enable Transparency" checkbox. Click on the "Default Color" button, and change the color by adding red. Performing these same operations on the second **ShowField** module, this time adding more green to the color. These operations are shown in Figure 3.4.

The network can now be viewed and the positions of the electrodes can be modified in the view window by holding down the Shift key and using the left mouse button to either grab the borders of the electrodes (shown as grey lines) and moving the box, or grab the small cylinders on the box faces and resizing the box. Alternatively, the position and sizes of the electrodes can be set manually in the **EditMeshBoundingBox** modules.

To manually set the electrode size, open the user interface to the first **EditMeshBoundingBox**, make sure the "Center" and "Size" check boxes are checked, and enter values for the center position and size. The remainder of this tutorial will use center and size values of (-1.0, 0.5, 0.5) and (0.2, 1.5, 1.5) for the first electrode and (1.0, -0.5, -0.5) and (0.2, 1.5, 1.5) for the second electrode. An example of this operation is shown in Figure 3.5.

Finally, we can view the scene with all our required geometry, by clicking the "Execute All" button and clicking "VIEW" on the **ViewScene** module. Results should be similar to those in Figure 3.6.

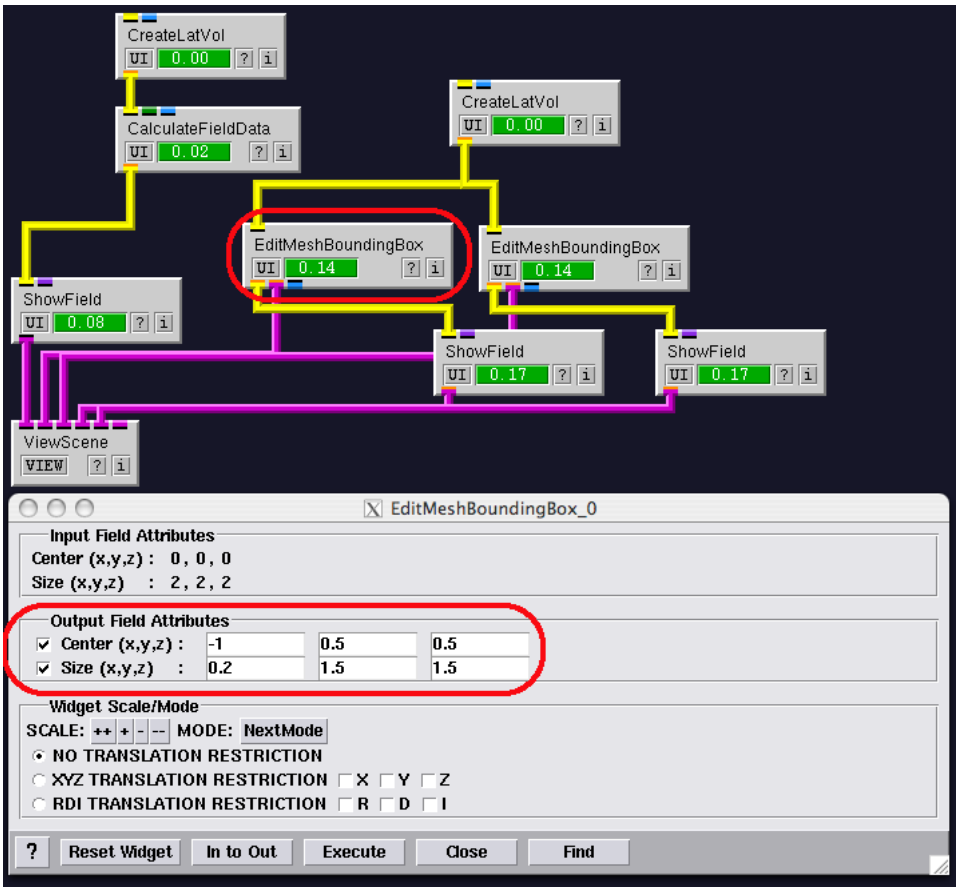

Figure 3.5. Editing the **EditMeshBoundingBox** modules for the electrodes.

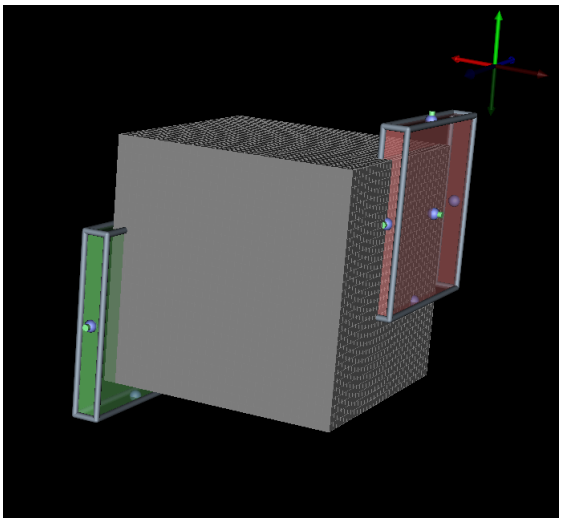

Figure 3.6. Geometry for the problem, including a box simulation domain and two electrodes (red and green).

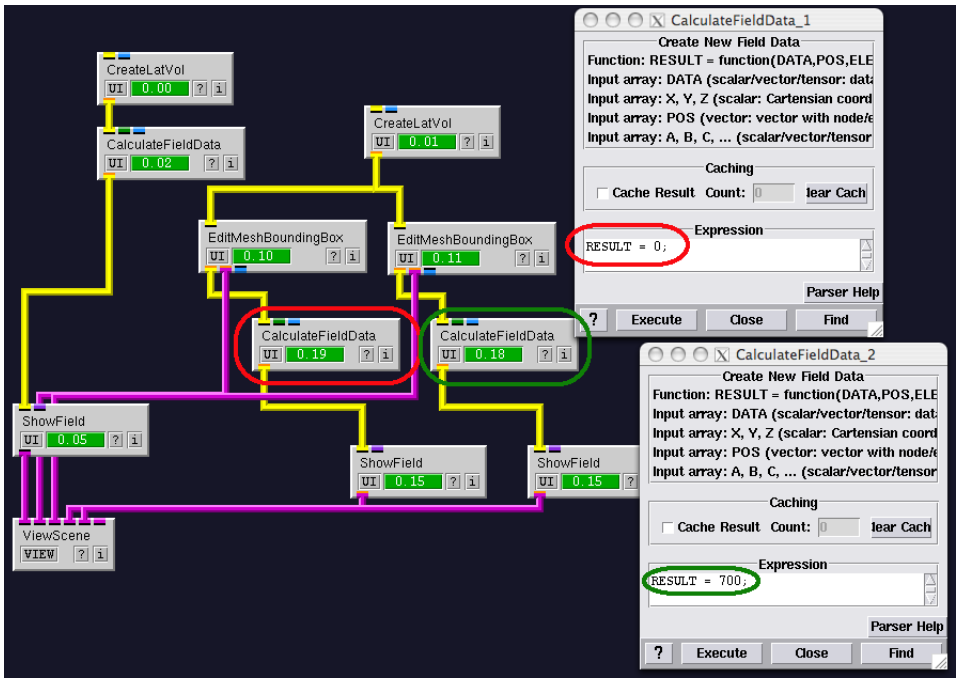

Figure 3.7. Setting potentials for the electrodes.

3.3 Building and Solving the Finite Element Simulation

Now that we have the geometry for our problem defined, we need to build our finite element matrices, set our known values, and solve the simulation. To begin, connect a **CalculateFieldData** module between each of the **EditMeshBoundingBox** modules and their respective **ShowField** modules. Open the first **CalculateFieldData** module and set the expression to `RESULT = 0;`. The expression in the second **CalculatFieldData** module should be set to `RESULT = 700;`. These are the known electric potentials assigned to the two electrodes. The resulting network is shown in Figure 3.7.

Next, we want to map the assigned potentials as known values onto corresponding nodes of the hexahedral mesh. This is accomplished by first joining the two electrode fields together using the **JoinFields** module, connecting the yellow outputs of the electrodes' **CalculateFieldData** modules to the first two inputs on the **JoinFields** module. Next, these values are mapped onto nodes in the Hex mesh using the **MapFieldDataOntoNodes** module. The output of the **JoinFields** module is connected to the first input of the **MapFieldDataOntoNodes** module while the output of the **CalculateFieldData** of the Hex mesh is connected to the third input.

The linear system solver in SCIRun uses values of NaN (not a number) to specify unknowns in one of the input vectors, and our added **MapFieldDataOntoNodes** module will also be used to set values in the mesh outside of the electrodes to NaN. To do this, open the user interface for the module and set the "Default Outside Value" to `nan` and the "Maximum Distance" to `inf`.

The values on the nodes now represent both the unknown (NaN) and known values of our system. We our now ready to build our finite element linear system. To do so, add a **BuildFEMatrix** module into the network, connecting the output of the Hex mesh **Calcu-**

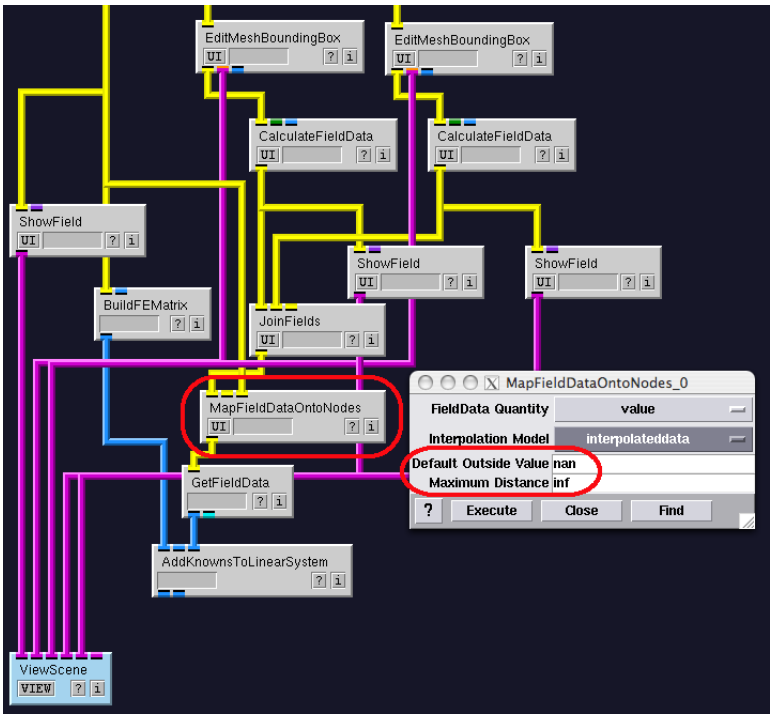

Figure 3.8. Network after adding **AddKnownsToLinearSystem** module.

lateFieldData module to the input of the **BuildFEMatrix** module. To specify the known values, first connect the output of the **MapFieldDataOntoNodes** to a **GetFieldData** module. Next, add a **AddKnownsToLinearSystem** module to the network, connecting the **BuildFEMatrx** to the first input, and the output of the **GetFieldData** to the third input. The current state of the network should look like Figure 3.8.

Continuing, we will solve the linear system, connecting the two outputs of the **AddKnownsToLinearSystem** to the two inputs of a new **SolveLinearSystem** module. The first output of the solver will contain the solution values. We map this onto the mesh by adding a **SetFieldData** module, connecting the outputs of the **MapFieldDataOntoNodes** and **SolveLinearSystem** modules to the first two inputs of the **SetFieldData** module. To visualize the field, we connect the output of the **SetFieldData** into a new **ShowField** module and connect that **ShowField** module to our **ViewScene** module. To make the visualization useful, we will create a color map by adding a **CreateStandardColorMaps** and connecting the output to a **RescaleColorMap** module. The output of the **SetFieldData** module is used as the second input to the **RescaleColorMap** module. And lastly, the output of the **RescaleColorMap** module is used as the second input to the last **ShowField** module. We now have a color mapped version of our Hex mesh as an input to the **ViewScene** module, so be sure to delete the original input to the **ViewScene** module coming from the Hex mesh visualization. The final piece of the network is shown in Figure 3.9 and the expected output is shown in Figure 3.10.

Lastly, the visualization can be made more appealing by making a few changes to the **CreateStandardColorMaps** module and the last **ShowField** module. First, open the user interface to the last **ShowField** module and uncheck the “Show Nodes” and “Show Edges” boxes on the “Nodes” and “Edges” tabs. This will result in a visualization only

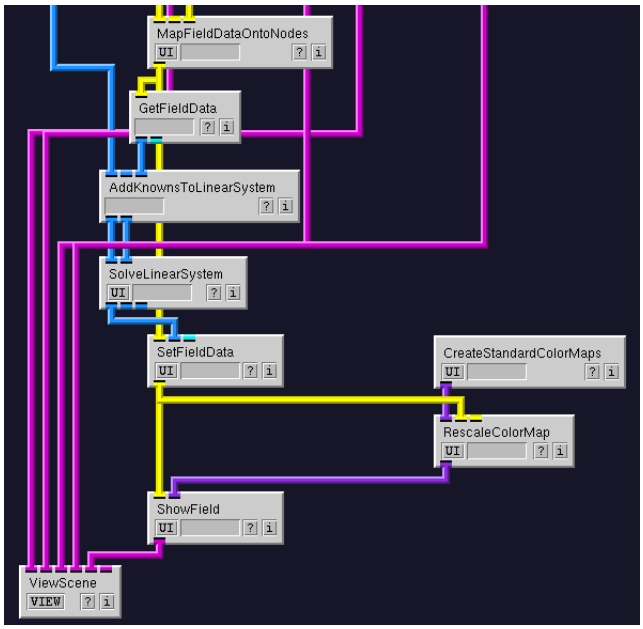

Figure 3.9. Final simulation network for the two electrode problem.

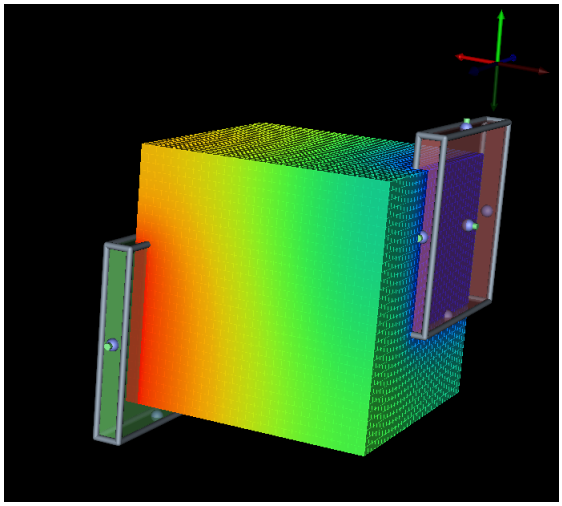

Figure 3.10. Results of the finite element simulation with two electrodes.

showing faces with a smooth gradient. Another effective change is to force the color map to have less resolution. Open the **CreateStandardColorMaps** user interface and change the "Resolution" slider to a setting of 25. This will result in a less smooth color mapping, where the boundaries of the discrete colors act as contour lines of the solution on the surface of the domain. These user interface changes are shown in Figure 3.11 and the new output is shown in Figure 3.12.

As a reminder, the positions and sizes of the electrodes in the visualization are interactively editable. The electrodes can be moved and the simulation will be rerun with the new electrode geometry.

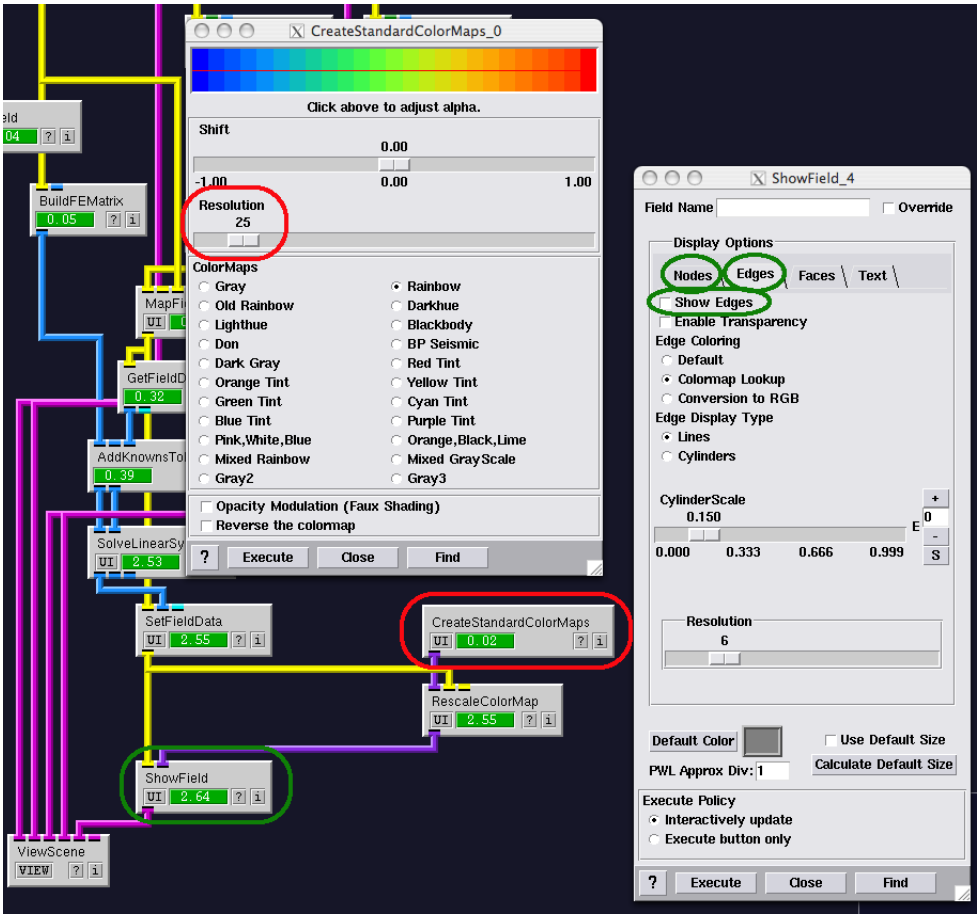

Figure 3.11. Changing the **ShowField** and **CreateStandardColorMaps** settings for an effective visualization.

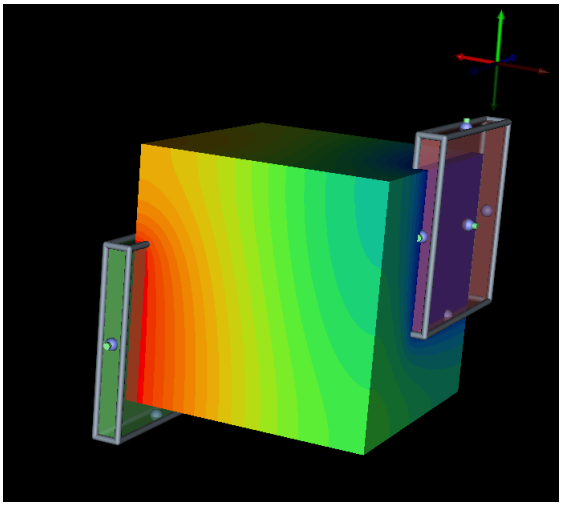

Figure 3.12. Results of the finite element simulation with two electrodes after adjusting view settings.

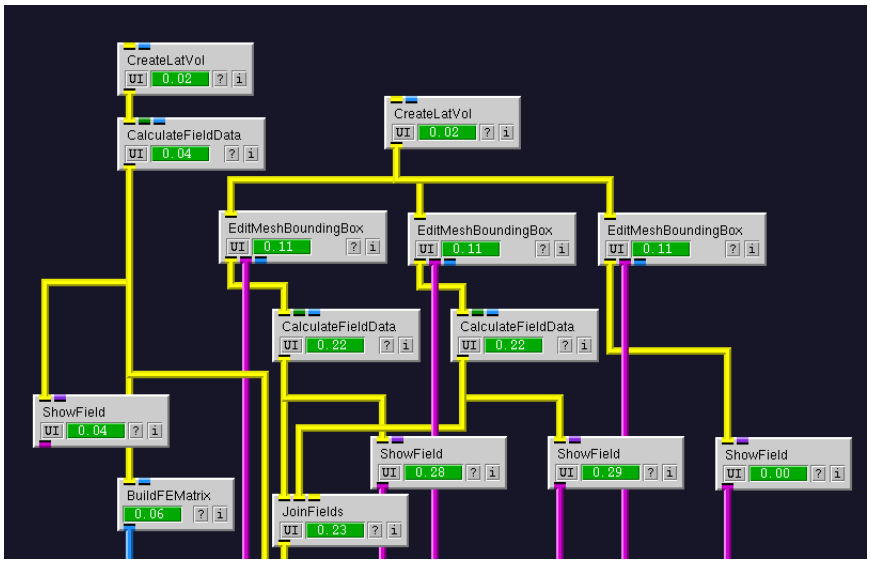

Figure 3.13. Adding a floating lead to the problem geometry.

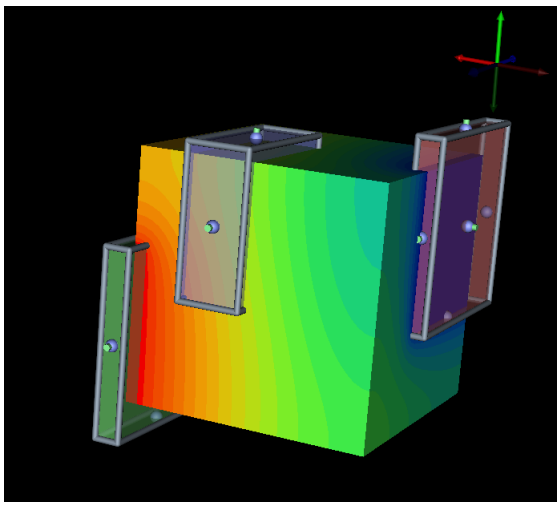

Figure 3.14. Simulation with added floating lead geometry.

3.4 Adding a Floating Lead

Of interest to some, is the addition of a floating lead into the simulation, *i.e.* an extra conductor without a known potential. The procedure for doing so begins with adding a third **EditMeshBoundingBox** module connected to a **ShowField** module, changing the **ShowField** settings to enable transparency and setting the default color this time to something blue. Connect the pink outputs of both the **EditMeshBoundingBox** and **ShowField** modules to the **ViewScene** module. In the remainder of this tutorial, the third **EditMeshBoundingBox** was set to have a center value of (0.3, 0.5, -0.5) and a size of (0.5, 1.2, 1.2) in the user interface. The modified network is shown in Figure 3.13 and the resulting visualization is shown in Figure 3.14. Note that the addition of this geometry has not affected the solution.

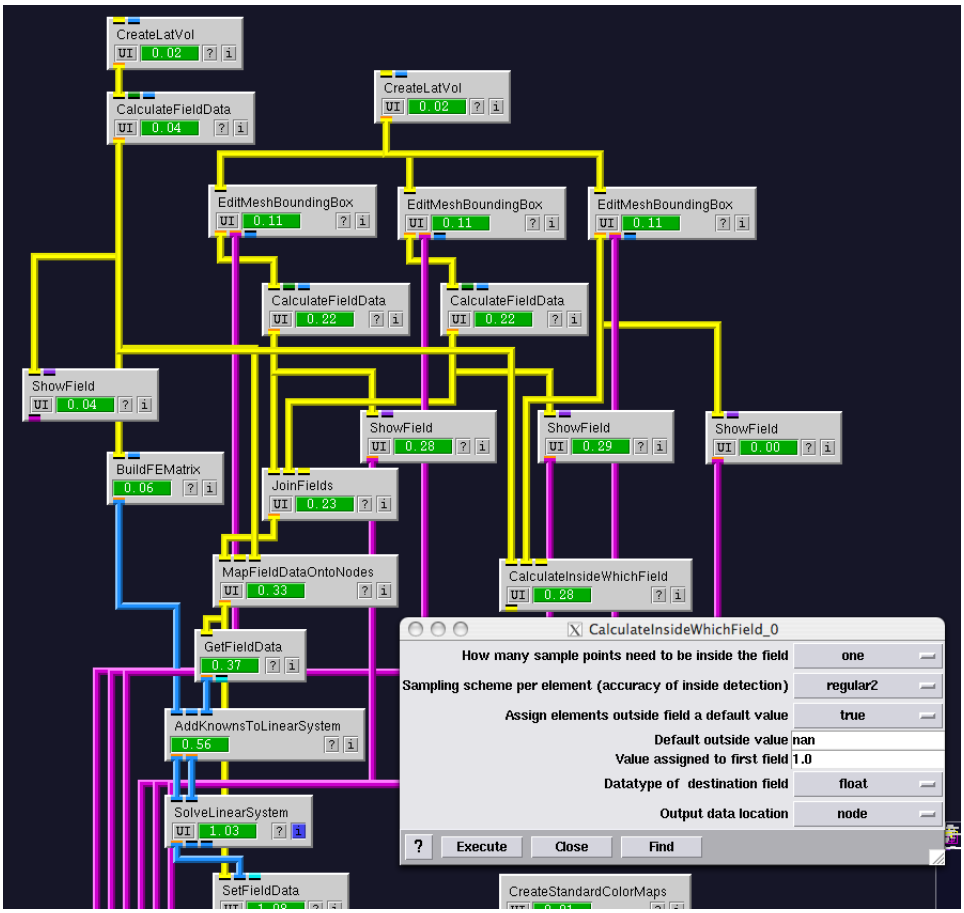

Figure 3.15. Network and user interface settings when adding the **CalculateInsideWhichField** module.

Next, we need to know which of the nodes are inside this new conductor. To calculate this, add a **CalculateInsideWhichField** module to the network, with the first input coming from the first hex mesh **CalculateFieldData** output, and the second input coming from the third **EditMeshBoundingBox** output. We again want nodes outside of the new conductor to be set to NaN, and values inside set to 1.0. Open the user interface and set the "Default outside value" to `nan`, the "Value assigned to first field" to `1.0`, the "Datatype of destination field" to `float` and the "Output data location" to `node`. The addition of this module and the user interface settings are shown in Figure 3.15.

The effect of adding a conductor into the problem domain is that the solution will be constant for all nodes inside the conductor. For this to happen, we need to modify the linear system and "link" all the nodes inside the conductor to have the same solution value. To do this, connect a **GetFieldData** module to the output of the **CalculateInsideWhichField** module. Next, delete the connections between **AddKnownsToLinearSystem** and **SolveLinearSystem** and add a **AddLinkedNodesToLinearSystem** module between these two, connecting the first two outputs of **AddKnownsToLinearSystem** to the first two inputs of **AddLinkedNodesToLinearSystem**. Connect **AddLinkedNodesToLinearSystem** and **SolveLinearSystem** in the same way. The third input for the **AddLinkedNodesToLinearSystem** module comes from the output of the new **GetFieldData** module. Next,

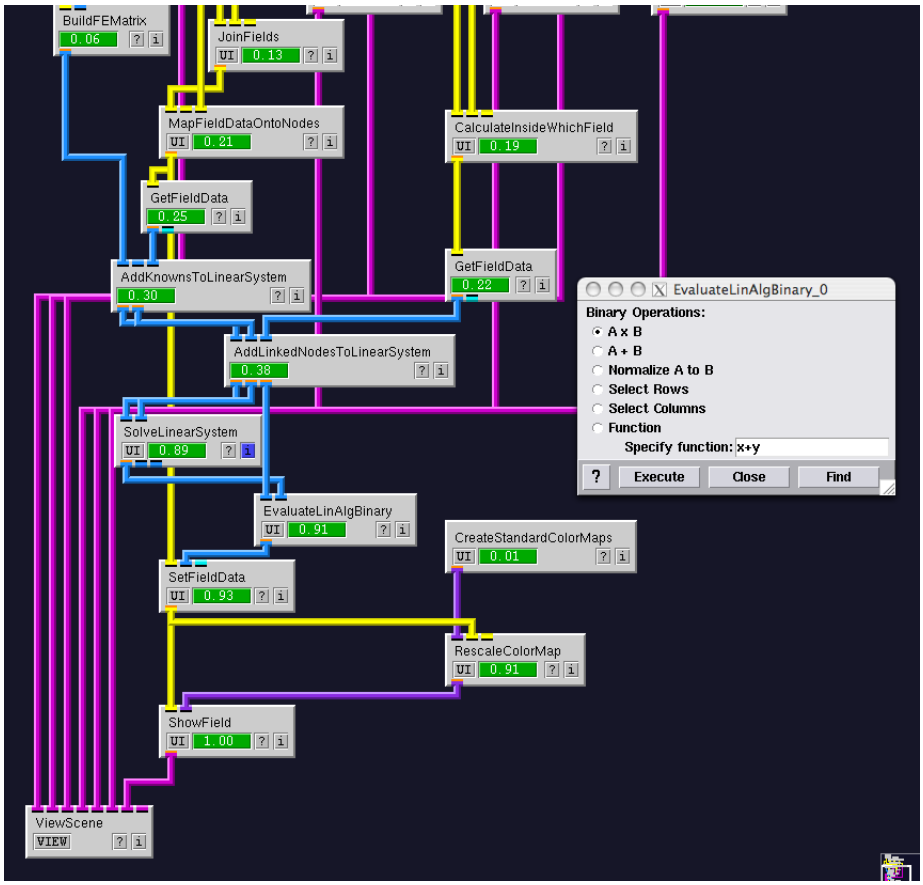

Figure 3.16. Completing the network with an added floating lead.

add an **EvaluateLinAlgBinary** module with the first input coming from the third output of the **AddLinkedNodesToLinearSystem** module. Delete the output of **SolveLinearSystem** and connect it instead to the second input of **EvaluateLinAlgBinary**. The output from **EvaluateLinAlgBinary** is now the second input for **SetFieldData**. This completes the modifications to the simulation. The final modified network is shown in Figure 3.16 and the simulation results are shown in Figure 3.17.

The fact that the solution field is constant inside the floating lead is difficult to see when the floating lead is visible. Figure 3.18 shows a final visualization where the "Show Faces" checkbox inside of the **ShowField** module corresponding the the floating lead is unchecked. Now, the constant field is easy to discern.

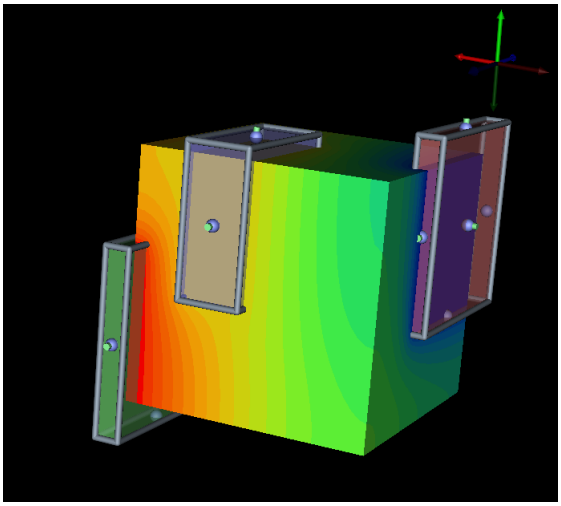

Figure 3.17. Simulation results with an added floating lead.

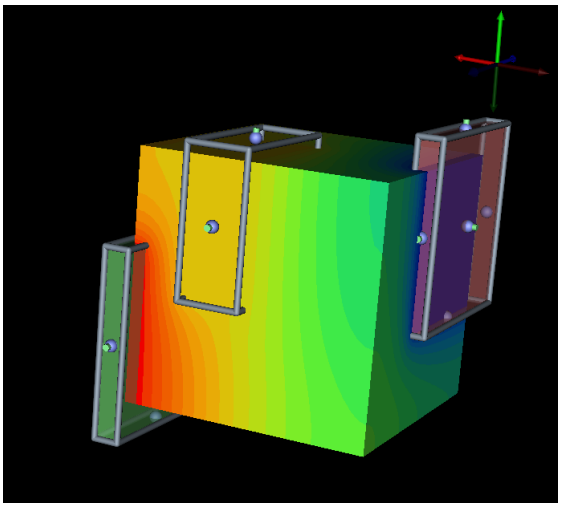

Figure 3.18. Simulation results with an added floating lead with the visualization of the floating lead's faces turned off.

Chapter 4

Placing Electrodes

In the previous chapter, we learned how to run a finite element simulation on a cube with two electrodes and a floating lead. In the remaining sections, we will perform a similar calculation on a human torso using previously segmented torso data, a model of a can electrode, a wire electrode, and a planar electrode.

4.1 Loading The Dataset

Start SCIRun and insert a **ReadField** module into the network. Open the user interface, select "SCIRun Field File (*.fld)" from the "Files of type" drop-down menu, browse to and select the `torso-defib/torso_segmentation_si.fld` file. Unlike the cube above, we are interested in the internal structure of this file, so we will use texture slices to view the data.

Connect the output of **ReadField** to a **ConvertFieldsToTexture** module and connect that module to a **ShowTextureSlices** module. Open the user interface to **ShowTextureSlices** and check the "X plane", "Y plane", and "Z plane" boxes. Place a **CreateStandardColorMaps** module in the network, and connect its output to the second input of the **ShowTextureSlices** module. Open the **CreateStandardColorMaps** user interface and click the mouse in the upper display to adjust alpha. Create one point in the lower left corner (this will make the area outside the torso black). Create another point back near the middle of the display, slightly to the right of the first point. Finally, connect the **ShowTextureSlices** output to a new **ViewScene** module. The network should look similar to that shown in Figure 4.1 and the display output is shown in Figure 4.2.

4.2 Visualizing Extra Geometry

The visualization in Section 4.1 shows texture slices through the entire segmented torso. One may also be interested in adding additional 3D visualizations of subsets of the segmentation to aid in the placement of electrodes. Next, we will add a visualization of the boundary of the heart as an example.

Add the following module chain to the network: Connect the **ReadField** module to a **GetDomainBoundary** module, followed by a **FairMesh** module, and lastly a **ShowField** module. The output of **ShowField** should be connected as the second input to the **ViewScene** module.

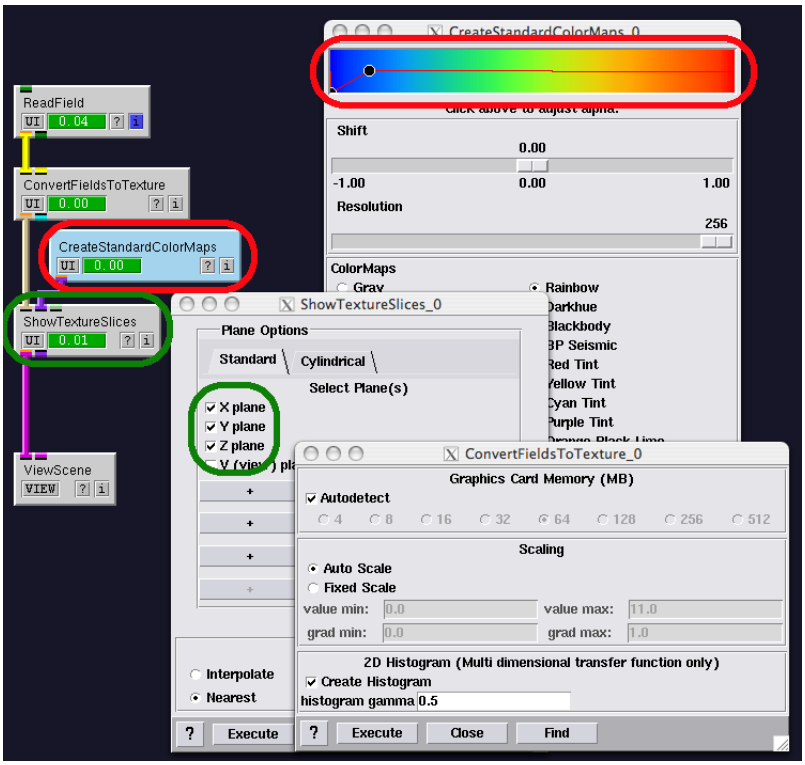

Figure 4.1. Start of a network to place electrodes in torso.

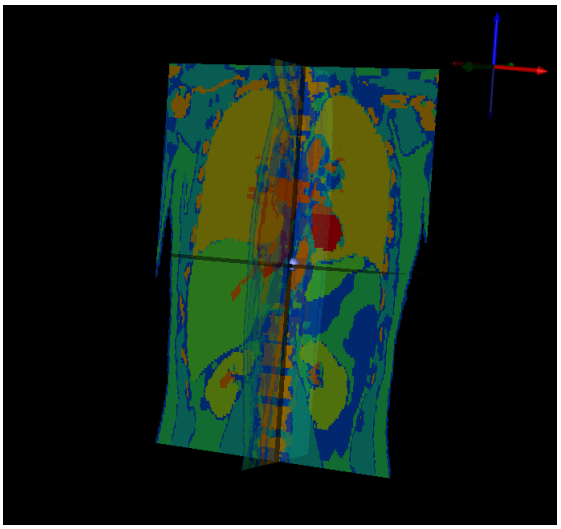

Figure 4.2. Output of initial torso visualization.

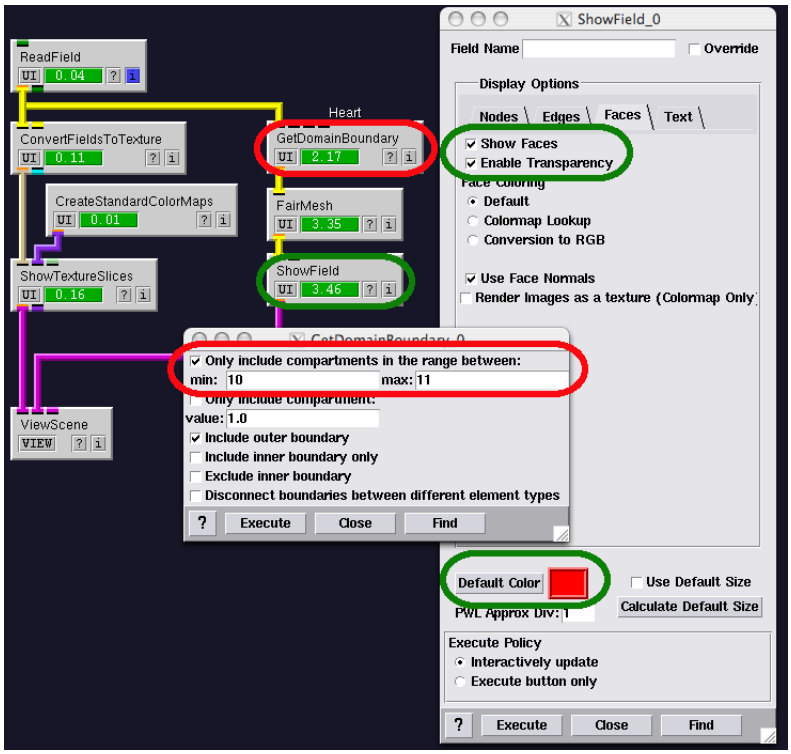

Figure 4.3. Network after adding visualization of the heart.

Open the user interface to the **GetDomainBoundary** module, make sure the "Only include compartments in the range between" checkbox is checked, and put values of `10` and `11` for the "min:" and "max:" fields. Also, make sure "Include outer boundary" is checked. Next, open the user interface to the **ShowField** module. As before, Disable the viewing of the nodes and edges, and enable transparency for the faces. Set the "Face Coloring" to "Default" and select a red default color. The new network should look like Figure 4.3 and the resulting visualization is shown in 4.4.

Next, a visualization of the outer boundary of the dataset may be useful. Add a similar module chain as for the heart, but this time in the new **GetDomainBoundary** module, only include compartments in the range between `1` and `255`. This will select all data. Also, we are only interested in the outer boundary of this data set, so make sure both "Include outer boundary" and "Exclude inner boundary" are both checked. The network and results are shown in Figures 4.5 and 4.6.

Other subsets of the segmentation can also be added to the visualization. Table 4.1 lists the segmentation indices in the torso dataset.

4.3 Adding a Can Electrode

Next, we will read in a model of a can electrode and send the model through a widget that will allow interactive editing of the electrodes' placement and size. Begin by adding another **ReadField** module to the network. Open the user interface, select "SCIRun Field File (*.fld)" from the "Files of type" drop-down menu, browse to and select the

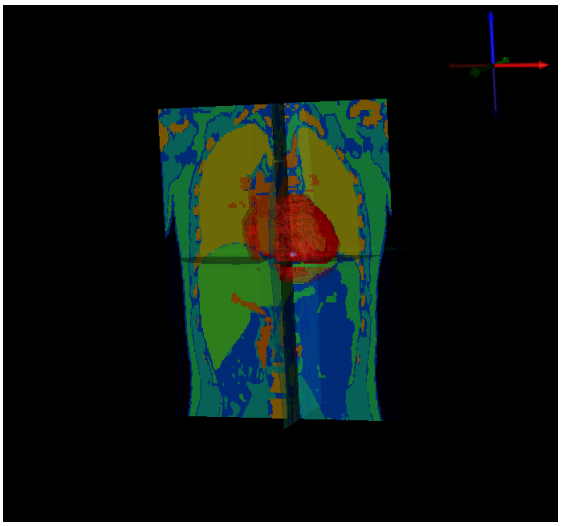

Figure 4.4. Visualization results, including heart.

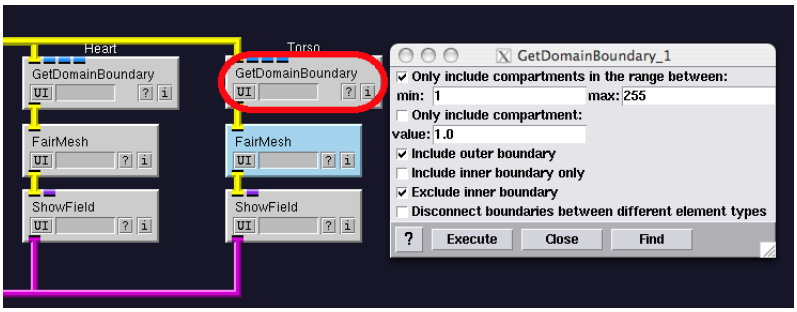

Figure 4.5. Addition of outer torso boundary visualization to the network.

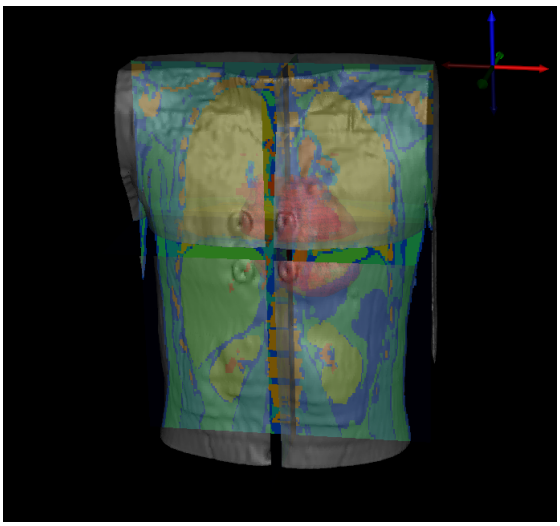

Figure 4.6. Visualization results, including heart and torso.

Material	Seg. Indx
Background	0
Connective Tissue	1
Bowel Gas	2
Muscle	3
Fat	4
Kidney	5
Liver	6
Lung	7
Bone	8
Blood	9
Heart-Atria	10
Heart-Ventricles	11

Table 4.1. Table of segmentation indices in the torso dataset.

`torso-defib/electrode_can_model_si.fld` file. Connect the output of **ReadField** to an **EditMeshBoundingBox** module. Connect the pink output of **EditMeshBoundingBox** directly to the **ViewScene** module - this will allow the visualization of the widget. Connect the yellow output to a new **ShowField** which is also connected to **ViewScene**. Open the **ShowField** user interface, disable the viewing of nodes and edges, and select the default color to be something green.

Similar to the placing of electrodes in the cube mesh previously, upon viewing the scene the location and size of the electrode can be modified through interaction with the **EditMeshBoundingBox** widget. Holding the shift key and grabbing the various controls (edges, spheres, and cylinders) will modify the position, rotation, and scaling of the electrode. Figure 4.7 shows this modified network and Figure 4.8 shows the placement used in the remainder of this tutorial.

4.4 Adding a Wire Electrode

A second type of electrode often used in defibrillation scenarios is a wire. In this section, we will create a small cylinder which represents the non-insulated section of a wire electrode lead. SCIRun provides a widget to generate and manipulate a wire electrode. Add a **GenerateWireElectrode** module to the network, connecting the pink output port to the **ViewScene** module and the yellow output port to a new **ShowField** module. Again, connect the **ShowField** module to the **ViewScene** module. Open the **GenerateWireElectrode** module and to start, set the length of the electrode to `0.1` (corresponding to 10 centimeters). Set the width of the electrode to `0.003`. Open the **ShowField** module, turn off visualization of the nodes and edges, and set the default color to something distinctive. Figure 4.7 shows an example of this modified network.

Upon viewing the scene, a wire electrode will now be present. On the wire electrode, 5 control points will be visible. Moving the control points changes the shape of the electrode in a fairly intuitive manner. Note that the electrode does not necessarily travel through the points, they are only used as control points for a spline function. Also note that the electrode will always start at one of the points, but will not necessarily end at the last point.

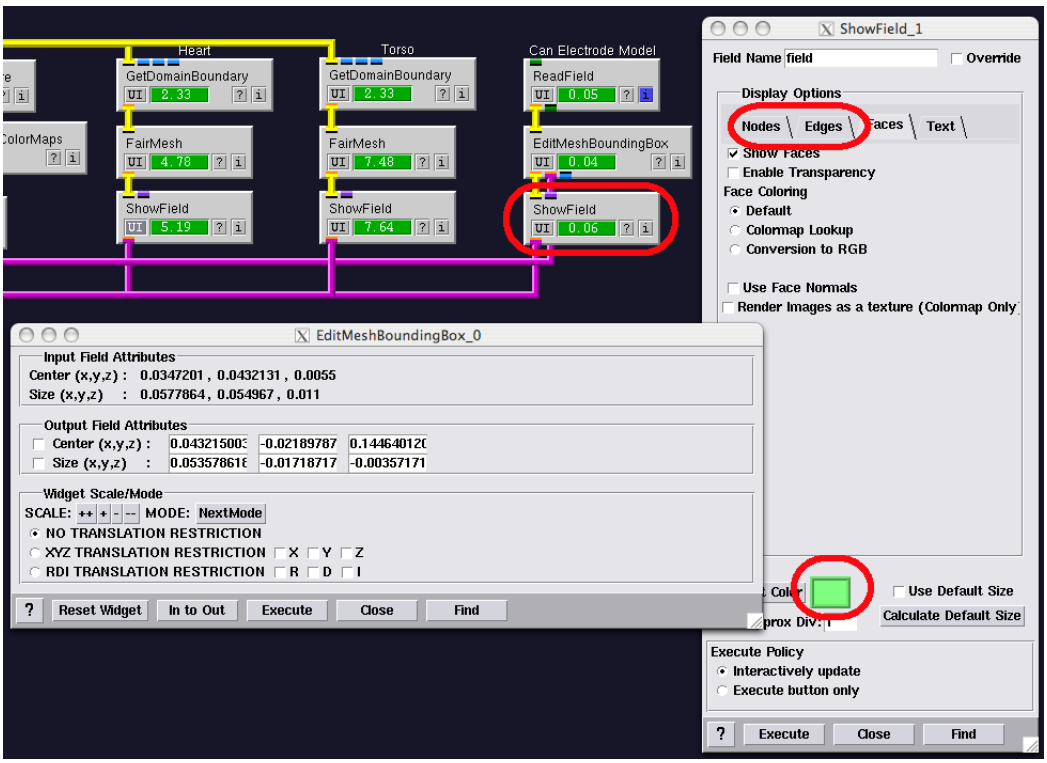

Figure 4.7. Addition of a can electrode to the network.

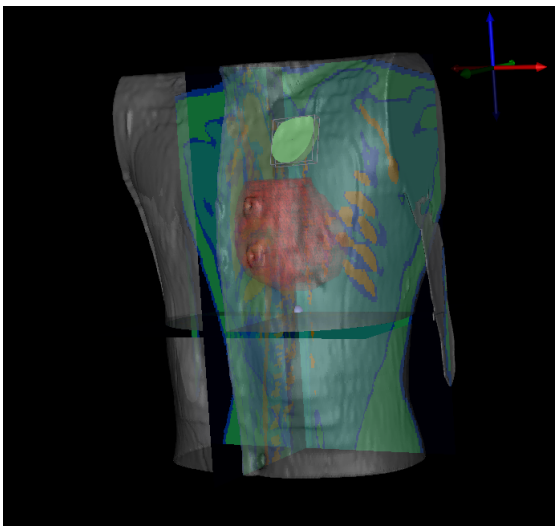

Figure 4.8. Placement of the can electrode.

Figure 4.9. Addition of a wire electrode.

The length is determined by the length value in the user interface, not the position of the last point. Figure 4.10 shows an example of the electrode placement used in the remainder of this tutorial.

4.5 Adding a Planar Electrode

Lastly, we will add a planar electrode. The creation of the planar electrode is similar to the wire electrode, with the exception that the planar electrode has a length, width, thickness, and normal vector rather than just a length and width. Similar to the wire electrode, add a **GeneratePlanarElectrode** module, connected to a **ShowField** module, both connected to the **ViewScene** module. Open the **GeneratePlanarElectrode** user interface and set the length of the electrode to `0.1`, thickness to `0.003`, and width to `0.03`. In the **ShowField** module, disable the viewing of the nodes and edges, and select a distinctive default color. Figure 4.11 shows an example of this modified network and Figure 4.12 shows an example of the electrode placement used in the remainder of this tutorial.

4.6 Writing Electrodes to a Bundle

One feature of SCIRun is the ability to "bundle" various fields together. We will take advantage of that feature when writing the electrode fields to a file. Add a **InsertFieldsIntoBundle** module into the network and connect the yellow output of the can electrode's **EditMeshBoundingBox** module to the first input. Connect the outputs of the **Gen-**

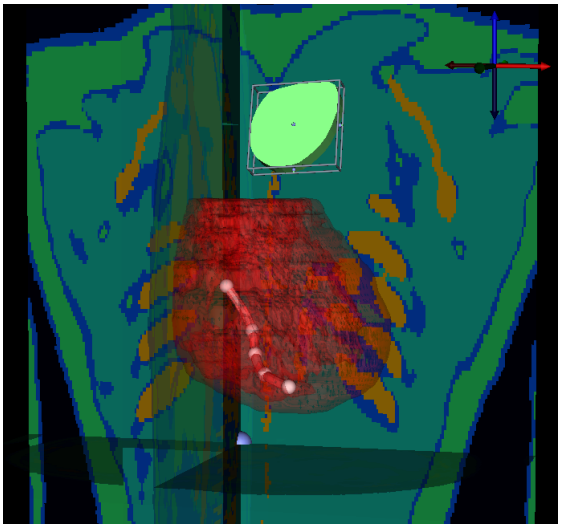

Figure 4.10. Placement of the wire electrode.

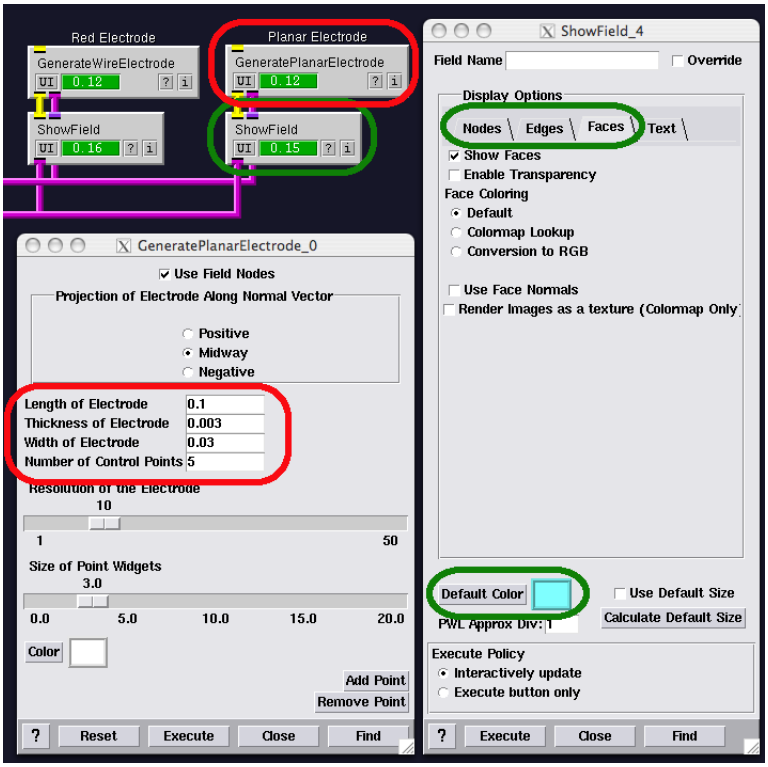

Figure 4.11. Addition of a planar electrode.

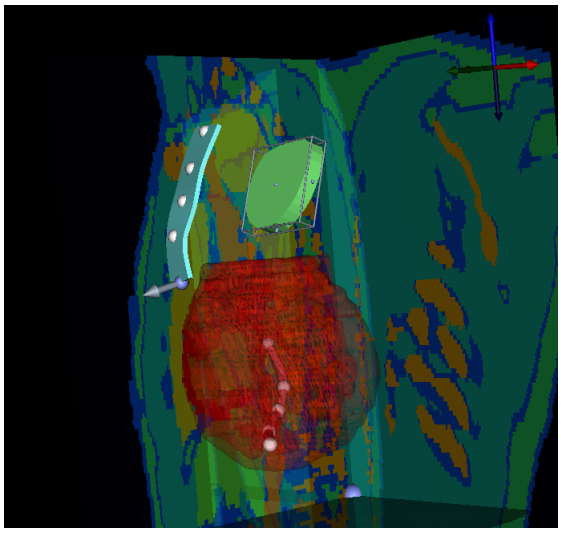

Figure 4.12. Placement of the planar electrode.

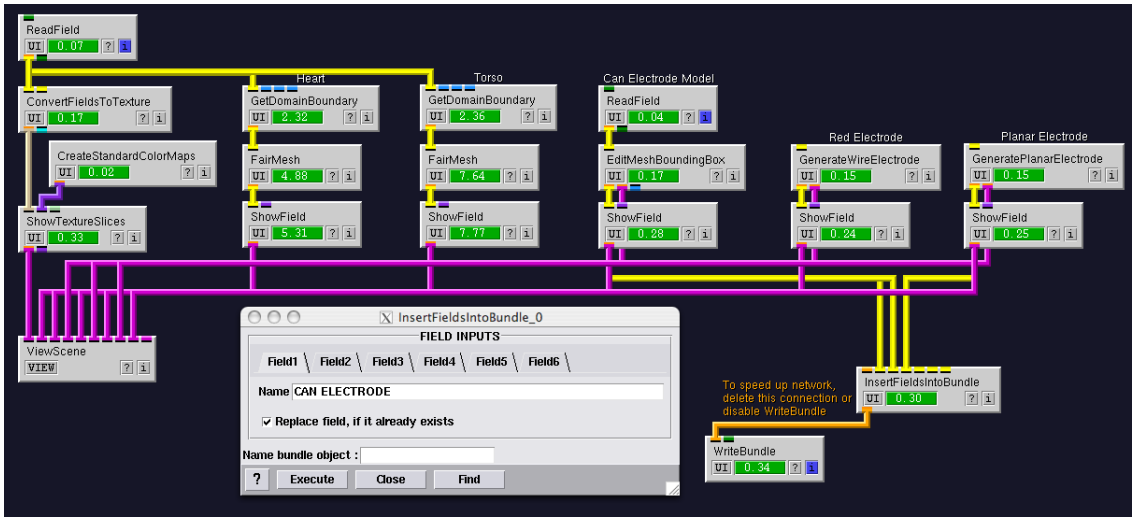

Figure 4.13. Complete network for generating and placing electrodes.

erateWireElectrode and **GeneratePlanarElectrode** modules to the second and third inputs of **InsertFieldsIntoBundle**. Open the user interface and click on the "Field 1" tab. In the "Name" textbox, type a descriptive name for the first field, such as `CAN ELECTRODE`. Click on the "Field 2" tab and provide a name, such as `WIRE ELECTRODE`. Click on the "Field 3" tab and provide a name, such as `PLANAR ELECTRODE`.

Next, connect the orange output of **InsertFieldsIntoBundle** to a new **WriteBundle** module. Open the user interface, browse for a directory, and type a name for the electrode field bundle save file. Executing that module will cause the file to be written. Figure 4.13 shows the complete network.

Note that if the **WriteBundle** module is connected and enabled, each time the network is run, the file will be overwritten. Thus, if you move one of the electrode in the view window, the old file will be automatically overwritten with the new file. To keep this from

happening, disconnect the **WriteBundle** module until you are ready to write the file, or right click on the module and select "Disable" from the dropdown menu.

Chapter 5

Finite Element Simulation on a Torso

In the previous chapter, we created a network which allows the placement of various electrodes within a torso. The result of that network was a SCIRun bundle being written to a file, consisting of three fields: a can electrode, a wire electrode, and a planar electrode. In this chapter, we will read the torso and the electrodes back in, assign known conductivities to various materials in the torso segmentation, assign known potentials to two of the electrodes, allow the third electrode to be a floating lead, create a finite element mesh, refine that mesh near the electrodes, and solve the electric field problem.

5.1 Building and Viewing the Finite Element Mesh

To simplify processing and reduce runtimes, we will create a smaller Hexahedral Mesh on which the finite element simulation will be run. To do this, start a new network and insert a **ReadField** module. Open the module, select "SCIRun Field File (*.fld)" in the "Fields of type" dropdown box, browse for and select the `segmentation_si.fld` file. Connect the output to a **CreateLatVol** module. In **CreateLatVol**, set the "X Size", "Y Size", and "Z Size" text fields to `50`, `50`, and `75`. Insert a **MapFieldDataOntoElems** module into the network, connecting the output of **ReadField** to the first input and the output of **CreateLatVol** to the third input. This will map the data values in the segmentation onto the nodes of the hex mesh. Open the **MapFieldDataOntoElems** module and select "mostcommon" in the "SampleMethodPerElement" dropdown box. This will ensure the values on the nodes will be one of the segmentation values and not an interpolated value. Figure 5.1 shows this new network.

To visualize the segmentation, first we will clip the data outside the actual torso (outside data has a value of `0`) by connecting a **ClipFieldByFunction** module to the output of **MapFieldDataOntoElems**. Open the user interface and set the expression to `RESULT=DATA>0;`. Next, connect the output of **MapFieldDataOntoElems** to the following module chain: **GetFieldBoudnary**, **FairMesh**, **ShowField**, and **ViewScene**. As we did in the entire last chapter, open the **ShowField** module and disable rendering of the nodes and edges. Figure 5.2 shows this new network and the resulting visualization.

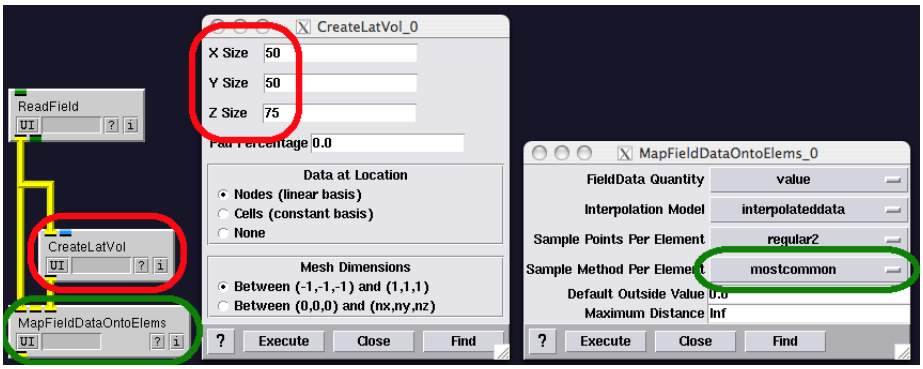

Figure 5.1. Beginnings of the defibrillation finite element simulation network.

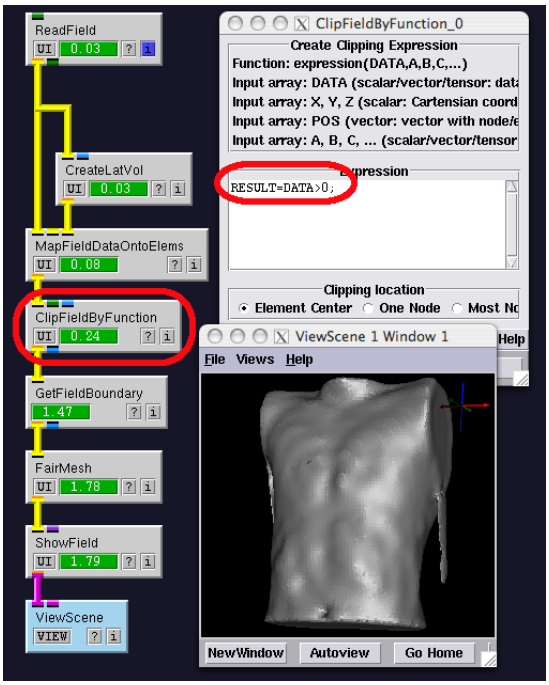

Figure 5.2. Viewing the finite element torso mesh.

5.2 Completing an Initial Two-Electrode Simulation

Now that our simulation mesh has been created, we need to build our finite element matrix, add our known electric potentials in the locations of the electrodes, and solve the finite element linear system. We will start by specifying the conductivities of various materials in the segmentation. To do so, insert a **CreateMatrix** module into the network. Open the user interface and set the number of rows to `12` and the number of columns to `1` by typing these values into the text boxes at the top. A matrix with 12 entries (0 through 11) will appear. Table 5.1 lists the conductivities used in this simulation. Populate the matrix with these, or modified values.

Connect the yellow output of **ClipFieldByFunction** to the input of a **ConvertIndicesToFieldData** module. Connect the output of **CreateMatrix** to the second input. Connect

Material	Seg. Indx	Conductivity (S/m)
Background	0	0.0
Connective Tissue	1	0.22
Bowel Gas	2	0.002
Muscle	3	0.25
Fat	4	0.05
Kidney	5	0.15
Liver	6	0.07
Lung	7	0.007
Bone	8	0.006
Blood	9	0.7
Heart-Atria	10	0.3
Heart-Ventricles	11	0.3

Table 5.1. Conductivity values used in this simulation.

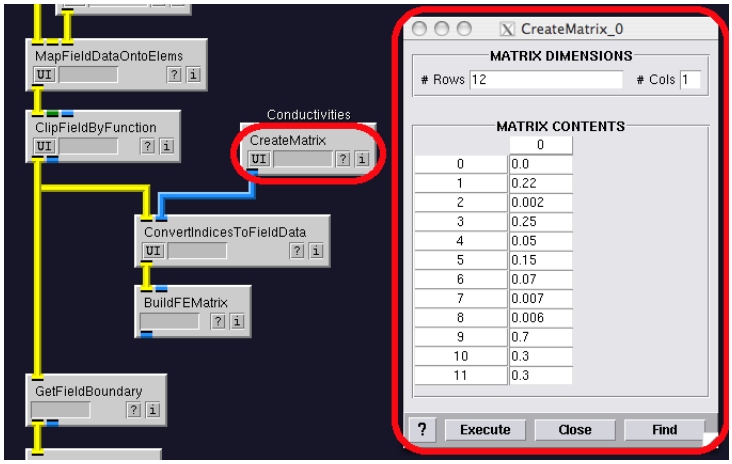

Figure 5.3. Network now incorporating conductivities and the finite element matrix.

the output of **ConvertIndicesToFieldData** to a **BuildFEMatrix** module. The network should look similar to Figure 5.3.

Next, the electrodes need to be incorporated into the simulation. Insert a **ReadBundle** module into the network. Open and select the electrode bundle written previously to a file in Section 4.6. Connect the output of **ReadBundle** to a **GetFieldsFromBundle** module. For the user interface of **GetFieldsFromBundle** to work correctly, the **ReadBundle** module needs to first be executed. Right click on **ReadBundle** and select "Execute" from the pop-up menu. Now, open the **GetFieldsFromBundle** user interface. We will need to specify which fields in the bundle will correspond to the output ports on the module. Make sure the "Field1" tab is selected, and click on `CAN ELECTRODE` in the selection list (or whatever the can electrode field was named previously. Open the "Field2" tab and select `WIRE ELECTRODE`. Open the "Field3" and select `PLATE ELECTRODE`. This will associate the can, wire, and plate electrodes with the first, second, and third yellow output ports. Figure 5.4 shows the addition of these modules to the network.

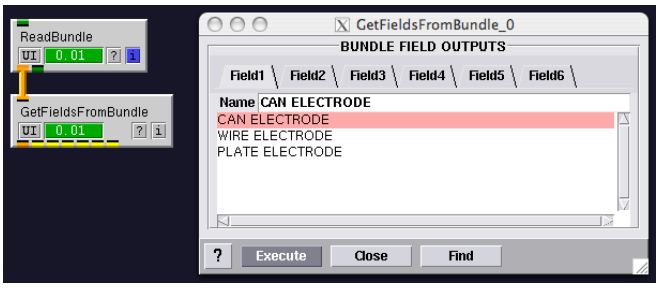

Figure 5.4. Reading in the electrode configuration bundle.

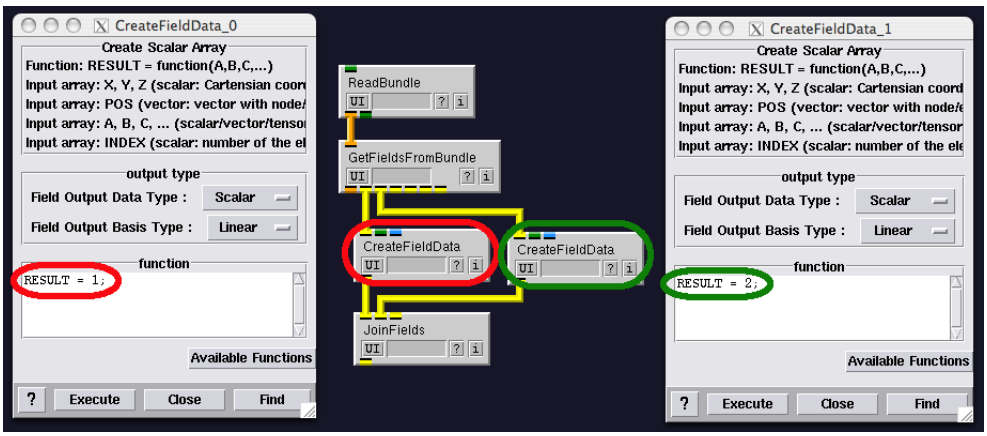

Figure 5.5. Specifying field labelings for the two electrode fields.

We will use a slightly different technique for associating known potentials to the electrodes than we did with the box simulation. To start, connect the first and second yellow outputs of **GetFieldsFromBundle** to separate **CreateFieldData** modules. Connect those modules to a **JoinFields** module. In the first **CreateFieldData** module, set the function to `RESULT = 1;`. In the second, set the function to `RESULT = 2;`. These network additions are shown in Figure 5.5.

Next, we will map these indices onto our mesh, giving us a mesh with values of `1` where at the can electrode, `2` at the wire electrode, and `0` everywhere else. We accomplish this by connecting the **JoinFields** module to the first input of a **MapFieldDataOntoNodes** module. Connect the output of **ClipFieldByFunction** to the third input of the new **MapFieldDataOntoNodes** module. Similar to how we mapped conductivities onto the mesh, we will map the known potentials onto the mesh using a new **ConvertIndicesToFieldData**, which receives its input from the **MapFieldDataOntoNodes** module and a new **CreateMatrix** module. Open the **CreateMatrix** user interface, and set the number of rows and number of columns to `4` and `1`.

Remember from the box tutorial that a value of NaN in our "known values" vector signifies an unknown value. Therefore, we set the first entry of our matrix to a value of `nan`. The next two entries represent the potentials of the two electrodes. We will set these to `450` and `0` for this simulation. Finally, for reasons that will become clear once we add a floating electrode, set the fourth entry to `nan`. The current state of the network is shown in Figure 5.6.

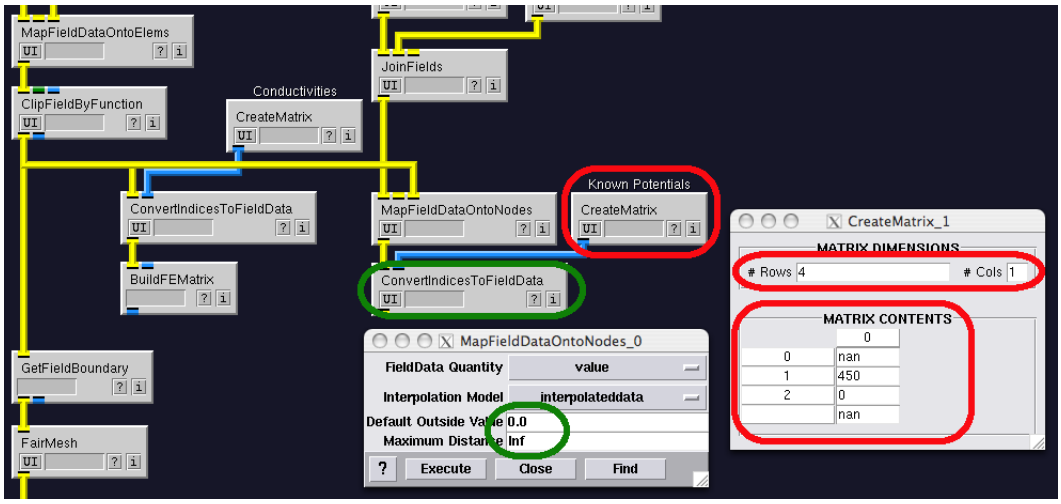

Figure 5.6. Associating known potential values with the two electrodes.

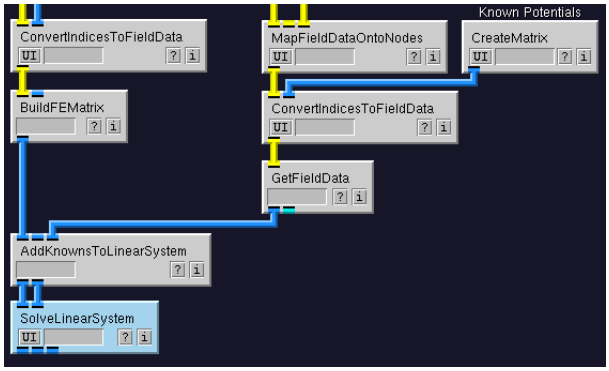

Figure 5.7. Solving the linear system.

The values on the mesh now represent the "known" values of our solution. All that remains to complete our first simulation is to get a vector of these known values and add them as known values to our linear system. To do this, connect the **ConvertIndicesToFieldData** module to a **GetFieldData** module. Add a **AddKnownsToLinearSystem** module with the first input coming from **BuildFEMatrix** and the third input coming from the **GetFieldData** module. Connect the two outputs of **AddKnownsToLinearSystem** to the two inputs of a **SolveLinearSystem** module. These modifications are shown in Figure 5.7.

To visualize the results at this point, insert a **SetFieldData** module between the current **ClipFieldByFunction** and **GetFieldBoundary** modules. This requires deleting the previous connection, adding the module, and connecting the three modules together. We will add a standard color map by adding a **CreateStandardColorMaps** module connected to a **RescaleColorMap** module. The second input of the **RescaleColorMap** module comes from the output of the **SetFieldData** module. The output of **RescaleColorMap** is used as the second input for the **ShowField** module. Open the **ShowField** module and make sure the "Face Coloring" on the "Faces" tab is now set to "Colormap Lookup". These changes and the resulting visualization are shown in Figure 5.8.

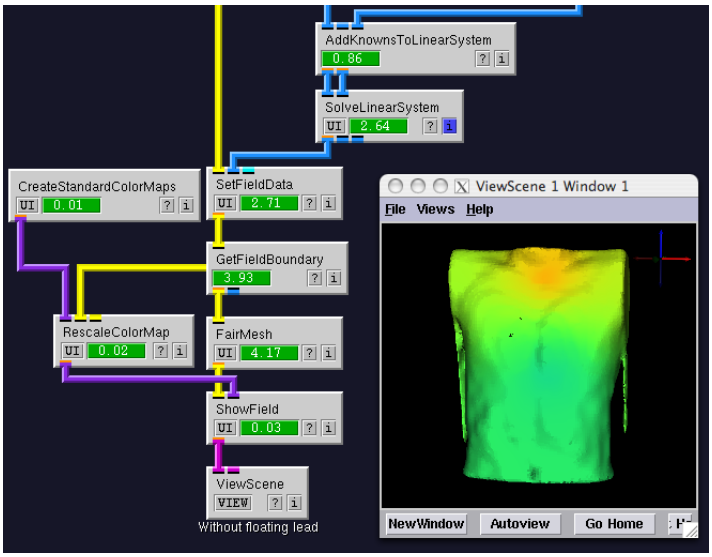

Figure 5.8. Initial simulation results for the two electrode problem.

5.3 Refining the Mesh

Depending on the resolution of the mesh generated in the **CreateLatVol** module, and depending on the size of the electrodes used in the simulation, the electrodes may be much smaller than can be resolved on our simulation mesh. Our first modification to the simulation in the previous section is to perform automatic refining of the mesh close to the electrodes.

Start by deleting the connection between **CreateLatVol** and **MapFieldDataOntoElems**. Connect the output of **CreateLatVol** to a **CalculateDistanceToField** module. Connect the output of **JoinFields** to the second input. Next, we will map this data back onto the the simulation mesh by connecting the output of **CalculateDistanceToField** to a **MapFieldDataOntoElems** module. The third input of the new **MapFieldDataOntoElems** comes directly from **CreateLatVol**. This distance information is now used to drive the refinement of the mesh. Elements with distances close to the electrodes will be refined while elements further away will be left as is. This refinement is accomplished by connecting the output of **MapFieldDataOntoElems** to a **RefineMesh** module. Lastly, connect the output of **RefineMesh** to the third input of our original **MapFieldDataOntoElems** module (which was partially disconnected at the beginning of this procedure).

The user interface settings are as follows: Open the **CalculateDistanceToField** module and check the "Truncate distance larger than:" checkbox. Insert a value of `0.02` in the text box. Open the newest **MapFieldDataOntoElems** module, and select "max" from the "Sample Method Per Element" dropdown menu. Open the **RefineMesh** module, select the "Do not refine nodes/elements with values greater than isovalue" radio button, and enter an IsoValue of `0.02` in the text box. Figure 5.9 shows these additions to the network while Figure 5.10 shows the simulation results.

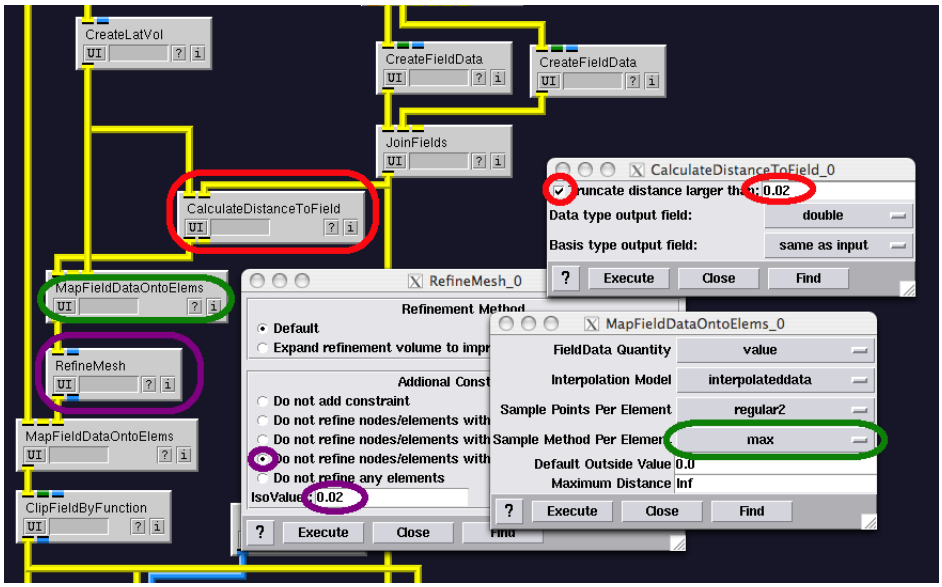

Figure 5.9. Adding functionality to refine the mesh near the electrodes.

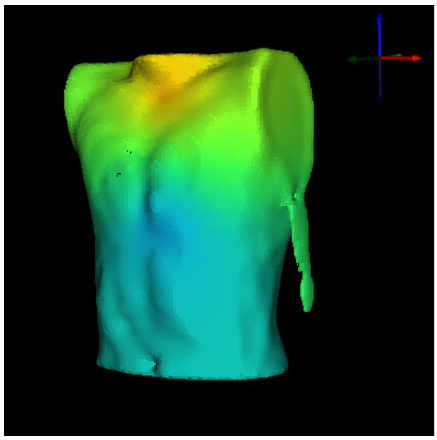

Figure 5.10. Simulation results after mesh refinement.

5.4 Adding a Floating Lead

As in the box simulation, we may wish to add a floating lead, or a conductor, to our simulation. We will use the plate electrode saved in our input bundle as this floating lead. The **JoinFields** module requires all input fields to be of the same type, however the plate electrode is saved as a hex mesh, while the can and wire electrodes are saved as tet meshes. Therefore, our first step will be to connect the third yellow output of the **GetFieldsFromBundle** module to a **ConvertHexVolToTetVol** module. Now, as before, connect that output to a **CreateFieldData** module (with equation set to `RESULT = 3;`), and connect that output to the **JoinFields** module. Remember that our second **CreateMatrix** containing known potentials contains and extra value of `nan` for the fourth entry. Therefore, the plate electrode will still be considered an unknown value. The remainder of the addition of the floating electrode is very similar to the box example in Chapter 3.

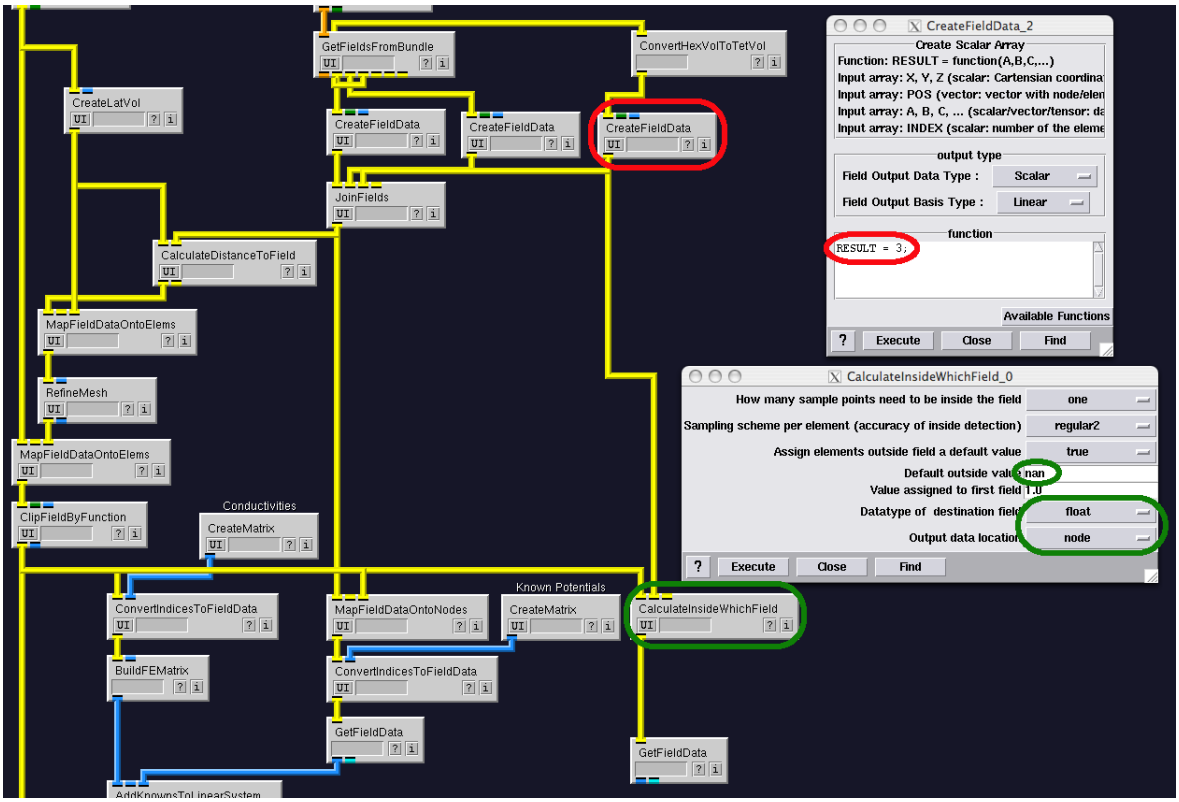

Figure 5.11. Adding a floating electrode to the simulation.

Insert a new **CalculateInsideWhichField** module, with the first input coming from the **ClipFieldByFunction** module, and the second input coming from the **CreateFieldData** module associated with the plate electrode. Open the user interface and set the "Default outside value" to `nan`. Set the "Datatype of destination field" to "float" and set the "Output data location" to "node". Connect the output to a **GetFieldData** module. The current state of the network is shown in Figure 5.11.

We will now create a second simulation pathway instead of modifying the first simulation which will help show the differences between the simulation with and without the floating electrode. Connect the two outputs of **AddKnownsToLinearSystem** to the first two inputs of **AddLinkedNodesToLinearSystem**. The third input comes from the latest **GetFieldData** module. Connect the first two outputs of **AddLinkedNodesToLinearSystem** to a **SolveLinearSystem** module. Connect the third output of **AddLinkedNodesToLinearSystem** to the first input of a **EvaluateLinAlgBinary** module. The second input of the **EvaluateLinAlgBinary** comes from the first output of **SolveLinearSystem**. Create a new **SetFieldData** module, the first input coming from the **ClipFieldByFunction** module, and the second input coming from the **EvaluateLinAlgBinary** module. Duplicate the remainder of the first simulation pathway after the **SetFieldData** module (starting with the **GetFieldBoundary** module. These modifications are shown in Figure 5.12 with the simulation results shown in Figure 5.13.

The differences between the two simulations in Figure 5.13 are admittedly difficult to

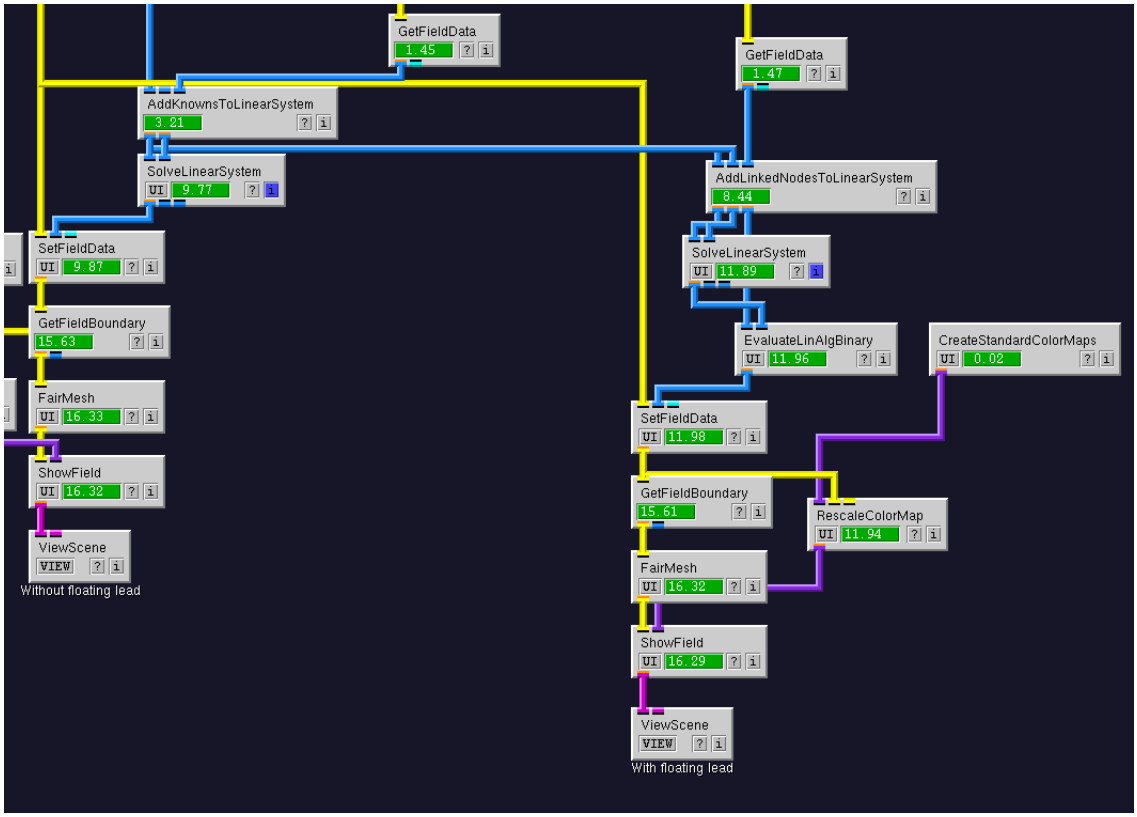

Figure 5.12. Second simulation pathway including a floating electrode.

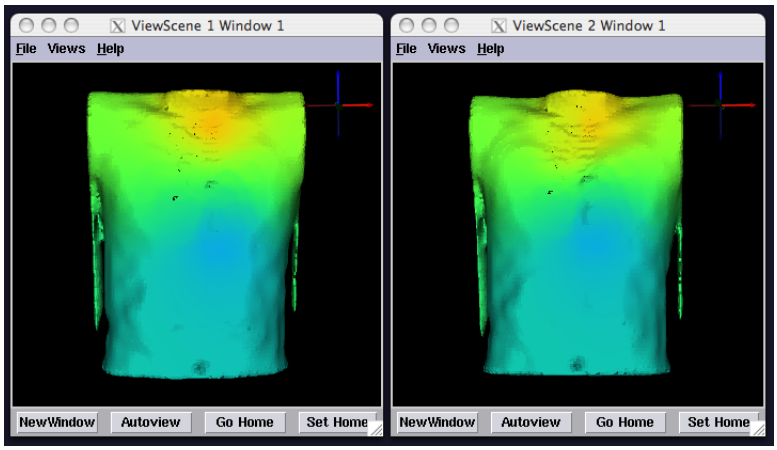

Figure 5.13. Results both with (right) and without (left) a floating electrode.

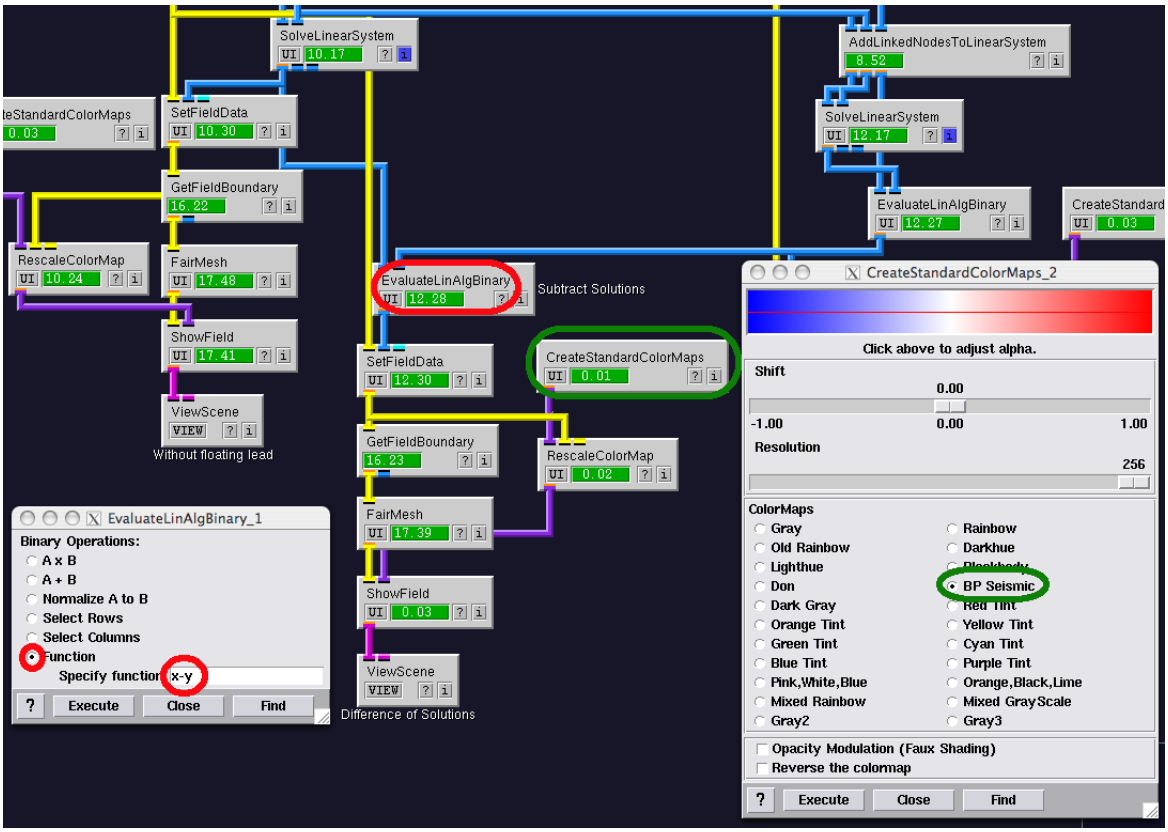

Figure 5.14. Adding a visualization pipeline to show differences between the solutions with and without a floating lead.

discern. Our last step, therefore, will be to create a visualization of the differences between these two simulations. Connect the outputs of the **SolveLinearSystem** (without the floating lead) and the **EvaluateLinAlgBinary** (with the floating lead) to another **EvaluateLinAlgBinary** module. Open the user interface and select the "Function" radio button, typing `x - y` into the "Specify function" text box. Create yet another **SetFieldData** module, with the first input again coming from the **ClipFieldByFunction** module and the second input coming from the newest **EvaluateLinAlgBinary** module. Duplicate the remaining visualization pipeline. This time, however, select a color map which deemphasizes values near zero. The "BP Seismic" colormap will do this. Figure 5.14 shows the final additions to the network while Figure 5.15 shows the differences between the two simulations.

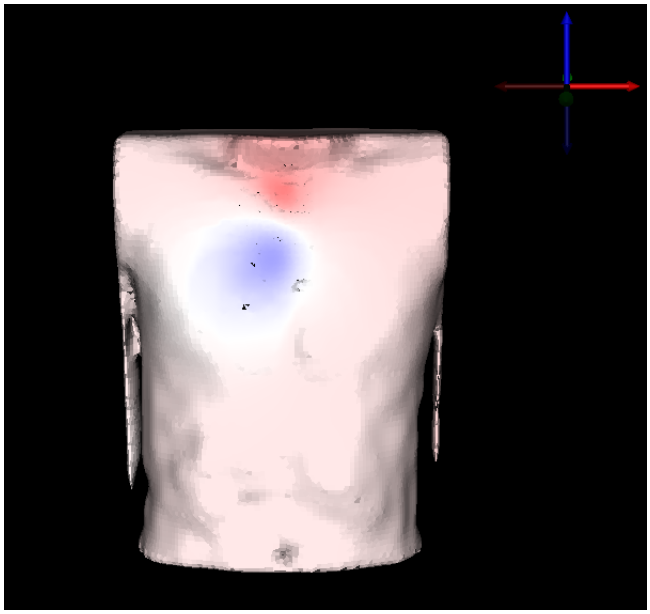

Figure 5.15. Results showing differences between two solutions–one with and one without a floating lead.