STUDYING RADIATION THERAPY TREATMENT PLANNING METHODS AND FORMULATIONS WITH TOLSTOY

Kyle M. Oliver
Department of Engineering Physics
University of Wisconsin-Madison
434 Engineering Research Building
1500 Engineering Drive
Madison, WI 53706
kmoliver@wisc.edu

ABSTRACT

TOLSTOY is a toy code for demonstrating important concepts in treatment planning for external beam x-ray radiation therapy. It is designed to strike a balance between authenticity and accessibility, helping users reach a conceptual understanding of how treatment planning codes work *without* forcing them to understand every detail of the radiation transport and optimization calculations. TOLSTOY uses Monte Carlo methods (MCNP) to calculate dose distributions for a given treatment geometry and given number of equally spaced treatment angles and then solves an optimization problem (with GAMS/CPLEX) to calculate a beam weight for each angle. This approach, known as conformal treatment planning, was chosen for ease-of-implementation and because it is relatively intuitive, in stark contrast to methods that use adjoint functions or intensity-modulated radiation sources. This paper briefly introduces the pedagogical motivation for developing this tool, describes its design and methodology, and presents results from an instructive test problem that approximates a prostate cancer irradiation.

Key Words: radiation therapy; conformal treatment planning; radiation transport; optimization; MCNP; GAMS/CPLEX; experiential learning

1 MOTIVATION

Radiation therapy treatment planning represents an interesting combination of challenges for the computational scientist. The associated radiation transport problems require sophisticated methods due to the geometric complexity of modeling the human body and to the exacting nature of clinical quality assurance measures. The powerful (and proprietary) nature of the optimization tools used to create treatment plans, to say nothing of the numerical methods that underlie these tools, has the potential to further mystify this active and fertile research area, which cannot be given a complete treatment below the advanced undergraduate level. The result for the would-be medical physicist is typical of an overall trend in STEM education classified by Brown, Luyendyk, and Ollis with a common remark from the archetypal student: "I didn't see what [my field] was all about until my final semester" (9 Brown, Ann 1997).

However, the underlying principles at work in the simpler dose optimization frameworks are well within the abilities of, say, a sophomore engineering or applied physics undergraduate. With some judicious "black boxing," these concepts can be taught with a simple toy program. This paper describes the design and the implementation of, and some sample results from, such a tool. Called TOLSTOY (**T**reatment **O**ptimization with **L**inear **S**coring **TOY**), it is designed to be a lightweight teaching and demonstration tool, not (as will be very obvious) a high-performance software package for clinical calculations. It attempts to strike a balance between authenticity and accessibility, giving students the chance to get a feel for how these calculations work without forcing them to first be able to derive every detail of the methods. This pedagogical approach is consistent with general frameworks for experiential learning (see, for instance, (Kolb, 1984)) and their various implementations (for instance, MIT's well-known Conceive – Design – Implement – Operate (Crawley, 2002)approach).(Brown, Luyendyk, & Ollis, 1997) One can envision a variety of meaningful class projects associated with TOLSTOY, including running problems, extending its capabilities, or improving its performance.

2 METHODS

There are a large and growing number of radiation therapy modalities, and even more methods for planning these treatments. One obvious approach for both internal (e.g., brachytherapy) and external (e.g., x-ray, electron, or hadron beam) radiation therapy planning is the use of adjoint functions(Lewis & Miller, 1993), which are especially helpful because they allow the specification of a desired detector response function. In this context, the detector response is the dose delivered to the tissues of interest. Many methods of brachytherapy (Yoo, Kowalok, Thomadsen, & Henderson, 2003; Yoo1, Kowalok, Thomadsen, & Henderson, 2007) and external beam (Wang, Goldstein, Xu, & Sahoo, 2005) planning have been implemented using adjoint techniques. However, the mathematical theory of adjoint functions is not especially intuitive and requires knowledge of advanced mathematics and transport theory to fully appreciate.

Much more accessible methodologically is the so-called conformal approach for external-beam therapy, which is described in some detail by Lim, et al. (Lim, Ferris, Wright, Shepard, & Earl, 2002). The basic idea is to calculate, via any appropriate transport method, the spatial dose distribution delivered to a patient by an x-ray beam incident at a number of different treatment angles. At each angle, the beam is shaped with a multi-leaf collimator (MLC) to achieve a "beam's eye view" of the treatment target. In other words, the x-ray source at each angle is a planar source whose boundary is the projection of the treatment target onto the plane perpendicular to the beam axis. Source particles originating outside of this projection have a better chance of depositing their energy in healthy tissue than in the target, so they should be excluded. A multi-leaf collimator's beam's eye view orientation for a given geometry and angle is shown in Figure 1.

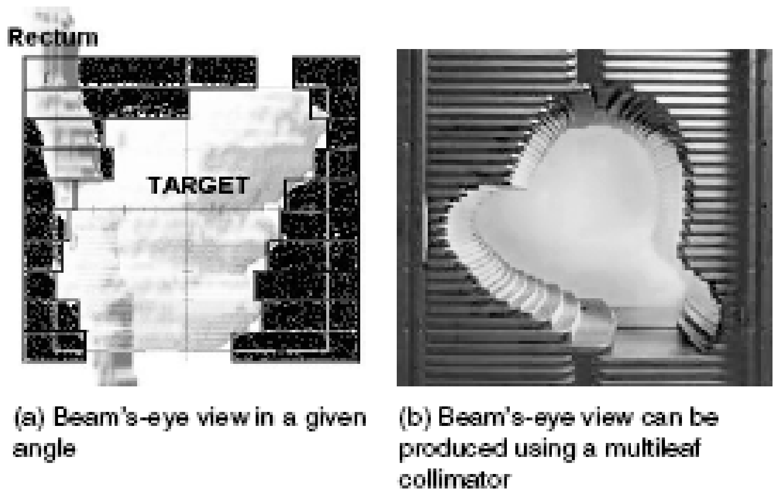

Rectum
TARGET
(a) Beam's-eye view in a given angle
(b) Beam's-eye view can be produced using a multileaf collimator

Figure 1: A multi-leaf collimator is used to create a "beam's eye view" of the treatment target at each angle. Image from (Lim et al., 2002).

After solving each angle-specific transport problem, one then solves an optimization problem to choose a weight for each beam that will give the final total dose distribution that most closely resembles some plan specified by the radiation oncologist. Figure 2 illustrates a five-angle treatment setup schematically.

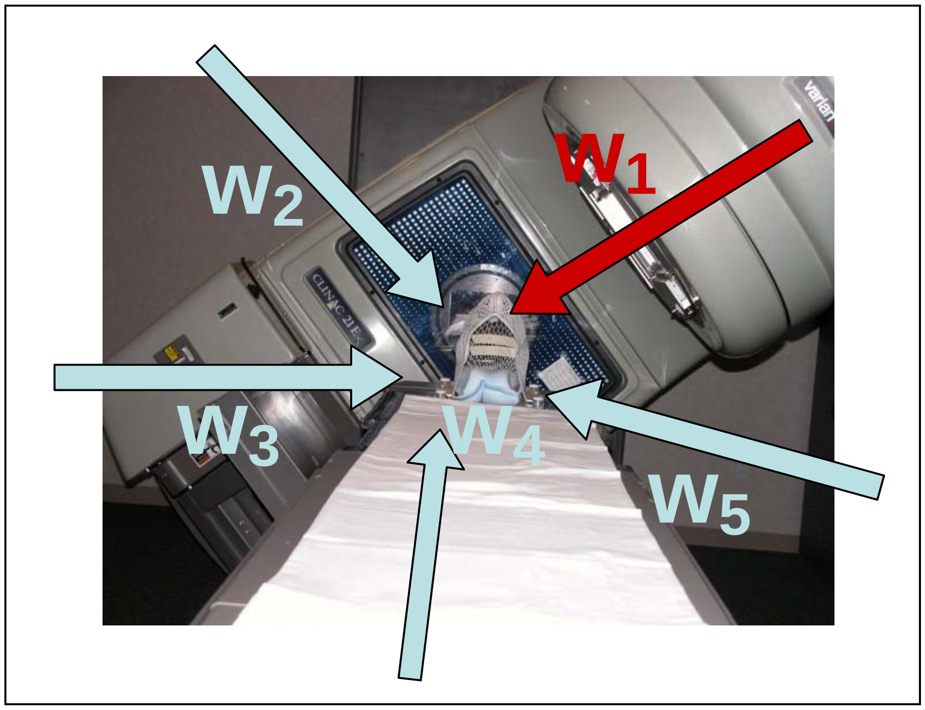


Figure 2: Five-angle treatment planning scheme. The red arrow represents the angle for which the linear accelerator is currently

aligned. Original image from (Fremont-Rideout Health Group, 2005).

The implementation of TOLSTOY makes direct use of two existing software tools to complete the tasks described above. Los Alamos National Laboratory's MCNP radiation transport code (Los Alamos National Laboratory, X-5 Monte Carlo Team, 2005)performs the x-ray transport calculations, and the commercially developed General Algebraic Modeling System (GAMS) (GAMS Development Corporation, 2008)with the (ILOG, 2008)CPLEX solver {{65 ILOG 2008}}performs the optimization. Thus, we can describe the design of TOLSTOY with respect to a number of single-purpose modules:

1. **Transport module**

   Input: a specially formatted MCNP input file, a number of angles for the treatment plan, and a value of the fluence of the x-ray beam

   Task: calculates a source transformation to model the x-ray beam at each treatment angle and runs the corresponding transport problems

   Output: raw data files containing the dose distribution data for each treatment angle

2. **Parsing module**

   Input: the specially formatted MCNP input file, the number of treatment angles, and the raw dose distribution data

   Task: creates a dose distribution matrix (and a dose distribution uncertainty matrix) for each treatment angle and creates a GAMS input file for solving the optimization problem subject to the prescription encoded in the MCNP input file

   Output: dose distribution matrices, dose distribution uncertainty matrices, information about problem geometry, and a GAMS input file

3. **Planning module**

   Input: the GAMS input file

   Task: runs the optimization problem

   Output: optimal beam weights, objective function score

4. **Plotting module**

   Input: the final plan's dose distribution matrix computed from the angle-specific matrices and the optimal weights, information about problem geometry

   Task: prepares data files for suitable visualization of the final treatment plan

   Output: data files for plotting

The following sections highlight key points of the methodology used in these modules. The source code for TOLSTOY, which is implemented in Python, is given in the Appendix. Complete information for preparing TOLSTOY input can be found in the introductory comments in each source code file; this paper is not intended to be documentation for using the code.

2.1 Transport methods

One of the ways in which TOLSTOY is very much a toy is that it uses a geometry specified analytically in accordance with MCNP's usual input file requirements, rather than real patient data from CT scans. Thus, we (arbitrarily) allow the patient region of interest to be specified inside a 10x10x10 cm cube. Patient structures (organs) can be specified with three types of MCNP surfaces: spheres, ellipsoids, and infinite cylinders (i.e., cylinders that span the entire patient volume—we do not support cylindrical volumes whose ends lie in the region of interest). Surfaces that define target and sensitive tissue (see "Optimization formulation" below) are marked with special tags that the TOSTLOY parsing module recognizes and that specify information about the dose prescription for the organs that those surfaces define.

The actual transport problems TOLSTOY solves with MCNP are very simple: a planar x-ray source travels through 95 cm of air and then enters the patient volume, depositing its energy via the usual photon-matter interactions (mostly Compton scattering). The source can be modeled however the user chooses to specify in the MCNP input file.

In the problems presented here, we use a mono-directional, mono-energetic, "perfect BEV" source. The mono-directional approximation is a good one, since the x-ray beam gets collimated inside the linear accelerator that produces it. The mono-energetic approximation is not as good. Blanchard, et al. (Blanchard et al., 2006) discuss in some depth the challenges associated with modeling the bremsstrahlung process used to create x-rays. That source also gives x-ray spectra produced by two different linac target designs; one of these spectra could be approximated in MCNP as a piecewise-linear function of energy to give a more realistic treatment of the beam energy. However, a mono-energetic source is acceptable for demonstration purposes. Finally, by "perfect BEV" above we mean that no effort is made to capture the geometry of the multi-leaf collimator; we simply specify that the shape of the planar source emerging from the MLC is the projection of the target structure onto the source plane. Figure 3 illustrates the problem geometry.

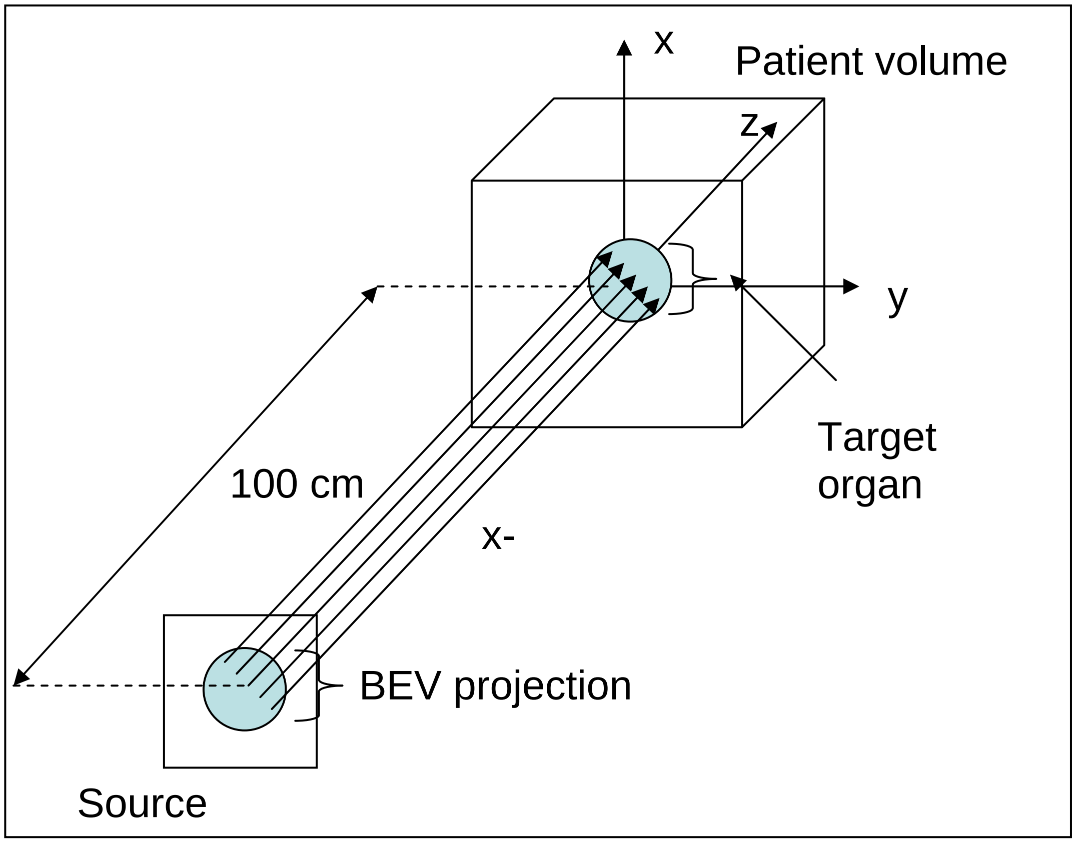


Figure 3: Schematic of a typical treatment planning geometry, for a given angle. TOLSTOY sets up one such problem for each treatment angle, rotating the source plane about the *x*-axis.

We collect the dose distribution data via an MCNP superimposed mesh tally (fmesh4) using the method suggested by Leone, et al. (Leone et al., 2005). The mesh tally splits the patient volume into a grid of (orthogonal) voxels and reports an x-ray flux in each one. We convert these fluxes to dose rates using a standard flux-to-dose table (ANS-6.1.1 Working Group, 1977) entered into the input file as a tally multiplier card. This tally multiplier operates on the uncertainty values MCNP reports as well as on the actual dose rates.

The only real work TOLSTOY does before running the transport problems is to create a new MCNP input file for each of the equally spaced beam angles. The number of angles is specified as input. TOLSTOY accomplishes this task by replacing a special placeholder in the given input file with a source transformation. This transformation rotates the source through an angle θ about the *x*-axis. Such a rotation can be described by the following matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \qquad (1)$$

2.2 Computational geometry concerns

As each transport problem runs, MCNP creates a mesh tally output file containing the dose rate distribution and associated uncertainty. These files get read by the TOLSTOY parsing module to create dose distribution matrices (see "Optimization formulation" below). Two points about this process are worth noting. First, because of the multidimensional nature of the optimization problem we need to solve and the difficulty in handling multi-dimensionality in GAMS, we'll actually vectorize the mesh tally grid and associated dose distribution matrices. To do so (and then to undo it when we're ready to prepare a data file for plotting), we use the following standard transformations:

$$\begin{aligned}
&l = i + I(j-1) + IJ(k-1)\\
&\Updownarrow\\
&i = (l \bmod IJ) \bmod I \qquad\qquad (2)\\
&j = ((l-i) \bmod IJ)/I + 1\\
&k = (l - i - I(j-1))/IJ + 1
\end{aligned}$$

where *i*, *j*, and *k* are the matrix indices of the *l*th vector element, and *I*, *J*, and *K* are the number of mesh cells in the *x*, *y*, and *z* directions, respectively.

Second, we need of way of distinguishing whether each vector element *l* lies in a target region, and sensitive organ region, or a tissue region. To perform this sort, we take advantage of the object-oriented capabilities of Python and create shape classes corresponding to the MCNP surfaces TOLSTOY supports for geometry specification. Thus, the parsing module must also read the original MCNP input file to create and store shapes associated with patient tissues. As it creates dose distribution matrices, it also checks whether element *l* is located in a "target shape" or "sensitive shape" (if neither is true, that element represents normal tissue). When it performs this check on all *IJK* voxels, it performs the simplest possible calculation and classifies the voxel based on the point at its center. Thus, a voxel could get classified as being in the target region even if it is not *entirely* in the target region. The treatment planning errors this decision introduces shrink as we let the mesh size get small, but this is nevertheless an important effect to keep in mind.

2.3 Optimization formulation

As in any optimization problem, conformal radiation treatment planning attempts to maximize or minimize some objective function subject to a list of constraints. We choose the simplest such scheme suggested by Lim, et al. (Lim et al., 2002). This formulation can be expressed as a linear program[1], a well studied class of optimization problem that can be solved very efficiently with CPLEX (ILOG, 2008).

We'll let *T*, *S*, and *N* denote the sets of voxels contained in target structures, sensitive structures, and normal tissue, respectively. Target structures are the tissue we want to irradiate;

[1] The objective function of which provides the titular **L**inear **S**coring in TO**LS**TOY.

sensitive structures are tissues whose dose we want especially to limit. Let *A* be the set of possible treatment angles. Let d_a be the matrix of dose rates delivered to the mesh cells (*i,j,k*) from angle *a*, where $i \in \{1,2,\ldots I\}$, $j \in \{1,2,\ldots J\}$, $k \in \{1,2,\ldots K\}$. Let *D* be the weighted sum of these dose distribution matrices that represents the final treatment plan. The units of d_a are Gy/s, so we can let the angle-specific weights w_a be a the time, in seconds, that the beam at angle *a* should be turned on. Using this notation, we can formulate the problem for solving for the beam weights as follows:

$$\begin{aligned} &\min_w f(D) \\ &s.t. D = \sum_{a \in A} w_a d_a \qquad (3) \\ &w_a \geq 0, \forall a \in A \end{aligned}$$

All that remains is the choice of the function *f*. We must choose *f* so that it measures the deviation from the prescribed dose distribution. If we choose *f* to be a linear function, (3) is a linear program, and we can be sure CPLEX will return an optimal solution as long at the problem is feasible and bounded. Thus, Lin, et al. suggest the following linear objective function (with additional constraint):

$$\begin{aligned} &f(D) = \lambda_T \frac{|D_T - Rx_T|_1}{card(T)} + \lambda_S \frac{|V_S|_1}{card(S)} + \lambda_N \frac{|D_N|_1}{card(N)} \qquad (4) \\ &s.t. V_S \geq 0, V_S \geq D_S - Rx_S \end{aligned}$$

where D_T, D_S, and D_N are the actual doses delivered to target, sensitive, and normal voxel sets, respectively, "card" denotes the cardinality operator (returns the number of elements in a set), and Rx_T and Rx_S are the dose prescription for the target cells and the dose limit for the sensitive cells, respectively (note that the *D*'s, *Rx*'s, and *V*'s are therefore vector quantities). Thus, this objective function penalized under-dose and over-dose to the target, over-dose to the sensitive tissue, and any dose to the normal tissue. The vector (λ_T, λ_S, λ_N) (hereafter "the importance vector") weights the relative importance of these three types of penalty. Thus, an importance vector of (1, 0, 1) only attempts to treat the target with the prescribed amount of radiation and to spare surrounding tissue to the extent possible, and it treats these goals as equally important. An importance vector of (1, 1, 1) adds the additional objective of trying to keep the dose to tissue in sensitive regions below some prescribed threshold. In a clinical setting, the radiation oncologist would make decisions about the dose prescriptions, sensitive limits, and importance vectors on a case-by-case basis.

In TOLSTOY, the importance vector is specified by setting λ_T and λ_S and taking λ_N to always equal one. As currently implemented, the code requires the target dose prescription, Rx_T, and sensitive limit, Rx_S, to be constant over the sets *T* and *S*, respectively, although one can certainly envision cases where some tissues are more sensitive than others. This choice was a matter of convenience and ease of implementation rather than a considered design decision, and it would be fairly easy to remedy. See the introductory comment in source code file "parse.py" for more information.

2.4 Visualization

There are two common visualization techniques used to review treatment plans; due to time constraints, we have only implemented one. The TOLSTOY plotting module currently supports plotting of dose distribution "slices." In fact, it doesn't really even plot these slices, it just creates a couple of data files (one in a column format suitable for gnuplot and one in a grid format suitable for MATLAB). Nevertheless, TOLSTOY's slice-plotting feature makes it possible to create dose contour plots that look very much like those used by treatment planners. Figure 4 shows a sample dose contour plot over CT image data.

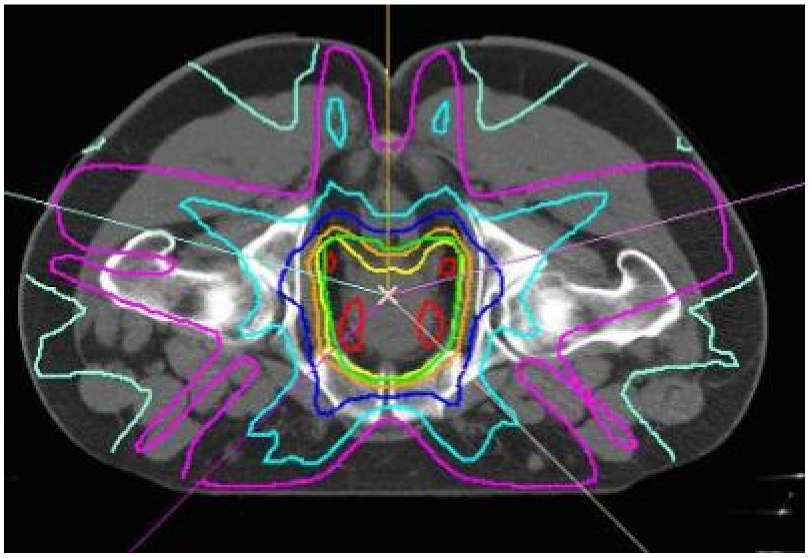


Figure 4: Sample slice-based dose contour plot of a treatment plan for prostate cancer treatment. Isodose lines are overlaid over CT image. Image from (Rensselaer Polytechnic Institute, 2007).

The second common visualization technique, which we might place at the top of a TOLSTOY "wish list," is the so-called dose-volume histogram. This image plots volume fraction of the target versus dose delivered, which allows the oncologist to see how much of the target receives a given dose (see sample in Figure 5). The data structures are certainly in place to implement a dose-volume histogram routine in TOLSTOY, we just ran out of time.

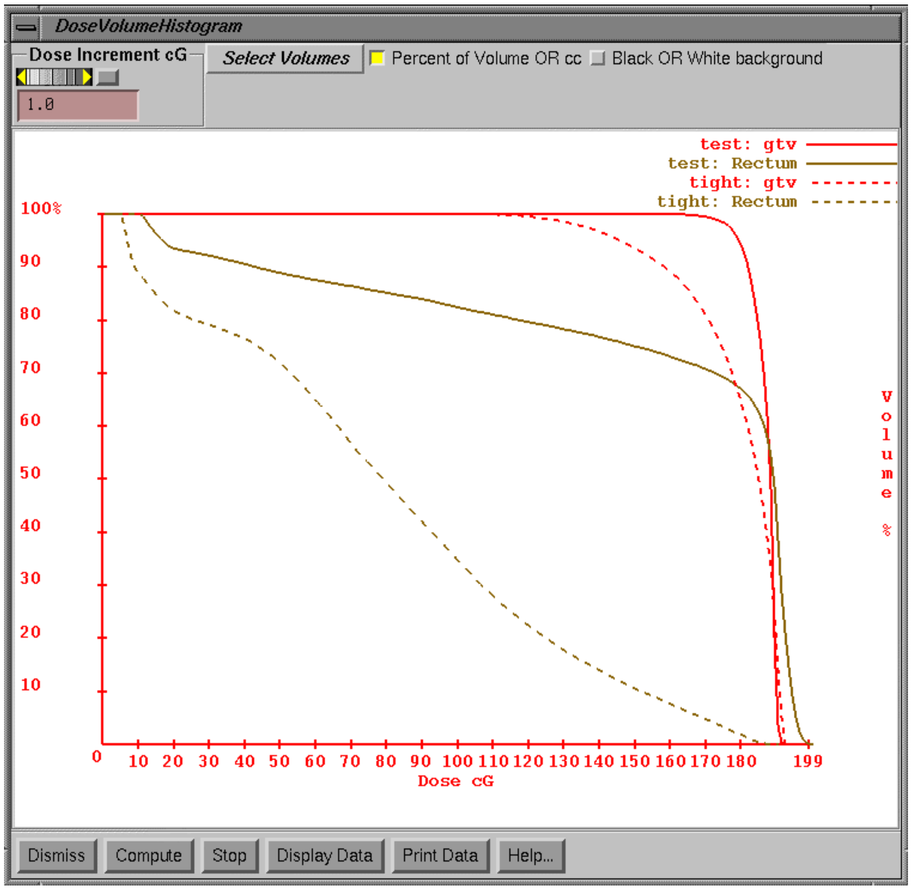


Figure 5: Sample dose-volume histogram comparing a number of different treatment plans. Image from (Math Resolutions, 2003).

3 RESULTS AND DISCUSSION

3.1 Problem description

This section will outline the results of an instructive test problem suggested by UW-Madison Computer Sciences Professor Robert Meyer, who is an active treatment planning researcher. In this prostate cancer problem, we will represent the target (prostate) as a sphere and the sensitive organs (bladder, rectum) as an ellipsoid and cylinder, respectively. We will use an image given in Wang, et al. (Wang et al., 2005) (see Figure 6) as our guide in building the geometry, and we will use the material data given in Usgaonker (Usgaonker, 2003). The geometry for this problem is specified in the sample input file in the Appendix and illustrated in Figures 7 and 8. For all runs, we'll let the dose prescription for the tumor be 60 Gy and the dose limit prescription for the sensitive organs be 5 Gy.

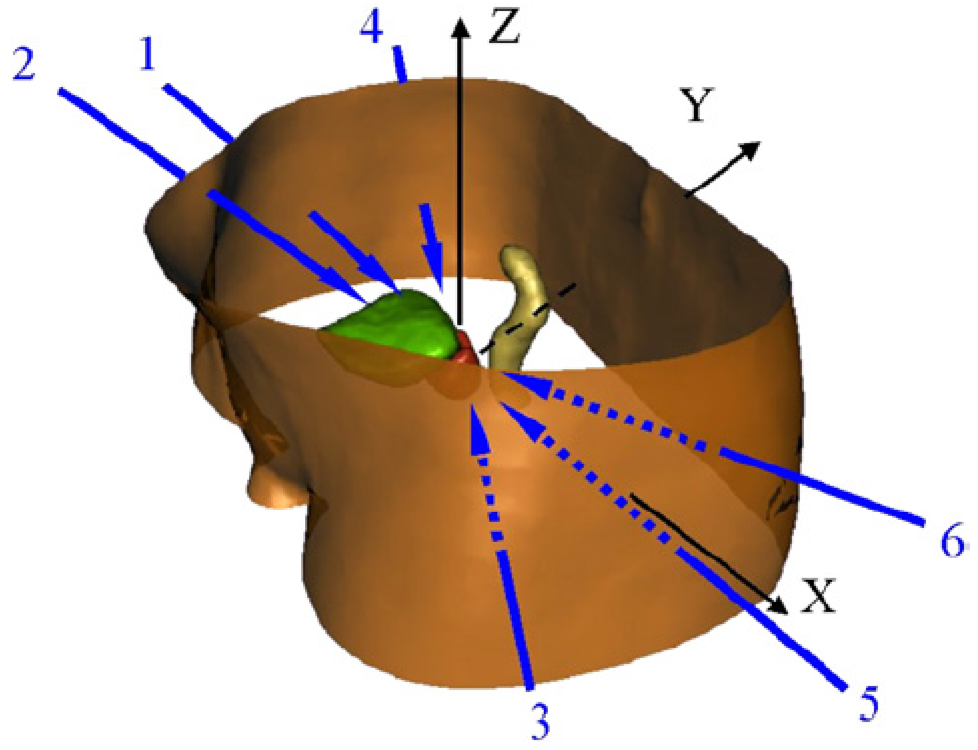


Figure 6: Real-world treatment planning model of prostate cancer patient. Numbered lines are optimal beam directions computed with an adjoint method. Image from (Wang et al., 2005).

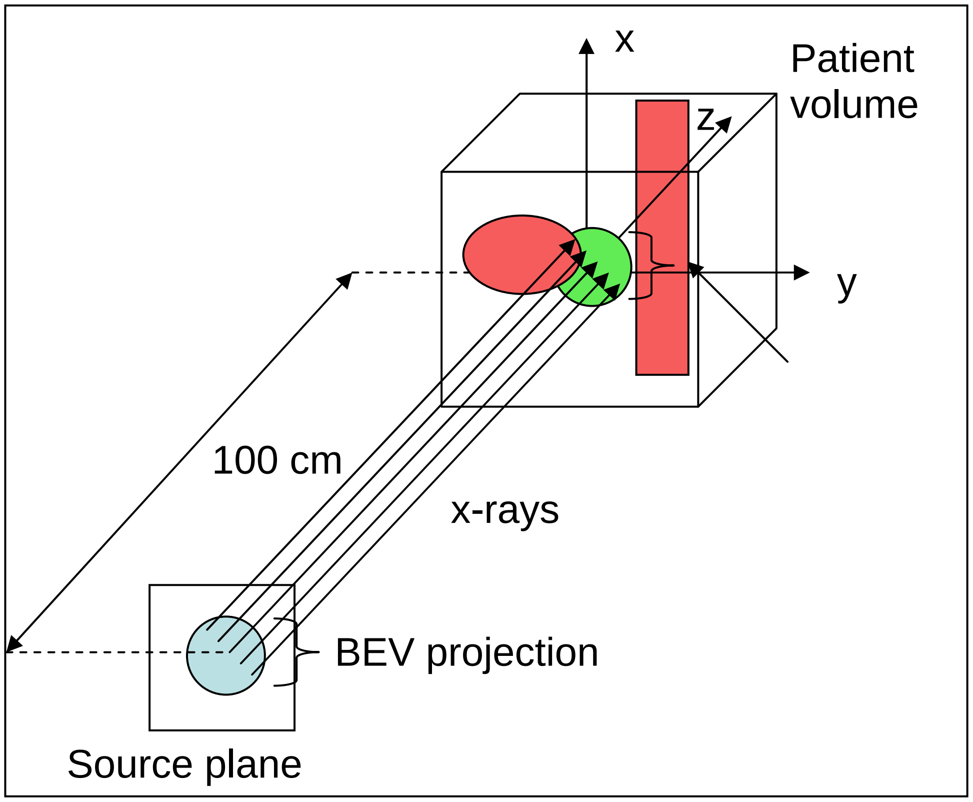


Figure 7: Schematic of a geometry for prostate cancer problem, for a given angle (θ = 0). Red organs (bladder and rectum) comprise the sensitive volume, the green organ (prostate) is the target.

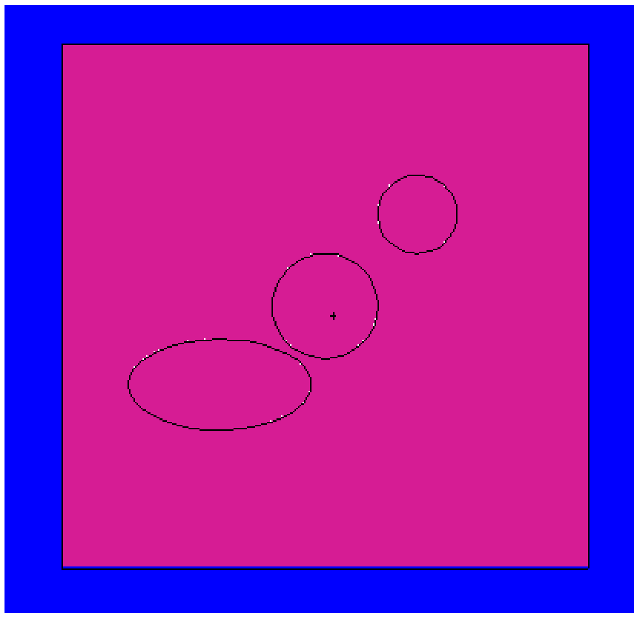


Figure 8: Slice through problem geometry at z = 0 cm. The magenta material is tissue, the blue, air. The organs shown are (from bottom-left) the bladder, prostate, and rectum.

Careful examination of Figure 7 leads to an observation that adds another item to our hypothetical TOLSTOY wish list. Note that if we wanted to treat the bladder rather than the prostate, we would need to manually re-specify the source at each angle, since the projection of the bladder onto the source plane is not invariant under our x-axis rotation. However, TOLSTOY can perform as advertised for a single treatment organ that is both centered at the origin and symmetric about the x-axis. It would be a fairly modest extension of the current code to implement a means of relaxing this requirement by automating the creation of the MCNP source specification card itself instead of merely a transformation card for a user-specified source. Source sampling using so-called "cookie cutter rejection" would likely be useful for this task, though note that it would require moving some of the functions currently in the TOLSTOY parsing module into the transport module.

3.2 Uncertainty analysis

Speaking of symmetry, the symmetry of the target organ suggests a test run that will allow us to probe several important properties of this treatment planning implementation. For starters, recall that Monte Carlo methods require a large number of simulated particles in order to achieve validity to within some statistical tolerance. TOLSTOY reads but does not currently process the uncertainty data reported by the MCNP mesh tally. A rigorous implementation would require a cell-by-cell check to ensure that, for the number of particles simulated in a given run, the uncertainty in each voxel was acceptable.[2]

Instead, we'll do a numerical experiment on a problem we know the answer to, and we'll check to see how our computed answer varies as a function of increasing number of simulated particles. Consider a treatment plan with an importance vector of (1, 0, 1). If our only goal is to deliver some specified dose to the tumor cells and to minimize the dose to the normal tissue, then the optimal plan for a beam angle set that is symmetric with respect to the problem geometry is to have equal weights on each beam. If we let the number of angles equal four (or two), this symmetry requirement is met. Thus, Figure 9 plots the computed beam weights for an MCNP problem size of $n = \{10^3, 10^4, 10^5, 10^6\}$ simulated x-rays. Note that with increasing problem size, the computed beam weights get closer to being equal, as we'd expect. The uncertainty error manifests itself as unequal beam weights. Based on this test (and our desire to be able to run cases in a timely manner) we run most other problems with $n = 5 \times 10^5$ particles and will leave a rigorous treatment of statistical uncertainty as an exercise to our hypothetical experiential learner.

[2] In fact, careful inspection of the source code will reveal that I tried to do this but ran out of time trying to debug what I assume was some statistics-related flaw in my reasoning about what the uncertainty should be. Push it on the wish list, I guess.

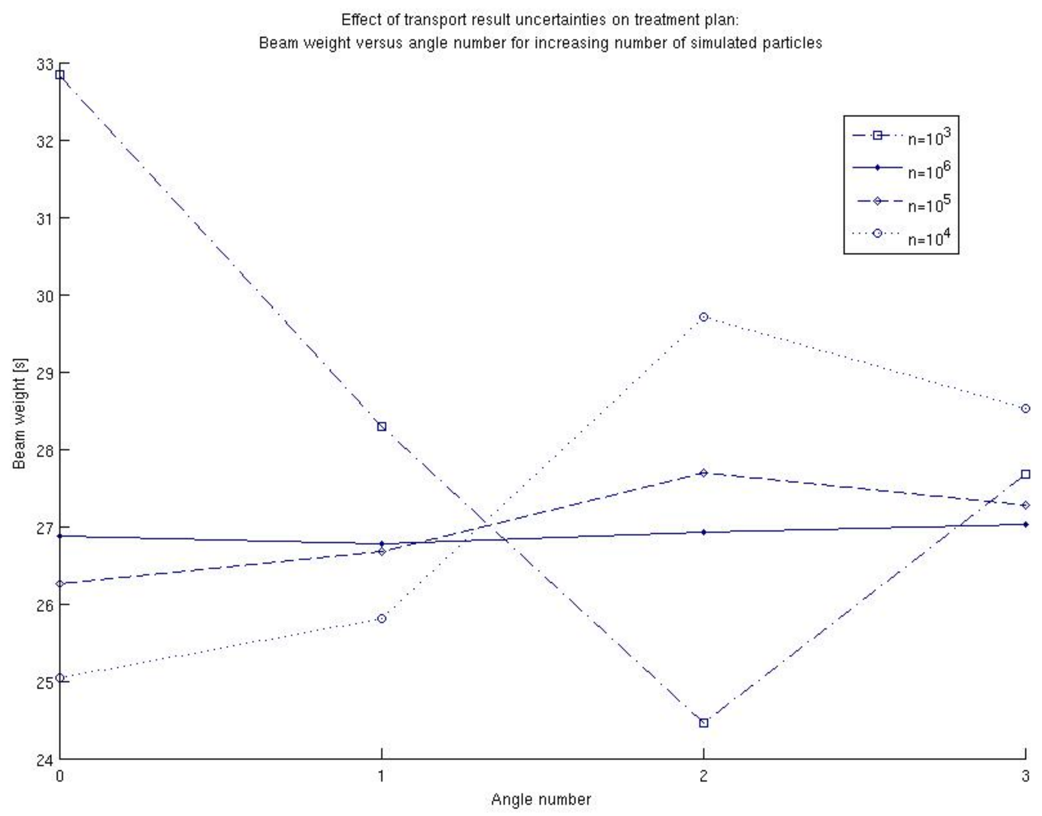


Figure 9: Results of numerical experiment to observe the effect of statistical uncertainty in the Monte Carlo method. Note that the flatter lines represent the beam weights computed from the data with less statistical uncertainty. Problem symmetry dictates that the weights should be equal.

3.3 Spatial fractionalization benefit

Of course, this obstruction-less, symmetric problem and the assumption that we used to reason through it (i.e., that we know the objective function penalizes irradiation of normal tissue, so we want to minimize that penalty by spreading the dose out evenly over many angles) suggests another numerical experiment. By running problems with an increasing number of angles, we should be able to measure this “multiple-angles gain” using the objective function itself. Thus, Figure 10 plots the optimal objective function for one-, two-, three-, and four-beam treatment plans. This plot is also pedagogically useful. It reinforces the point that the objective function isn’t some aphysical entity that just happens to help us find the right answer; it’s a score that can be computed for all feasible treatment plans. The lower the score, the better the plan.

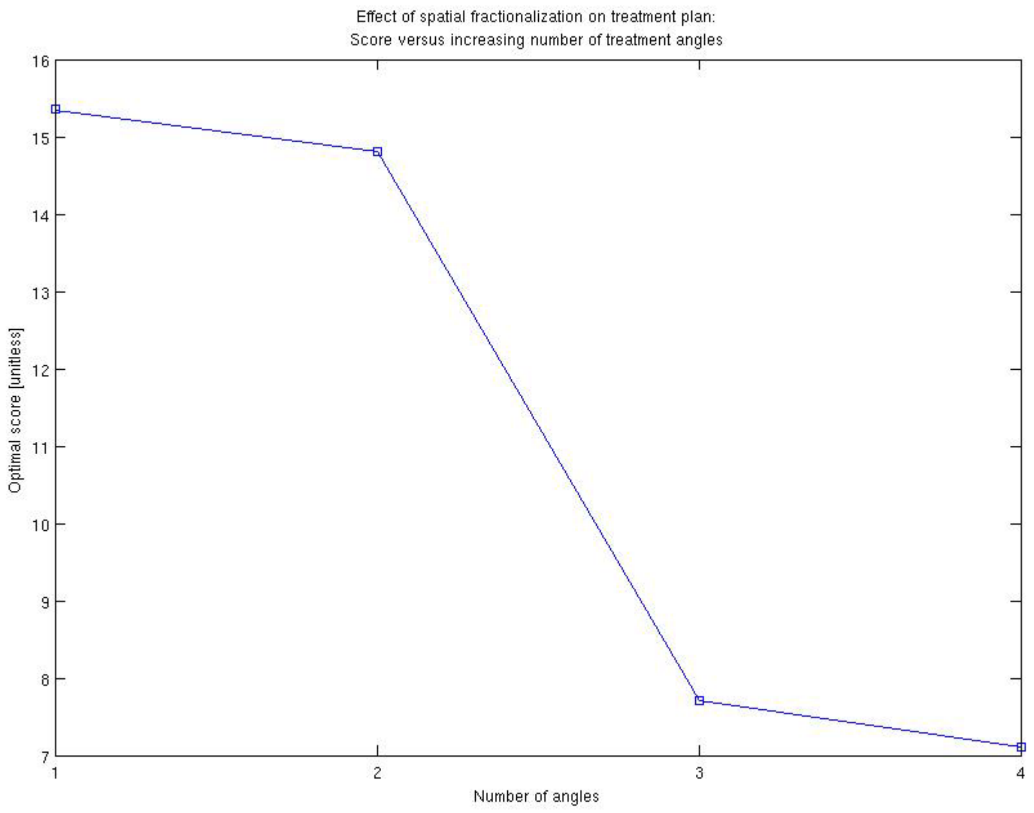


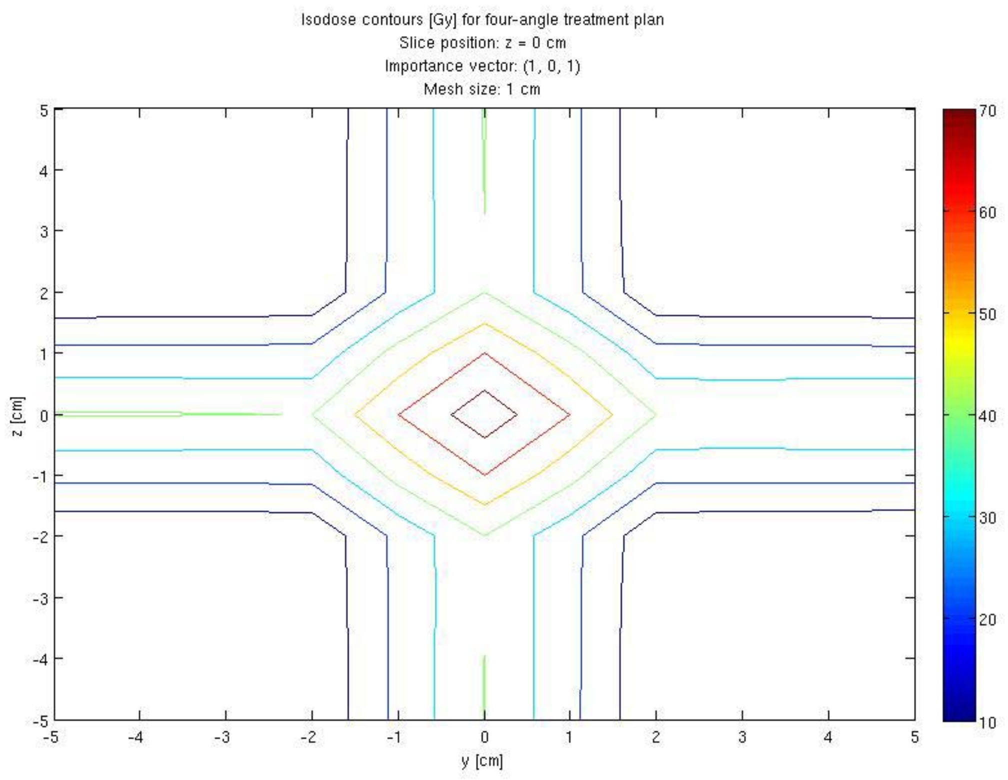
Figure 10: Results of numerical experiment to observe the benefit of a larger number of treatment angles on the optimal plan's objective function.

Two questions come to mind here. The first is "Why not let the treatment angles vary continuously rather than discretely?" One might guess that much of the motivation for the development of intensity-modulated radiation therapy is tied up in that question, and one would hope that exploring treatment plans with TOLSTOY would help students perhaps come to ask this question on their own. The second question is a finite version of the first: "Why didn't you plot, say, five- or six-angle plans on this plot?" The answer to this second question brings us to another important area of investigation: mesh size.

3.4 Mesh scaling difficulties and benefit

The reason a five-angle plan isn't plotted in Figure 10 is that the objective function actually *didn't* improve as the five-angle problem was run. Why? Well, eventually adjacent beams start to overlap, and mesh cells that lie in the overlapping regions receive a "double dose," which the objective function penalizes in kind. As we'll see in a moment, though, a five-angle plan doesn't actually introduce enough overlap to make such a plan disadvantageous. The problem is that, for the gigantic 1 cm mesh size, the overlap penalty gets artificially exaggerated. Unfortunately, smaller mesh sizes cause the optimization problem to exceed the constraint limit on the free trial version of GAMS/CPLEX, which is all the UW-Madison College of Engineering has installed on its machines. The CS department controls access to the full version, but their GAMS liaison is understandably hesitant to disclose the UW-Madison research- and teaching-use GAMS license key so that the full version of can be run on CAE machines. Conversely, Los Alamos National Labs is understandably hesitant to allow MCNP to be installed on unauthorized machines (it might even be a federal offense). The point is, this self-sufficient code becomes sort of an awkward monster when running finer-mesh problems. You have to run the transport calculations on CAE machines, then move everything over to CS machines and run the optimization calculations there. It can be (and indeed was) done, but it's a huge time sink, and so we'll limit the discussion of problems that require smaller mesh sizes.[3]

The first demonstration that shrinking the mesh does indeed work (and does present all the accuracy benefits and execution-time detriments that one expects) is shown in Figure 11, where we give a comparison of our (1, 0, 1) importance vector problem run with four treatment angles and mesh sizes of 1 cm and 0.5 cm, respectively. Note the improved smoothness of the isodose contours for the problem with the smaller mesh size.



[3] Which, I have to admit, is pretty much all of them.

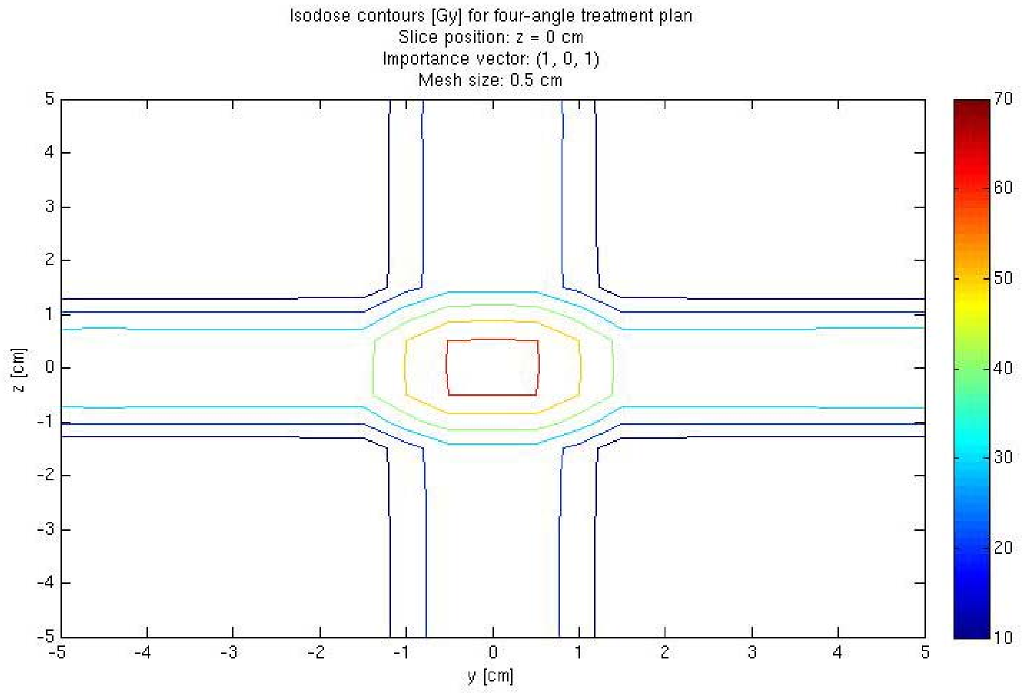
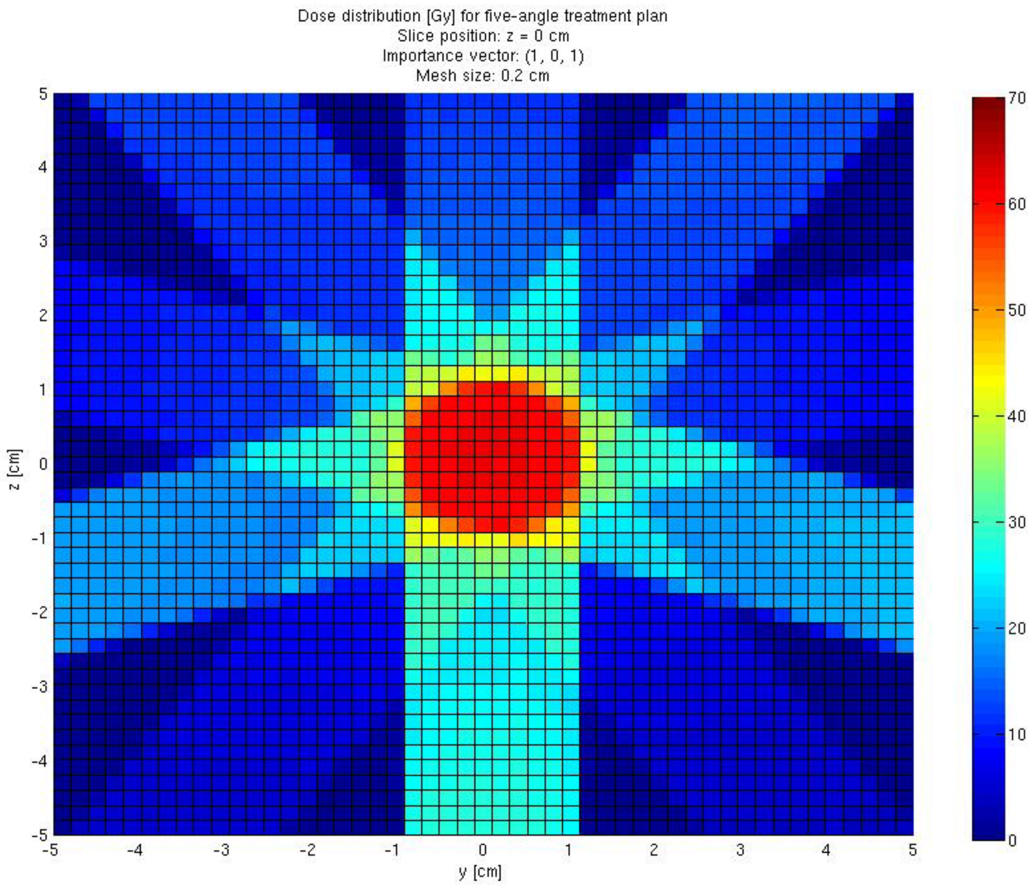


Figure 11: Slice-based isodose contour plots for the optimal treatment plans computed with mesh sizes of 1 cm (above) and 0.5 cm (below).

Aside from the data porting issues, there are of course serious performance issues to consider as the mesh size gets closer to something that might be used in the real world. Since this isn't a high-performance computing class, we didn't worry too much about performance, but it's important to demonstrate that, given enough computing time, TOLSTOY could work on a larger scale problem. Thus, Figure 12 plots the dose distribution and isodose contours[4] for our (1, 0, 1) problem with a more realistic mesh size of 0.2 cm. In this case, the mesh size is small enough that the "artificial overlap penalty" effect discussed earlier goes away, and we get a final treatment plan that makes pretty full use of each beam in a five-beam problem.

[4] The actual dose distribution map doesn't seem to come up in a clinical context, but it just looked so cool in this case that I had to include it.


Dose distribution [Gy] for five-angle treatment plan
Slice position: z = 0 cm
Importance vector: (1, 0, 1)
Mesh size: 0.2 cm
z [cm]
y [cm]
5
4
3
2
1
0
-1
-2
-3
-4
-5
70
60
50
40
30
20
10
0

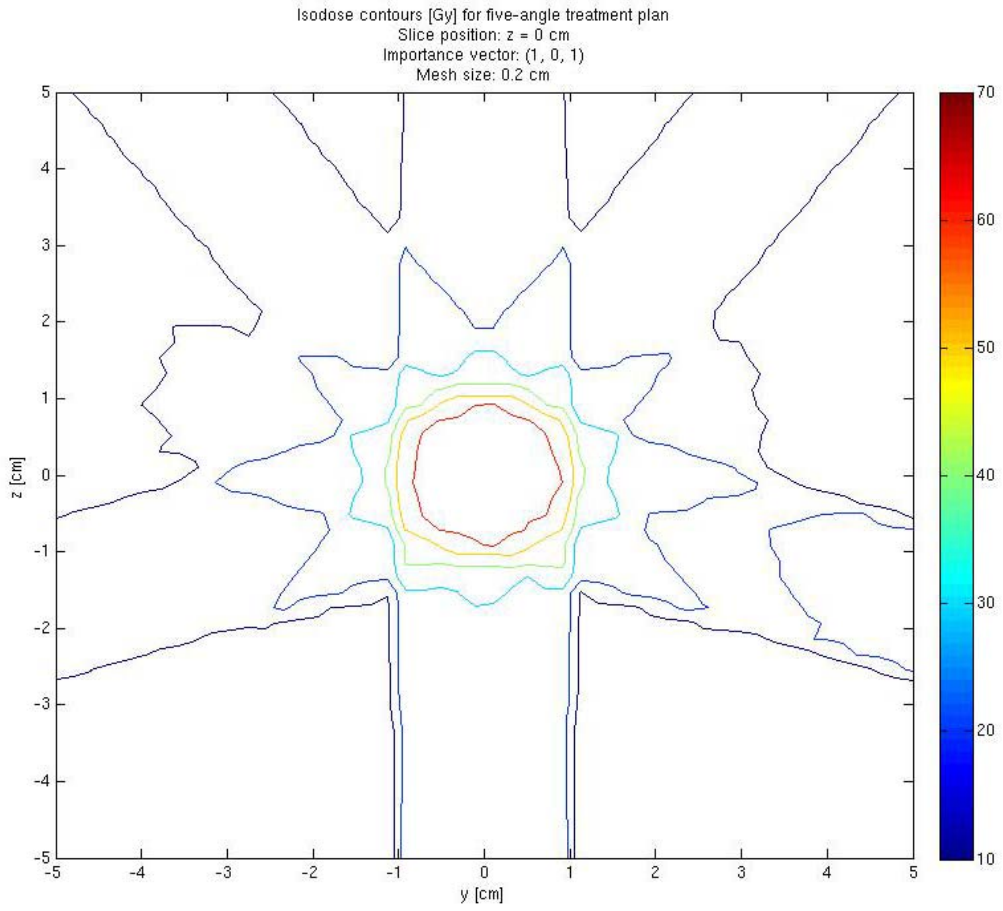
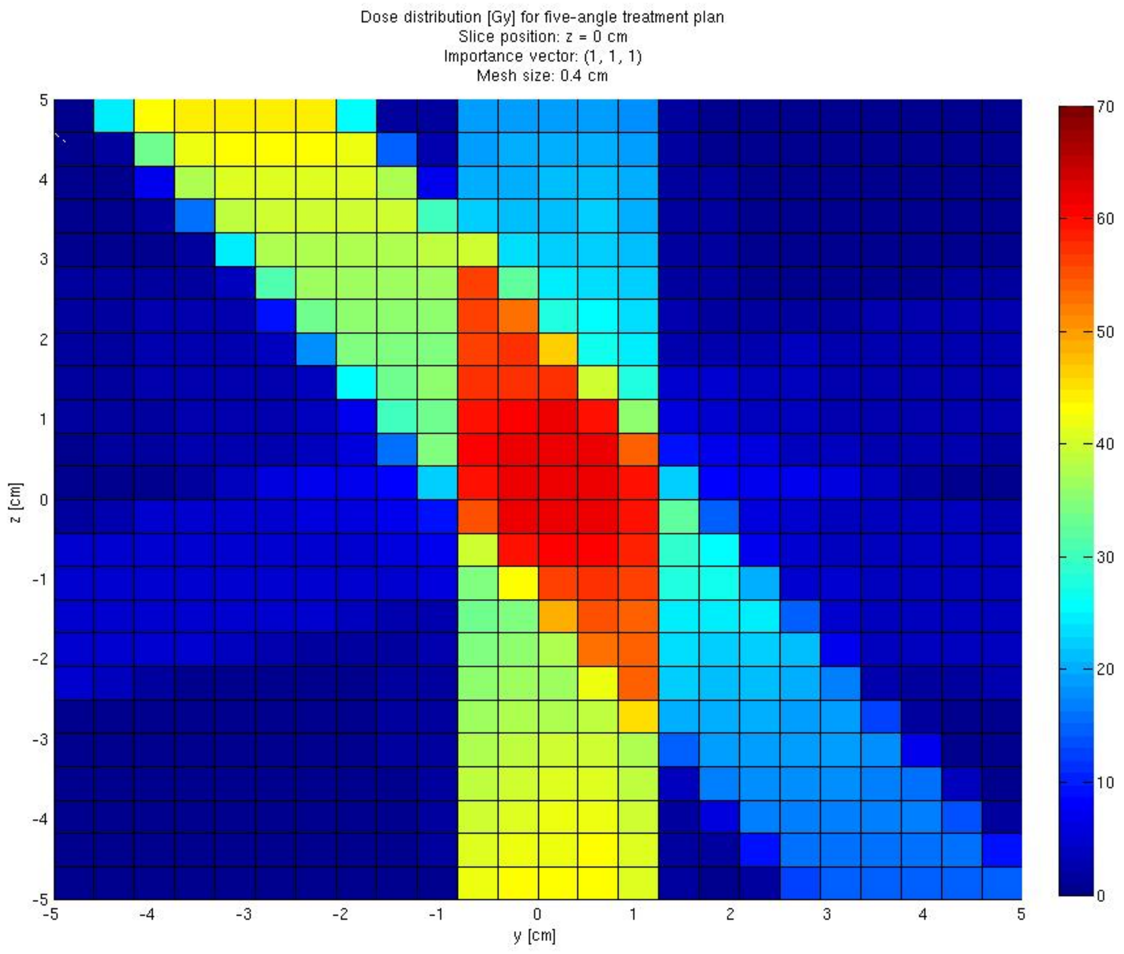


Figure 12: Dose distribution (above) and isodose contours (below) for (1, 0, 1) problem with 0.2 cm mesh size and five treatment angles. Note that Figure 12 is starting to somewhat resemble the real-world isodose plot in Figure 4.

3.5 Importance vector variation

Lastly, we will address variation of the importance vector. After all, the choice of test problem was largely motivated by the desire to calculate treatment plans that spare sensitive tissue (i.e., the bladder and rectum). Figure 13 plots the dose distribution and isodose contours for a problem with an importance vector of (1, 1, 1), five treatment angles, and a more modest 0.4 cm mesh size. We've roughly sketched in the organs of interest in the isodose contour plot (compare again to Figure 4) to illustrate that TOLSTOY has done a reasonable job of sparing the sensitive tissue.


Dose distribution [Gy] for five-angle treatment plan
Slice position: z = 0 cm
Importance vector: (1, 1, 1)
Mesh size: 0.4 cm
z [cm]
y [cm]
5
4
3
2
1
0
-1
-2
-3
-4
-5
70
60
50
40
30
20
10
0

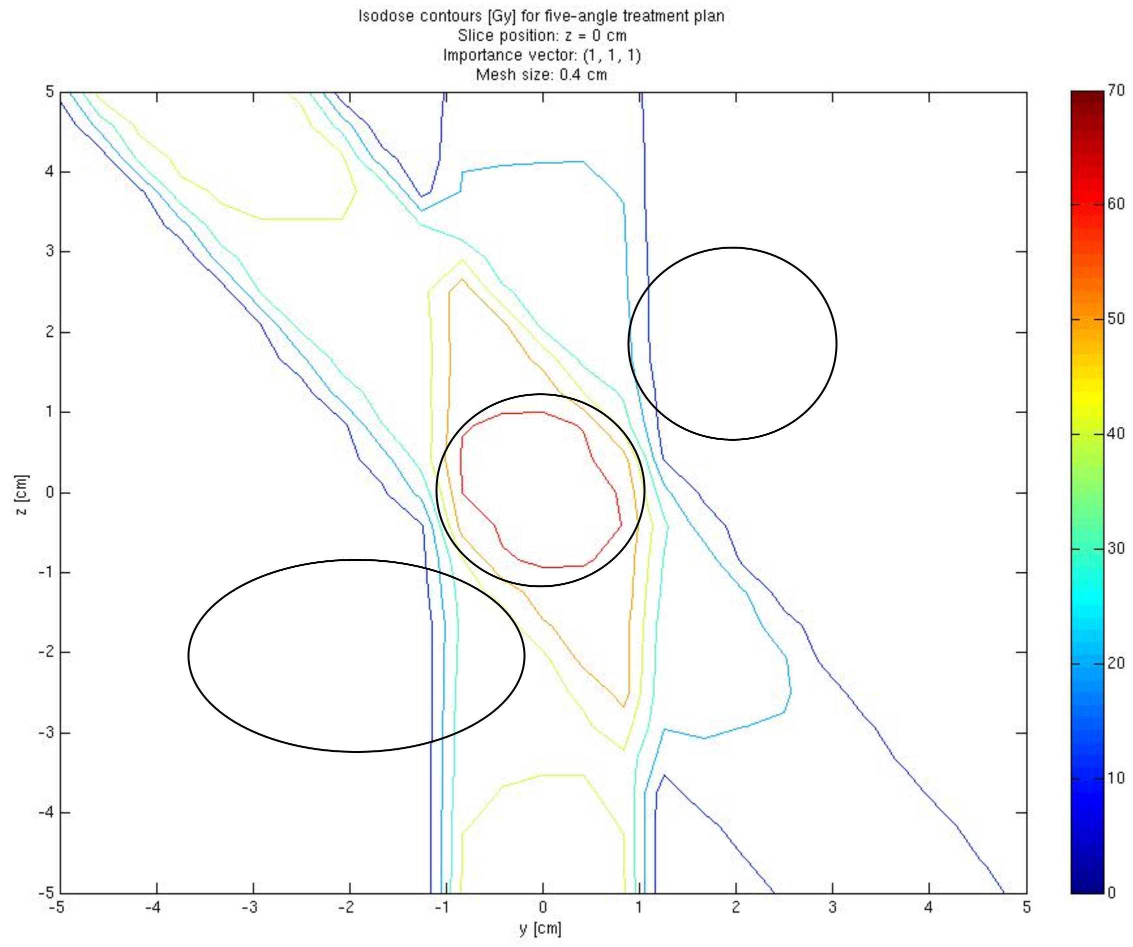
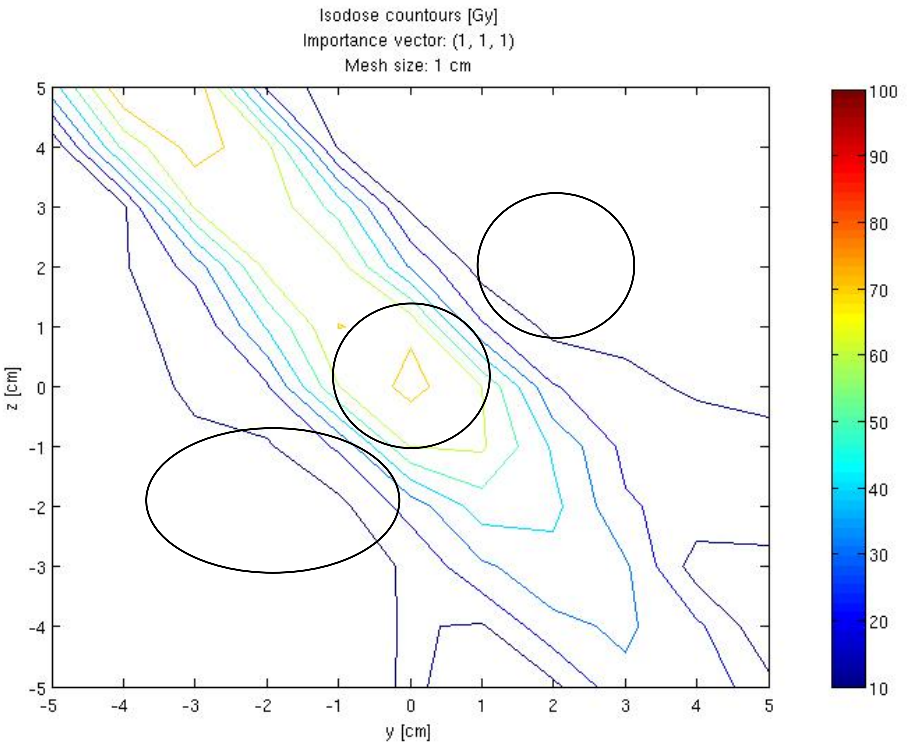
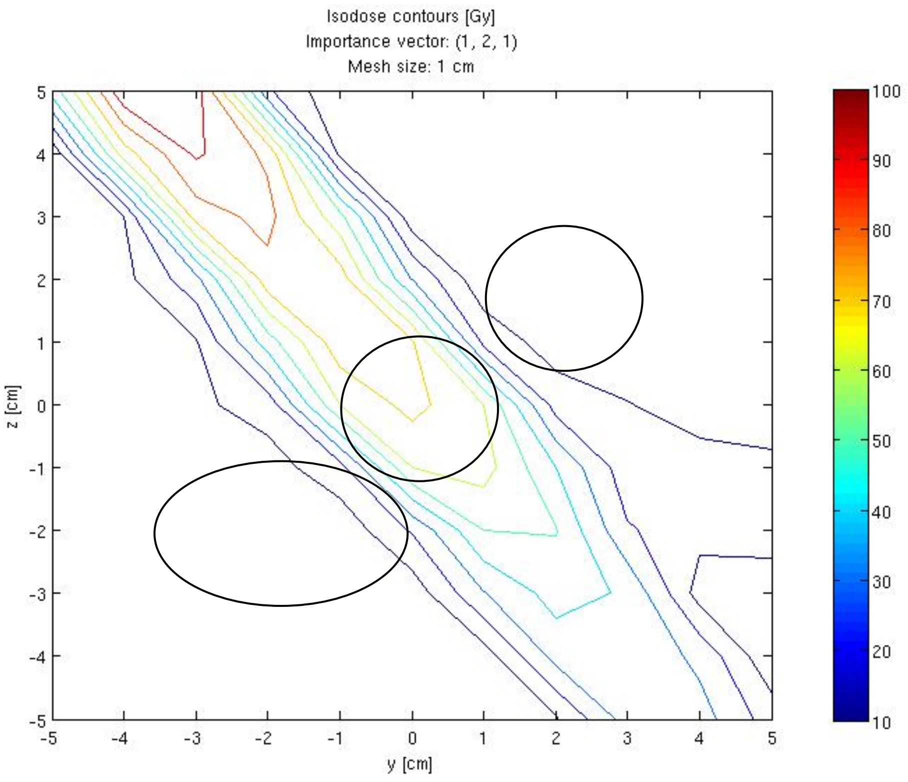


Figure 13: Dose distribution (above) and isodose contours (below) for (1, 1, 1) problem with 0.4 cm mesh size and five treatment angles. The contour plot also includes approximate sketches of the organs of interest.

As we let all elements of the importance vector be non-zero, we start to better appreciate the complexity of the treatment planning optimization problem. Whereas for the previous runs we could have perhaps chosen beam weights (or at least relative beam weights) by inspection and done a pretty good job, here we calculate a result that is at first counterintuitive. While the optimal plan obviously succeeds at treating the prostate and sparing the rectum, is appears as if giving a large weight to the beam originating at “3:30” rather than the beam at “6 o’clock” would do a better job of sparing the bladder. However, the third component of the importance vector (the one that weights the penalty for not sparing normal tissue) is still non-zero in this run, and increasing the weight of the 3:30 beam would deliver an increased dose to the voxels in the upper-lefthand corner of the plot.

We can support this claim by parametrically increasing the second component of the importance vector. As the objective function treats sparing the rectum and bladder as an increasingly high priority goal relative to sparing normal tissue and delivering the correct dose to the tumor, the optimization routine returns plans that divert more weight from the 6 o’clock beam to the 3:30 beam. This effect is captured visually in Figure 14, which plots isodose contours for problems with importance vectors of (1, 1, 1), (1, 2, 1), and (1, 3, 1), respectively.





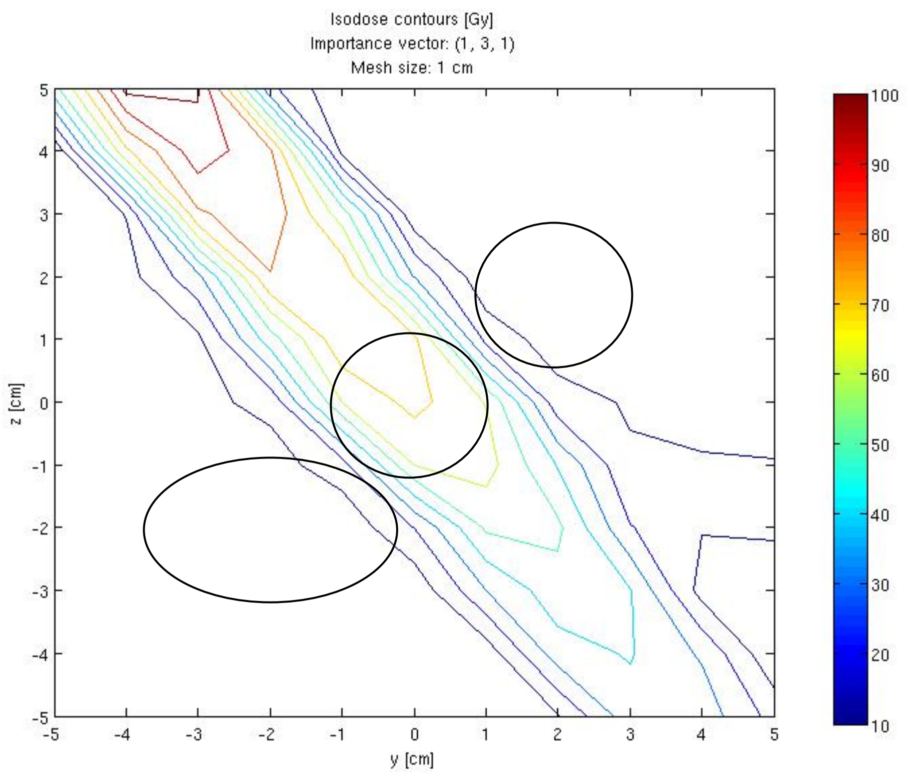


Figure 14: Isodose contours for parametrically increasing values of the second importance vector component: (1, 1, 1), top; (1, 2, 1), middle; (1, 3, 1), bottom. Note the final plan's willingness to just absolutely fry the tissue in the top left portion of the plot in the name of sparing the sensitive organs.

4 CONCLUSIONS AND FUTURE WORK

Hopefully, the previous section indicates some of the technical power of this admittedly low-performance tool. With even just a few modest generalizations and performance enhancements (or with just a bunch of patience and available computer time) some fairly complex treatment planning problems could be studied—limited in the end by one's ability (and patience) to specify the relevant geometries in MCNP.

More important, though, is the value of having a flexible demonstration tool that straddles the authenticity/accessibility spectrum and can serve a variety of pedagogical purposes. One can certainly imagine the value of a learning experience in which students get to play around with one of these toy problems, observing the effects of varying the mesh size, the number of treatment angles, the importance vectors, etc. Many of TOLSTOY's weakness can be viewed as strengths when one considers how difficult it would be to create a comparable learning environment with real-world tools. It seems reasonable to assert that a tool like TOLSTOY isolates and illustrates the *concepts* involved in performing treatment planning calculations better than a more strictly authentic tool could. Making an analogy to NE 705 material, it's fair to say that students learned more about discrete ordinate concepts by playing with the 1-D MATLAB code we used for homework than by learning how to write an input file for a commercial code. Moreover, if one's pedagogical objectives include giving students some experience actually developing software (rather than just using it), having a toy program to work from is almost essential, since just understanding the source code of a commercial tool, let alone the underlying methods, requires a huge investment of time and mental effort.

5 ACKNOWLEDGMENTS

I'd like to thank Professor Robert Meyer for sharing his considerable expertise in this field, for useful advice that helped me limit the scope of the otherwise totally intractable path I'd originally set myself on, and for letting me divert some time from my CS 719 coursework when I let TOLSTOY spiral out of control anyway. I'd also like to thank Professor Greg Moses for his course project flexibility and especially for letting me calculate the dose distributions in MCNP instead of having to learn a more curricularly relevant deterministic tool.

6 REFERENCES

ANS-6.1.1 Working Group. (1977). *American national standard neutron and gamma-ray flux-to-dose rate factors* No. ANSI/ANS-6.1.1-1977 (N666)). LaGrange Park, IL: American Nuclear Society.

Blanchard, A., Dalton, L., Geurts, M., Jaeger, M., Oliver, K., & Uselmann, A. (2006). *Design of a medical linear accelerator X-ray target*. Madison, WI: UW-Madison Department of Engineering Physics.

Brown, A., Luyendyk, S. & Ollis, D. (1997). *Writing across engineering: A first year implementation.* Retrieved 11/11, 2004, from http://www.succeed.ufl.edu/content/Ollis%20Process%20Dissection%20Lab/E123%20Main/P0.html

Crawley, E. F. (2002). Creating the CDIO syllabus, A universal template for engineering education. Paper presented at the Boston, MA.

Fremont-Rideout Health Group. (2005). *Radiation therapy.* Retrieved May 14, 2008, from http://www.frhg.org/hospital.aspx?id=114

GAMS Development Corporation. (2008). *General algebraic modeling system* Retrieved from http://www.gams.com/Default.htm

ILOG. (2008). *CPLEX* Retrieved from http://www.ilog.com/products/cplex/

Kolb, D. A. (1984). *Experiential learning: Experience as the source of learning and development*. Englewood Cliffs, N.J.: Prentice-Hall.

Leone, J., Furler, M., Oakley, M., Caracappa, P., Wang, B., & Xu, X. G. (2005). Dose mapping using MCNP5 mesh tallies. *Health Physics, 88*(2), S31-S33.

Lewis, E. E., & Miller, W. F. (1993). *Computational methods of neutron transport*. La Grange Park, IL: American Nuclear Society.

Lim, J. H., Ferris, M. C., Wright, S. J., Shepard, D. M., & Earl, M. A. (2002). *An optimization framework for conformal radiation treatment planning* No. 02-10)UW-Madison Computer Sciences Optimization Group.

Los Alamos National Laboratory, X-5 Monte Carlo Team. (2005). *MCNP — A general monte carlo N-particle transport code* Retrieved from http://mcnp-green.lanl.gov/

Math Resolutions, L. (2003). *RtDosePlan.* Retrieved May 13, 2008, from http://www.mathresolutions.com/rtdesc.htm

Rensselaer Polytechnic Institute. (2007). *Machine learning could speed up radiation therapy for cancer patients.* Retrieved May 13, 2008, from http://www.medicexchange.com/mall/departmentpage.cfm/MedicExchangeUSA/_81675/777/departments-contentview

Usgaonker, S. R. (2003). *MCNP modeling of prostate brachytherapy and organ dosimetry.* Unpublished Master of Science, Texas A&M University, Retrieved from http://txspace.tamu.edu/bitstream/handle/1969.1/305/etd-tamu-2003A-2003032612-1.pdf?sequence=1

Wang, B., Goldstein, M., Xu, X. G., & Sahoo, N. (2005). Adjoint monte carlo method for prostate external photon beam treatment planning: An application to 3D patient anatomy. *Physics in Medicine and Biology, 50*, 923-935. Retrieved from http://www.iop.org/EJ/article/0031-9155/50/5/015/pmb5_5_015.pdf?request-id=6f056716-549c-4ee6-80e4-9c1468064bc5

Yoo, S., Kowalok, M. E., Thomadsen, B. R., & Henderson, D. L. (2003). Treatment planning for prostate brachytherapy using region of interest adjoint functions and a greedy heuristic. *Physics in Medicine and Biology, 48*(24), 4077-4090.

Yoo, S., Kowalok, M. E., Thomadsen, B. R., & Henderson, D. L. (2007). A greedy heuristic using adjoint functions for the optimization of seed and needle configurations in prostate seed implant. *Physics in Medicine and Biology, 52*(3), 815-28.

APPENDIX: COMPLETE TOLSTOY SOURCE CODE

Sample input file

```
c MCNP input file for generating BEV dose distribution matrices
c for treatment optimization on spherical tumor with sensitive organs
c
c NOTE: USER IS RESPONSIBLE FOR SPECIFYING CELLS AND SURFACES FOR
PROBLEM
c GEOMETRY, THE BEV X-RAY SOURCE ITSELF, AND THE MESH SIZE. SEE
tolstoy.py AND
c parse.py FOR MORE INFORMATION ABOUT INPUT FILE PREPARATION
c
c Created by Kyle Oliver 3/3/08
c Revised 5/12/08
c
c Importance vector: (1, 1, 1)
c Mesh size: 0.5 cm
c
c Target prescription: 60 Gy
c Sensitive limit: 5 Gy
c
c Cells
1  1  -1.04     -301  103 -104  $ rectum
2  1  -1.04     -302            $ bladder
3  1  -1.04     -201            $ prostate
4  1  -1.04      301  302  201  $ rest of patient
                 101 -102
                 103 -104
                 105 -106
5  2 -0.001020  -401            $ air
              ( -101: 102:
                -103: 104:
                -105: 106)
6  0             401            $ rest of universe

c Surfaces
c Patient
101      pz  -5
102      pz   5
103      px  -5
104      px   5
105      py  -5
106      py   5
c TREAT_ORGS 1 60 1
201      so   1                             $ prostate
c SENS_ORGS 2 5 0
301      c/x  1.75 1.75  0.75               $ rectum
302      sq   1 0.25 1 0 0 0 -1 0.5 -2 -1.5  $ bladder
c Problem boundary
401      so   106

c Materials
c
```

```
c Adult tissue (density = 1.04 g/cc)
m1   1001   -0.10454
     6000   -0.22663
     7014   -0.02490
     8016   -0.63525
     11023  -0.00112
     12000  -0.00013
     14000  -0.00030
     15031  -0.00134
     16000  -0.00204
     17000  -0.00133
     19000  -0.00208
     20000  -0.00024
     26000  -0.00005
     30000  -0.00003
     37085  -0.00001
     40000  -0.00001
c
c Air (density = 0.001020 g/cc)
m2   6000 -0.00012
     7014 -0.75527
     8016 -0.23178
     18000 -0.01283
c
c Source definition
sdef  erg = 0.5 pos = 0 0 -100 tr  = 1
      par = 2   vec = 0 0  1   dir = 1
      ext = 0   axs = 0 0  1   rad = d1
c
c Source transformation
SRC_TRANS
c default: tr1  0 0 0  1 0 0  0 1 0  0 0 1  1
c
c Source distribution (uniform in x direction)
si1  0 1
sp1  -21 1
c
c Mode specification: photon mode
mode p
c
c Variance reduction
imp:p 1 1 1 1 1 0
c
c Scoring
c
fmesh4:p geom = xyz origin = -5 -5 -5 $ Mesh over patient
         imesh = 5 iints = 11 $ x mesh size
         jmesh = 5 jints = 11 $ y mesh size
         kmesh = 5 kints = 11 $ z mesh size
         out = ij $ gives 2-D x-y matrices
c
fc4 Dose mesh tally computed with ANSI/ANS conversion factors
c
de4 log  0.01 0.03 0.05 0.07 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45
0.50
         0.55 0.60 0.65 0.70 0.80 1.00 $ MeV
c
```

```
df4 log  3.96e-6 5.82e-7 2.90e-7 2.58e-7 2.83e-7 3.79e-7 5.01e-7 6.31e-7
         7.59e-7 8.78e-7 9.85e-7 1.08e-6 1.17e-6 1.27e-6 1.36e-6 1.44e-6
         1.52e-6 1.68e-6 1.98e-6 $ (rem/hr)/(p/cm^2-s)
c
c Number of particles
nps 500000
```

Template that the planning module uses to build the GAMS input file

```
$ontext

template.gms

A template from which to build GAMS input files for treatment planning.

$offtext

sets t "target voxels"
             / TAR_SET /
             s "sensitive voxels"
             / SENS_SET /
             n "normal voxels"
             / NORM_SET /
             a "treatment angles"
             / ANG_SET / ;

parameters th "target dose prescription"
                               phi "sensitive dose limit"
                               imp_t "target plan importance"
                               imp_s "sensitive plan importance"
                               imp_n "normal plan importance" ;

th = THETA_VAL ;
phi = PHI_VAL ;

imp_t = TAR_IMP ;
imp_s = SENS_IMP ;
imp_n = NORM_IMP ;

table Dt(t, a) angle-specific dose distribution in target voxels

TAR_TABLE ;

table Ds(s, a) angle-specific dose distribution in sensitive voxels

SENS_TABLE ;

table Dn(n, a) angle-specific dose distribution in normal voxels

NORM_TABLE ;


positive variables w(a) "treatment angle weights" ;

free variables score "plan-specific objective function"
```

```
                                          Vt(t) "target prescription
violation"
                                          Vs(s) "sensitive prescription
violation" ;
equations

                        tv1(t) "penalize overdose"
                        tv2(t) "penalize underdose"
                        sv1(s) "penalize overdose"
                        sv2(s) "no penalty below sensitive limit"
                        obj "objective function" ;

tv1(t)..

                        Vt(t) =g= sum(a, Dt(t, a) * w(a)) - th ;

tv2(t)..

e                       Vt(t) =g= th - sum(a, Dt(t, a) * w(a)) ;

sv1(s)..

                        Vs(s) =g= sum(a, Ds(s, a) * w(a)) - phi ;

sv2(s)..

                        Vs(s) =g= 0 ;

obj..

                        score =e=
                        imp_t * sum(t, Vt(t)) / card(t) +   imp_s *
sum(s, Vs(s)) / card(s) +
                        imp_n * sum(n, sum(a, Dn(n, a))) / card(n) ;

model dose_opt /all/ ;

solve dose_opt using lp minimizing score ;
```

Main program

```
# tolstoy.py

# The application wrapper for the modules in the TOLSTOY treatment
planning
# tool.

import sys, xport, parse, plan, plot, math, pickle

# Check number of arguments; individual modules will check argument
content.
if len(sys.argv) != 4 :
      print "\nUsage: python tolstoy.py inputFile numAngles fluence"
      sys.exit(0)
```

```
mcFile = sys.argv[1]
numAngs = sys.argv[2]
fluence = sys.argv[3]

# Note the filenames for the modules
xName = "xport.py"
parName = "parse.py"
plaName = "plan.py"
ploName = "plot.py"

# Run the transport module
xport.run([xName, mcFile, numAngs])

# Run the parsing module


#T, S, N, DDMs, DDMErrs, mmm, MMM, IJK = parse.run([parName, mcFile,
numAngs, fluence])

mmm, MMM, IJK = parse.run([parName, mcFile, numAngs, fluence])

# Run the planning module
w, obj = plan.run([plaName, mcFile, numAngs])

print "weights: "
weights = w.keys()
for i in weights :
      print i, " = ", "%e" % w[i]

# Calculate the total dose rates and errors using the weights. Don't
forget
# to sum errors in quadrature.

Tf = open("tar.vxl", "r")
Sf = open("sens.vxl", "r")
Nf = open("norm.vxl", "r")
DDMf = open("norm.ddm", "r")
DDEf = open("norm.dde", "r")

T = pickle.load(Tf)
S = pickle.load(Sf)
N = pickle.load(Nf)
DDMs = pickle.load(DDMf)
DDErrs = pickle.load(DDEf)

# Initialize.
DM_tot = {}
DMErr_tot = {}
for l in DDMs[0].keys() :
      DM_tot[l] = 0
      DMErr_tot[l] = 0

# Sum.
for l in DDMs[0].keys() :

      for i, x in enumerate(DDMs) :
            DM_tot[l] = DM_tot[l] + x[l] * w[i]
            DMErr_tot[l] = DMErr_tot[l] + math.pow(DDErrs[i][l]/ 100 *
w[i], 2) # quad.

      DMErr_tot[l] = math.sqrt(DMErr_tot[l])

# Plot the angle-specific DDMs if desired.
for i in range(0, int(numAngs)) :
      plot.run([ploName, "ang" + str(i), "slice", DDMs[i], mmm, MMM,
IJK, 0])

# Run the plotting module.
plot.run([ploName, mcFile, "slice", DM_tot, mmm, MMM, IJK, 0])
```

Transport module

```
# xport.py
# Runs the transport calculations necessary for a dose optimization routine,
# given a specially formatted MCNP input file as the first argument and a
# number of angles for which to generate dose distribution data at the second
# argument.

# The input file must contain a  treatment area centered at
# the origin. The source will be rotated through the given number of angles
# about the x axis. Thus, it should be defined with respect to an MCNP
# transformation tr1, which will be inserted at the SRC_TRANS placeholder.
# The presence of this placeholder is not checked, though failure to define
# it will crash MCNP.
#
# The treatment area must be covered by a uniformly spaced mesh tally that
# corresponds to the voxelization. It should use the appropriate tally
# multiplier card to convert to dose units.
#

# See parse.py for notes on special handling of the formatting at and near
# the surface cards defining the treatment and sensitive organs.

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Functions

# Creates an MCNP geometry transformation card for rotation about the x-axis
# by the given angle (in radians)

def makeTrans(theta) :

   pt1 = "tr1 0 0 0  1 0 0  0 "
   pt2 = str(round(math.cos(theta), 4)) + " "
   pt3 = str(round(0 - math.sin(theta), 4)) + "  0 "
   pt4 = str(round(math.sin(theta), 4)) + " "
   pt5 = str(round(math.cos(theta), 4)) + "  1"

   return pt1 + pt2 + pt3 + pt4 + pt5
```

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Get modules we need
import sys, os, shutil, math

def run(argv) :

   # Check arguments
   if len(argv) != 3 :
      print "\nUsage: python xport.py inputFile numAngles"
      sys.exit(0)
   if int(argv[2]) < 1 or int(argv[2]) > 10 :
      print "\nError: must run between one and ten angles"
      sys.exit(0)
   mcInp = argv[1]
   numAngs = int(argv[2])

   # Do some file cleanup in case of previous runs in this directory
   print "\nCleaning up old files"
   mcFiles = ["out", ".out", ".sav", "runtp", "mesh", mcInp + "o", mcInp +
"r", ".gms", ".lst"]
   thisDur = "./"

   for file in os.listdir(thisDur) :
      for name in mcFiles :
            if file.find(name) >= 0 :
                  os.remove(file)

   # Run MCNP on the given input file if it exists (for the given number of
   # angles), or exit
   if not os.path.exists(argv[1]) :
      print "\nError: missing input file"
      sys.exit(0)

   mcProg = "mcnp5"
   print "\nRunning MCNP"

   # Calculate the theta increment for the number of angles specified.
   th_incr = 2 * math.pi / numAngs

   for i in range(0, numAngs) :

      # Make a new input file name for the one with angle-specific
alterations.
      mcInpAng = mcInp + str(i)

      # Replace the source transformation placeholder in the original input
file.
      fileIn = open(mcInp)
      text = fileIn.read()
      textAng = text.replace("SRC_TRANS", makeTrans(i * th_incr))
      fileAng = open(mcInpAng, 'w')
      fileAng.write(textAng)
      fileIn.close()
      fileAng.close()
      # Do some more file cleanup so we don't crash MCNP.
      try :
            os.remove(mcInpAng + "o")
            os.remove(mcInpAng + "r")
            os.remove(mcInpAng + "m")
      except OSError :
            pass

      # Do the run.
      mcArg = "name=" + mcInpAng
      (runIn, runOut) = os.popen4(mcProg + " " + mcArg)
      print runOut.read()

      # More file cleanup.
      try :
            os.rename("meshtal", mcInpAng + "m")
      except OSError:
            print "\nError: No mesh tally file created"
            sys.exit(0)
```

Parsing module

```
# parse.py

# Parses the MCNP mesh tally files generated from the given MCNP input file,
# number of angles, and treatment beam fluence,  and creates a GAMS input
file
# for the optimization routine. To do so, it must also read the MCNP
geometry.
#
# Treatment and sensitive organs are parsed by reading the sphere, cylinder,
# and ellipsoid surfaces immediately following the comment lines
#
# c SENS_ORGS num sensLim sensImp
# and
# c TREAT_ORGS num treatRx treatImp
#
# where num is the number of surface specification lines that follow these
# tagged lines. No checking is done to confirm that these organ surfaces are
# entirely within the patient volume or that MCNP sq surfaces are ellipsoids
# and not hyper- or para-boloids. Also, ellipsoids must have axes parallel
# to the main coordinate axes. Mesh elements are considered to be contained
# by the organ if their center point is contained by the organ.
#
# sensLim is the desired limit on dose rate to the sensitive tissue, sensImp
is
# the relative importance of this limit (see main reference), treatRx is
# the desired dose to the treatment region, and treatImp is the relative
# imporance of this limit (by default, the relative importance of sparing
# normal tissue is 1)


# Get modules we need
import sys, os, shutil, math, re, string, pickle
```

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Classes

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Shape is parent to Spheres, Cylinders, Ellipsoids
class Shape(object) :
   def contains(self, x, y, z) :
      raise GeoException, "Can't call contains() on abstract class"

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Sphere class suppors MCNP surfaces so, s, sx, sy, sz
class Sphere(Shape) :

   # Data members
   x0 = 0
   y0 = 0
   z0 = 0
   r = 0

   # Methods
   def __init__(self, rad, tag = "", bar1 = 0, bar2 = 0, bar3 = 0) :
      self.r = rad

      if tag == "s" or tag == "S" :
            self.x0 = bar1
            self.y0 = bar2
            self.z0 = bar3

      elif tag != "" :
            if re.search(r'sx', tag, re.IGNORECASE) : self.x0 = bar1
            elif re.search(r'sy', tag, re.IGNORECASE) : self.y0 = bar1
            elif re.search(r'sz', tag, re.IGNORECASE) : self.z0 = bar1
            elif re.search(r'so', tag, re.IGNORECASE) : pass
            else : raise GeoException, "Illegal sphere constructor call"

   def contains(self, x, y, z) :
      dx = x - self.x0
      dy = y - self.y0
      dz = z - self.z0
      LHS = math.sqrt(math.pow(dx, 2) + math.pow(dy, 2) + math.pow(dz, 2))
      return LHS < self.r

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Cylinder class supports MCNP surfaces c/x, c/y, c/z, cx, cy, cz
class Cylinder(Shape) :

   # Data members
   axis = ""
   r = 0
   oc1 = 0
   oc2 = 0
   # Methods
   def __init__(self, tag, rad, offCoord1 = 0, offCoord2 = 0) :
      if offCoord1 != 0 and offCoord2 != 0 and tag.find("/") == -1 :
            raise GeoException, "Illegal cylinder constructor call"

      if re.search(r'[cx/]', tag, re.IGNORECASE) : self.axis = "x"
      elif re.search(r'[cy/]', tag, re.IGNORECASE) : self.axis = "y"
      elif re.search(r'[cz/]', tag, re.IGNORECASE) : self.axis = "z"
      else : raise GeoException, "Illegal cylinder constructor call"

      self.r = rad
      self.oc1 = offCoord1
      self.oc2 = offCoord2

   def contains(self, x, y, z) :
      if self.axis == "x" :
            d1 = y - self.oc1
            d2 = z - self.oc2
      elif axis == "y" :
            d1 = x - self.oc1
            d2 = z - self.oc2
      else :
            d1 = x - self.oc1
            d2 = y - self.oc2
      return math.sqrt(pow(d1, 2) + pow(d2, 2)) < self.r

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Ellipsoid class supports MCNP surface sq but does not check that the
# parameters given truly define an ellipsoid and not a hyper- or para-boloid.
class Ellipsoid(Shape) :

   # Data members
   A = 0
   B = 0
   C = 0
   G = 0
   x0 = 0
   y0 = 0
   z0 = 0

   # Methods
   def __init__(self, xS, yS, zS, gTerm, xBar, yBar, zBar) :
      self.A = xS
      self.B = yS
      self.C = zS
      self.G = gTerm
      self.x0 = xBar
      self.y0 = yBar
      self.z0 = zBar

   def contains(self, x, y, z) :
      xTerm = self.A * pow((x - self.x0), 2)
      yTerm = self.B * pow((y - self.y0), 2)
      zTerm = self.C * pow((z - self.z0), 2)
      return xTerm + yTerm + zTerm + self.G < 0
```

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Functions

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# makeShape() accepts a line from the MCNP input file and returns the
# corresponding Shape object

def makeShape(s) :
   args = re.split(r'\s*', s)
   shType = args[1].lower()
   for i in range(2, len(args)) :
      try : args[i] = float(args[i])
      except ValueError : pass

   # Figure out what type of Shape it is and call the appropriate
constructor.
   if shType == "so" :
      return Sphere(args[2])

   elif shType == "s" :
      return Sphere(args[2], args[1], args[3], args[4], args[5])

   elif shType == "sx" or shType == "sy" or shType == "sz" :
      return Sphere(args[2], args[1], args[3])

   elif shType.find("c/") != -1 :
      return Cylinder(args[1], args[2], args[3], args[4])

   elif shType == "cx" or shType == "cy" or shType == "cz" :
      return Cylinder(args[1], args[2])

   elif shType == "sq" :
      return Ellipsoid(args[2], args[3], args[4], args[8], args[9], args[10],
args[11])

   else :
      raise SystemExit("Error: unsupported surface shape specified: ")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# makeTable() constructs a table for entering angle-specific dose
# distribution information into GAMS.

def makeTable(set, doses) :

   s13 = "             "
   s12 = "            "
   s11 = "           "
   s10 = "          "
   s9  = "         "
   s8  = "        "
   s7  = "       "
   s6  = "      "
   s5  = "     "
   s4  = "    "
   s3  = "   "
   s2  = "  "
   s1  = " "
   tbl = s12


   # Create the labels we need.
   for lab, a in enumerate(doses) :
      tbl = tbl + str(lab) + s12

   tbl = tbl + "\n"

   #Iterate over the voxels in the set.
   for l in set :
      if   l < 10          : tbl = tbl + str(l) + s11
      elif l < 100         : tbl = tbl + str(l) + s10
      elif l < 1000        : tbl = tbl + str(l) + s9
      elif l < 10000       : tbl = tbl + str(l) + s8
      elif l < 100000      : tbl = tbl + str(l) + s7
      elif l < 1000000     : tbl = tbl + str(l) + s6
      elif l < 10000000    : tbl = tbl + str(l) + s5
      elif l < 100000000   : tbl = tbl + str(l) + s4
      elif l < 1000000000  : tbl = tbl + str(l) + s3
      elif l < 10000000000 : tbl = tbl + str(l) + s2
      else : raise SystemExit("Error: Too many voxels")

      #Iterate over angels in the DDM list.
      for a in doses :
            elt = '%e' % a[l]
            tbl = tbl + elt + s1

      tbl = tbl + "\n"

   return tbl

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def run(argv) :

   # Check arguments
   if len(argv) != 4 :
      print "\nUsage: python parse.py inputFile numAngles fluence"
      sys.exit(0)
   if int(argv[2]) < 1 :
      print "\nError: must run at least one angle"
      sys.exit(0)
   mcInp = argv[1]
   numAngs = int(argv[2])
   fluence = float(argv[3]) # photons / cm^2 / s
   srcArea = 100 # cm^2

   # Grab and sort the files we need.
   meshFiles = []
   thisDur = "./"
   for file in os.listdir(thisDur) :
      match = re.search(r'[0-9]m\Z', file)
      if match : meshFiles.append(file)

   if len(meshFiles) == 0 :
      print "Error: couldn't find mesh tally files."
      sys.exit(0)

   meshFiles.sort()
   meshFiles = meshFiles[0:numAngs]

   # Get info about the mesh. (The following is inelegant, but it works.)
   mcText = open(mcInp).read()

   match = re.search(r'iints\s*=\s*(.*?)\s', mcText, re.IGNORECASE)
   if match : I = int(match.group(1))
   else : raise SystemExit("Error in MCNP input file: iints undefined")

   match = re.search(r'jints\s*=\s*(.*?)\s', mcText, re.IGNORECASE)
   if match : J = int(match.group(1))
   else : raise SystemExit("Error in MCNP input file: jints undefined")

   match = re.search(r'kints\s*=\s*(.*?)\s', mcText, re.IGNORECASE)
   if match : K = int(match.group(1))
   else : raise SystemExit("Error in MCNP input file: kints undefined")

   match = re.search(r'imesh\s*=\s*(.*?)\s', mcText, re.IGNORECASE)
   if match : xM = float(match.group(1))
   else : raise SystemExit("Error in MCNP input file: imesh undefined")

   match = re.search(r'jmesh\s*=\s*(.*?)\s', mcText, re.IGNORECASE)
   if match : yM = float(match.group(1))
   else : raise SystemExit("Error in MCNP input file: jmesh undefined")

   match = re.search(r'kmesh\s*=\s*(.*?)\s', mcText, re.IGNORECASE)
   if match : zM = float(match.group(1))
   else : raise SystemExit("Error in MCNP input file: kmesh undefined")

   match = re.search(r'origin\s*=\s*([0-9\s-]*)', mcText, re.IGNORECASE)
   if match :
      coords = match.group(1)
      vals = re.split(r'\s', coords, 2)
      xm = float(vals[0])
      ym = float(vals[1])
      zm = float(vals[2])
   else :  raise SystemExit("Error in MCNP input file: origin undefined")

   dx = (xM - xm) / I
   dy = (yM - ym) / J
   dz = (zM - zm) / K

   # Get info about the rest of the geometry and dose prescription. First,
   # figure out where the lines we need are.

   mcLines = (open(mcInp)).readlines()

   numTreat = 0
   numSens = 0
```

```
tLine = -1
sLine = -1

tRx = 0 # prescribed dose for treatment regions
tImp = 0 # importance of treatment prescription
sLim = 0 # dose limit for sensitive tissue
sImp = 0 # importance of sensitive tissue dose limit
nImp = 1 # importance of sparing normal tissue (1 is default)


# Look for the markers and read the numbers that follow.
for i, x in enumerate(mcLines) :
   if x.find("c TREAT_ORGS") >= 0 :
         try :
               tLine = i + 1
               tTokens = re.split('\s', x)
               numTreat = int(tTokens[2])
               tRx = float(tTokens[3])
               tImp = float(tTokens[4])

         except Error :
               raise SystemExit("Error: bad formatting near TREAT_ORGS")

   if x.find("c SENS_ORGS") >= 0 :
         try :
               sLine = i + 1
               sTokens = re.split('\s', x)
               numSens = int(sTokens[2])
               sLim = float(sTokens[3])
               sImp = float(sTokens[4])

         except Error :
               raise SystemExit("Error: bad formatting near SENS_ORGS")

if numTreat == 0 and numSens == 0 :
   raise SystemExit("Error: no optimization constraints defined")

# Next, read the lines and create the Shapes.

tShapes = []
tCount = numTreat
while tCount > 0 :
   tShapes.append(makeShape(mcLines[tLine]))
   tLine = tLine + 1
   tCount = tCount -1

sShapes = []
sCount = numSens
while sCount > 0 :
   sShapes.append(makeShape(mcLines[sLine]))
   sLine = sLine + 1
   sCount = sCount -1

# Create dose distribution matrices. We'll actually make these
# vector-like, since that will be way easier to pass to GAMS. We'll
# map the mesh point (i,j,k) onto (l) to make the set one-dimensional:
# l = i + I(j-1) + IJ(k-1)
```

```
   DDMs = [] # list of dose distribution matrices
   DDMErrs = [] # list of dose distribution matrices' error

   # A note on units: Now, the units in the mesh tally are rem*s/h/particle.
   # We need to get to a dose rate of Gy/s, so our conversion factor will be
   # the fluence times the beam x-section times some conversion to fix the
time
   # and dose units:

   conv = fluence * srcArea / 3600 / 100

   for mFile in meshFiles :

      fDDM = {}
      fDDMErr = {}

      mLines = (open(mFile)).readlines()
      lNum = 0
      i = 1
      j = 1
      k = 1
      while lNum < len(mLines) :
            if mLines[lNum].find("Tally Results:") != -1 :

                  while j <= J :
                        dTokens = re.split(r'\s*', mLines[lNum + 1 + j])
                        eTokens = re.split(r'\s*', mLines[lNum + 4 + j + J])

                        while i <= I :

                              l = i + I * (j - 1) + I * J * (k - 1)
                              fDDM[l] = float(dTokens[i+1]) * conv
                              fDDMErr[l] = float(eTokens[i+1]) * conv
                              i = i + 1

                        i = 1
                        j = j + 1

                  i = 1
                  j = 1
                  k = k + 1
                  lNum = lNum + 5 + 2 * J

            else :
                  lNum = lNum + 1

      DDMs.append(fDDM)
      DDMErrs.append(fDDMErr)

   # Need to sort mesh points into three different sets: T, S, N (see
   # formulation reference).

   # Sets
   T = []
   S = []
   N = []
   for k in range(1, K + 1) :
      for j in range(1, J + 1) :
            for i in range(1, I + 1) :
                  vPlaced = False
                  l = i + I * (j - 1) + I * J * (k - 1)
                  x = xm + dx / 2 + dx * (i - 1)
                  y = ym + dy / 2 + dy * (j - 1)
                  z = zm + dz / 2 + dz * (k - 1)

                  for sh in tShapes :
                        if sh.contains(x, y, z) :
                              if vPlaced == True :
                                    raise GeoException, "Error: voxel
contained by multiple shapes."
                              else :
                                    vPlaced = True
                                    T.append(l)

                  if False == vPlaced :
                        for sh in sShapes :
                              if sh.contains(x, y, z) :
                                    if True == vPlaced :
                                          raise GeoException, "Error: voxel
contained by multiple shapes."
                                    else :
                                          vPlaced = True
                                          S.append(l)

                  if False == vPlaced : N.append(l)

   # OK, now we've got everything we need to make the GAMS input file. Let's
do
   # it.

   # Copy from the template.
   gStr = open("GAMS_TMPL").read()
   if gStr == "" :
      raise SystemExit("Error: bad GAMS_TMPL file")

   # Now replace the placeholders with the stuff we need.

   # Create the sets.
   TAR_REPL = ""
   for l in T :
      newV = str(l) + ",\n"
      TAR_REPL = TAR_REPL + newV
   TAR_REPL = TAR_REPL[0:len(TAR_REPL)-2]
   gStr = gStr.replace("TAR_SET", TAR_REPL)

   SENS_REPL = ""
   for l in S :
      newV = str(l) + ",\n"
      SENS_REPL = SENS_REPL + newV
   SENS_REPL = SENS_REPL[0:len(SENS_REPL)-2]
   gStr = gStr.replace("SENS_SET", SENS_REPL)
   NORM_REPL = ""
   for l in N :
      newV = str(l) + ",\n"
      NORM_REPL = NORM_REPL + newV
   NORM_REPL = NORM_REPL[0:len(NORM_REPL)-2]
   gStr = gStr.replace("NORM_SET", NORM_REPL)

   ANG_REPL = ""
   for l in range(0, numAngs) :
      ANG_REPL = ANG_REPL + str(l) + ", "
   ANG_REPL = ANG_REPL[0:len(ANG_REPL)-2]
   gStr = gStr.replace("ANG_SET", ANG_REPL)

   # Replace the prescription, dose limit, importance placeholders
   gStr = gStr.replace("THETA_VAL", str(tRx))
   gStr = gStr.replace("PHI_VAL", str(sLim))
   gStr = gStr.replace("TAR_IMP", str(tImp))
   gStr = gStr.replace("SENS_IMP", str(sImp))
   gStr = gStr.replace("NORM_IMP", str(nImp))

   # Replace the table placeholders

   gStr = gStr.replace("TAR_TABLE", makeTable(T, DDMs))
   gStr = gStr.replace("SENS_TABLE", makeTable(S, DDMs))
   gStr = gStr.replace("NORM_TABLE", makeTable(N, DDMs))

   # Write the actual GAMS file.

   gFile = open(mcInp + ".gms",'w')
   gFile.write(gStr)
   gFile.close()

   Tf = open("tar.vxl", "w")
   Sf = open("sens.vxl", "w")
   Nf = open("norm.vxl", "w")
   DDMf = open("norm.ddm", "w")
   DDEf = open("norm.dde", "w")

   pickle.dump(T, Tf)
   pickle.dump(S, Sf)
   pickle.dump(N, Nf)
   pickle.dump(DDMs, DDMf)
   pickle.dump(DDMErrs, DDEf)

   return ((xm, ym, zm), (xM, yM, zM), (I, J, K))
```

Planning module

```
# plan.py

# Runs GAMS on the input file generated by the given MCNP treatment planning
# input file.

import sys, os, shutil, math
```

```
def run(argv) :

   # Check arguments
   if len(argv) != 3 :
      print "\nUsage: python plan.py inputFile numAngs"
      sys.exit(0)
   if int(argv[2]) < 1 :
      print "\nError: must run at least one angle"
      sys.exit(0)
   mcInp = argv[1]
   numAngs = int(argv[2])

   # Run GAMS

   gProg = "gams"
   gArg = mcInp + ".gms"

   (runIn, runOut) = os.popen4(gProg + " " + gArg)
   print runOut.read()

   # Extract weights if they're there. Report errors if they exist.
   gOut = mcInp + ".lst"

   gStr = open(gOut).read()
   if gStr == "" :
      raise SystemExit("Error: bad " + gStr + " file")
   if gStr.find("Optimal solution found.") == -1 :
      raise SystemExit("Could not find optimal treatment plan as specified.")
   gLines = open(gOut).readlines()

   # Define number of lines in a page-break message.
   numBrLines = 9

   # Find the placeholder.
   for i, x in enumerate(gLines) :
      if x.find("---- VAR w  treatment angle weights") != -1 :
            wLines = range(i + 4, i + 4 + numAngs + numBrLines)
            # last term accounts for if there's a page break

   w = {}
   bCol = 23 # "hundred digit" of the weight is here, if this weight is non-
zero
   eCol = 31 # column "just past" the end of the weight, if this weight is n-
z
   pCol = 26 # column where the period is

   for i in wLines :
      lStr = gLines[i]

      try :
            ss = int(lStr[0:1]) #subscript (of w)

            if lStr[pCol-1:pCol+2] == " . " : # means this weight is zero
                  w[ss] = 0
            else :
                  val = lStr[bCol:eCol]
                  w[ss] = float(val)
      except ValueError :
            pass

   # Extract the objective function.

   # Find the placeholder.
   for i, x in enumerate(gLines) :
      if x.find("Objective :        ") != -1 :
            oLine = i


   # Get the value.
   bCol = 22
   eCol = 30
   val = (gLines[oLine])[22:30]
   obj = float(val)

   return (w, obj)
```

Plotting module

```
# plot.py

# Plots the given data given some plotting option.

# Get modules we need
import sys, os, shutil, math, re, string

def run(argv) :

   # Check arguments
   if len(argv) != 8 :
      print "\nUsage: python plot.py inputFile plotOpt doses geo optPar"
      sys.exit(0)
   if argv[2] != "slice" and argv[2] != "hist" :
      print "\nError: bad plot option"
      sys.exit(0)
   mcInp = argv[1]
   plotOpt = argv[2]
   DM = argv[3]
   mmm = argv[4]
   MMM = argv[5]
   IJK = argv[6]
   optPar = argv[7]

   xm = mmm[0]
   ym = mmm[1]
   zm = mmm[2]

   xM = MMM[0]
   yM = MMM[1]
   zM = MMM[2]

   I = IJK[0]
```

```
J = IJK[1]
K = IJK[2]

dx = (xM - xm) / I
dy = (yM - ym) / J
dz = (zM - zm) / K

   # Plot the slice.
if plotOpt == "slice" :

   # Figure out what slice we're supposed to plot.
   pt = optPar
   if pt < xm or pt > xM :
         raise SystemExit("Error: bad option-specific plotting parameter")

   # Open the appropriate .dat file
   gFile = mcInp + "_slice_%*.*f.gdat" % (1, 3, pt)
   mFile = mcInp + "_slice_%*.*f.dat" % (1, 3, pt)
   try :
         os.remove(gFile)
   except OSError :
         pass

   try :
         os.remove(mFile)
   except OSError :
         pass

   gf = open(gFile, 'w')
   mf = open(mFile, 'w')

   # Pull out the data for the slice.

   # Get the nearest index.
   islice = round((pt - xm) / (xM - xm) * I, 0)

   sp = "    "

   # Iterate over the dose matrix to generate vector-form of the data file
   for l in DM.keys() :
         i = (l % (I*J)) % I

         # If we're in the slice, grab the data.
         if i == islice :
               j = ((l - i) % (I*J)) / I + 1
               k = (l - i - I * (j - 1)) / (I*J) + 1

               y = ym + dy / 2 + dy * (j - 1)
               z = zm + dz / 2 + dz * (k - 1)

               yp = "%+*.*f" % (1, 3, y)
               zp = "%+*.*f" % (1, 3, z)

               d = DM[l]

               # Write it.
               gf.write(yp + sp + zp + sp + str(d) + "\n")
   gf.close()

   # Loop over J and K values to generate matrix-form of the data file
   for j in range(1, J+1) :
         for k in range(1, K+1) :
               l = islice + I * (j - 1) + I * J * (k - 1)
               dp = "%e" % DM[l]
               mf.write(dp + sp)
         mf.write("\n")

   mf.close()
```