Optimization of Voxelization Parameters in Geant4 Tracking and Improvement of the Shooter Benchmarking Program

by:
Zachary Siegel

With help and supervision from:
Dr. John Apostolakis

A report submitted to document work done as a 2013 Summer Student at CERN

In association with George Mason University and Dr. Phil Rubin

25 June 2013 - 16 August 2013
# Table of Contents

ABSTRACT ................................................................................................................................. 3  
TERMINOLOGY AND VARIABLE NAMES ................................................................................. 4  
INTRODUCTION ........................................................................................................................ 5  
SOFTWARE DESIGN PROCESS AND BENCHMARKING METHODS .......................................... 6  
RESULTS AND ANALYSIS ....................................................................................................... 11  
CONCLUSIONS AND OPEN QUESTIONS .................................................................................... 15  
APPENDIX A: Using Shooter .................................................................................................. 16  
APPENDIX B: Code of ‘shooter.cc’ .......................................................................................... 18
ABSTRACT

The geometry-based tracking of the ubiquitous particle physics simulation toolkit Geant4 utilizes the idea of voxels, which effectively partition regions into multi-dimensional slices that can decrease simulation time. The extent of voxelization and the size of the voxels is determined by a set of parameters, which until now, defaulted to arbitrary numbers. In this report I document how I tested different values for these parameters and determined which values should be the default. I modified the existing G01 Geant4 example program to get an initial look at how the performance depended on the parameters. Then I modified the Shooter benchmark program, which lacks extraneous physics processes, to collect more refined data and to provide a tool for future testers to perform comprehensive benchmarks. To this end, I created a new geometry, added features to aid in testing over ranges of parameters, and setup the default tests to provide a good sampling of different simulation scenarios.
Volume - Any 3 dimensional shape that exists in the simulated world, which is itself a volume. For example, in the context of high energy particle physics, volumes are used to define the geometry of a detector being simulated. The geometry of a world is represented in the form of a hierarchy, with sub-volumes (called Daughters or Daughter Volumes) contained inside of parent volumes (called Mothers or Mother Volumes).

Voxel - The resulting divisions of a mother volume when applying regular slices to separate daughter volumes by location in order to improve geometry-based tracking. Also called slices.

Max Nodes - Refers to the variable kMaxVoxelNodes, which sets the upper limit for how many slices a mother volume can have along a single coordinate axis.

Min Nodes Lv1 - Refers to the variable kMinVoxelVolumesLevel1, which sets the lower limit for how many daughter volumes a mother volume must have to warrant any voxelization.

Min Nodes Lv2 - Refers to the variable kMinVoxelVolumesLevel2, which sets the lower limit for how many volumes a voxel node must have to warrant another dimension of voxel slices.

Min Nodes Lv3 - Refers to the variable kMinVoxelVolumesLevel3, which sets the lower limit for how many volumes a voxel node must have to warrant a third dimension of voxel slices, if it is already the result of two dimensional slicing.

Ratio - Refers to the variable kDefaultSmartless or Ratio, depending on the context, which sets an upper limit on the density of voxels in a volume. Specifically, the number of voxels created in a volume is the number of daughter volumes multiplied by the Ratio.

Step - Refers to Geant4’s process of determining a particle’s next intersection/scattering point

Navigator - Refers to Geant4’s engine that computes the steps of particles
INTRODUCTION

Geant4 is a complex software toolkit used worldwide to aid in particle physics simulations. It tracks the paths of particles as they move through and between the volumes that define the geometry of the simulated world. Since a computer program has no a priori knowledge of which volumes lie along a particle’s trajectory, intersection points of all volumes must be calculated - the closest intersection will be the next scattering point.

To aid in checking the volumes, voxelization is introduced. With this scheme, a mother volume is divided into equal-sized voxel slices. Each voxel references all the daughter volumes that intersect with the slice. This way, the navigator needs only to check the trajectory of a particle currently within a voxel against the volumes referenced by the voxel. Additionally, a “blocking list” records the volumes that have already been checked, so that they don’t need to be checked again. The checking only needs to be done once per step because the algorithm for finding an intersection with a volume must check the entire volume anyway. Because the voxels don’t represent a physical boundary, voxels that reference the same volumes are collected into “equivalent nodes,” such that a single voxel is used to represent multiple slices that intersect the same volumes (regardless of how the volumes are intersected).

The Geant4 Geometry Manager performs voxelization when it “closes” the geometry, which effectively locks the physical setup and checks to see that all volumes are properly configured and placed. The geometry must be closed before the simulation can be started, therefore the process of building the voxel information is done once at the beginning of running a simulation. Thus, it is preferable to spend more time and processing power during this voxelization step in order to reduce the amount of time spent for tracking each individual particle.
SOFTWARE DESIGN PROCESS AND BENCHMARKING METHODS

At first, I utilized the included Geant4 examples of increasing complexity to understand how the toolkit works. Then I switched to using a benchmarking program written specifically for testing Geant4 geometry processes, called Shooter.

**Example G01/Nav_Vol**

After exhausting the utility of the lower level examples (Basic-B1, Novice-N1,N2,N3,N7), I moved on to the G01 GDML example. GDML stands for Geometry Description Mark-up Language and it is a data format used to store geometrical models, such as those of LHC detectors. G01 was useful because it allows a programmer to load such GDML files into Geant4 for use as simulated worlds. I played around with the code to explore how I could best test voxelization parameters and probe complex geometries. I decided to create my own Geant4 program as a modification of G01, called Nav_Vol, which contained most of my changes to G01 and made use of my modifications of the Geant4 source code.

I noted that all of the relevant voxelization parameters existed as constants in the voxeldefs.hh header file, located in the geometry/management/include Geant4 source folder. I wanted to be able to edit these values on the fly, so that I could change parameters and rerun the tests without having to restart the program and reload all the relevant Geant4 libraries. I accomplished this by incorporating a singleton class in voxeldefs called kVoxelVolumeLimits, which allows one to set and get the various parameters during code execution.

I was also able to look at a realistic geometry: cms.gdml, a model of the Compact Muon Solenoid detector. At my supervisor’s direction, I implemented daughter organization to help deal with the complex hierarchy of volumes existing in this sort of model. I created a short algorithm that would sort the daughters of a volume by their solid entity type (roughly speaking, their shape). The Geant4 navigator can check simple volumes (such as a box, rather than a slice
of a cone) quicker and it can check many of the same type of volume quicker if they are checked sequentially. The purported advantage of sorting the daughters (referred to as organization), is that the navigator is faster in handling a certain check (deals with the ‘safety’ of the location of a track, which allows the navigator to make an educated guess as to where the next step will take the particle). I determined rankings of entity types somewhat arbitrarily, but in general I put simpler polyhedra higher (to be first in a list). I stored these rankings as a mapping in the kVoxelVolumeLimits singleton that could be referenced when sorting.

While a useful tool to advance my knowledge of Geant4, I didn’t pursue testing with the Nav_Vol example for very long, because too many extra processes slowed down the simulations and thus the testing.

**Shooter**

The Shooter benchmark has a simple design. Besides allowing for magnetic fields, it does not simulate any other physical processes that one may encounter in high energy particle physics. This is to reduce the amount of overhead spent on any part of the simulation not directly related to tracking the path of a particle. Moreover, the particles themselves have no physical properties and are better thought of as tracks in the context of this benchmark - the program merely determines where intersections will happen along given trajectories, without computing scattering effects. Shooter’s lack of physics robustness makes it perfect for testing how well Geant4 handles and navigates the geometry of a given setup because most of the time is spent on that.

It originally provided three different detector setups to benchmark with: a simple box and two variations on a calorimeter comprised of a bundle of long cylinders. One version used replicas, which are not relevant to my testing; the version I used, at first, built the calorimeter with many placements of the same physical G4Tubs volume, a generalized tube. This version of
the geometry is labeled ‘CaloLoop.’ Each G4Tubs placement was 2000 mm long and had a radius of 22.5 mm. The placements were arranged into a honeycomb-like structure, which almost had the shape of a box of approximate dimensions 2200 mm x 1900 mm x 2000 mm. However, this homogenous setup wasn’t conducive to accurately testing Geant4’s geometry engine because it did not provide enough variation in the calculations done behind the scenes. Thus, I created a new geometry labeled ‘CaloVaryLoop,’ which introduces a few more types of volumes: G4Para, G4Box, and G4Cons. These respectively represent a generalized parallelsipiped, a rectangular prism, and a generalized section of a cone. I essentially just replaced some of the G4Tubs placements with one of these volumes having approximately equal dimensions. The result is that instead of having 100% G4Tubs as the target volumes in the ‘CaloLoop’ geometry, the ‘CaloVaryLoop’ geometry has 37.5% G4Tubs, 37.5% G4Box, 12.5% G4Cons, and 12.5% G4Para. Below, you can see a rough visualization of this honeycomb-like structure:

![Figure 1. CaloVaryLoop Geometry visualized with OpenGL, QuickTime, and Geant4. Note that not all aspects of the shapes are shown, such as the cylindrical outsides of the tube. This is merely a limitation of the rendering and not a deficiency of the geometry.](image)

However, there is the possibility of too much variability with this setup. When navigating tracks shot in nearly all directions, there was a clear dependence of the simulation time on
direction. Figures 2 and 3 below of a 3d plot show the dependence on the spherical coordinates \( \theta \) (azimuthal) and \( \phi \) (polar).

It was then necessary to determine which directions would have the tracks go through many volumes in order to better stress the Geant4 geometry engine. I found that a single particle shot from the origin transitioned between volumes about 41.17 times on average, and that the standard deviation for the 288 directions I tested was 21.85. I decided that a good test should transition between at least 63 volumes, the average plus the standard deviation and rounded down. I looked at the directions that produced these results and determined that when generating random directions to shoot tracks in, \( \theta \) should sweep through \( \frac{\pi}{3} \) to \( \frac{2\pi}{3} \) and \( \phi \) should sweep through \( -\frac{\pi}{4} \) to \( \frac{\pi}{4} \). This wasn’t based on a strict analysis, but rather a heuristic of which easy-to-use angles (fractions of \( \pi \)) would likely produce the intensive test cases I was looking for, by looking at the following plots.
Figure 3. Both of the above plots are from the same data, with different viewing angles. All (10000) particles were shot from the origin in this case. These are results from 648 directions, by sampling in dθ and dφ.

I identified the location of the peaks and determined that θ should be confined to a small region about ½π and that φ should be confined to a similarly sized region about -¼π. Keep in mind that my goal wasn’t to choose a region that had a uniform intensity, but rather to choose a region that would provide a good sampling of intensities of different scales, these values should only be used for randomized directions since sampled directions are probably intended to include the full range of values.

Regardless, I still needed to add more complexity to the runs by shooting from different sources besides the origin. I hardcoded in the limits of the calorimeter (from the CaloVaryLoop geometry) and generated position vectors in this region as the source of the tracks. This measure was less to provide homogeneity of results, and more to have some inclusion of a variety of scenarios.
RESULTS AND ANALYSIS

All of these results were collected with the Shooter example unless otherwise stated. Moreover, unless it is stated, the simulations used the default voxelization parameters:

- Max Nodes = 1000
- Min Nodes Lv. 1 = 2
- Min Nodes Lv. 2 = 3
- Min Nodes Lv. 3 = 4
- Organization = Off
- Ratio = 2

It is important to note that while a certain value of a parameter may appear to be optimal, it may only result in the best performance when used in conjunction with the CaloVaryLoop geometry used here. It may not have the same effect on any other geometry, especially complex geometries. As such, these results are done more as a proof of concept.

Angular Dependence - $\phi$

![Simulation Time vs. Shooting Angle in Phi with Random Sources](image)

Figure 4.
I ran the Shooter program for the CaloVaryLoop geometry and sampled 5000 different directions with regular increments of $d\phi = 2 \pi \div 5000$, $\theta$ held constant at $\pi$. For each direction, I generated a seeded random location in the calorimeter and made 10000 tracks, recording the time it took for each direction and location. I ran this 10 times with the same seed; I had to throw away the data of one of the runs because it became corrupted. In the plot above, the red is the average simulation time for a single direction (averaged over the 9 data points), the blue is the uncertainty in this average, and the thin green line is the average of the simulation time for all directions.

I interpreted this data to mean that while there may be trends in $\phi$, the average simulation time for a sampling of many different angles would be consistent enough to compare.

**Angular Dependence - $\theta$**

![Figure 5.]

Figure 5.
In order to get data on the $\theta$ dependence, I used the same method as for the $\phi$ dependence. I kept $\phi$ fixed at 0, while I sampled $\cos(\theta)$ angles in the range $-1 - 1$, incrementing by $d\cos(\theta) = 2 \div 200$. I averaged over 10 iterations and the uncertainty is the standard deviation once again.

This plot shows that there is a strong correlation between simulation intensity and the shooting angle, and that a small change in direction could have a large impact on performance.

**Ratio (Voxel Density)**

![Figure 6.](image-url)
For determining how the performance of Geant4 depends on Ratio, I used a similar method to the ones I used for testing the angular dependence. I generated 100000 tracks that I shot over 50 directions, sampled by incrementing dϕ and dcosθ, from randomly generated sources. I repeated this 5 times with the same seeds, and plotted the average for the 5 runs for a single value of the ratio, along with the standard deviation as the uncertainty. In the first plot, I only incremented the ratio by 0.2 to get an idea of which values to look for, and then incremented by 0.05 to obtain the second plot.

Some quick conclusions can be drawn from these results: Ratios in the range 0 - 1.6 are likely inefficient choices, while Ratios in the range 1.75 - 2.5 and 4 - 4.5 will likely result in better performance. I did not scan higher values of the Ratio, mostly because of time constraints. I can speculate that higher values are less efficient when dealing with complex geometries, such as modeled detectors, since much more memory would be required to handle the increased voxel density.
CONCLUSIONS AND OPEN QUESTIONS

I believe the main result of my work is the improvement and modification of the Shooter program, as it will likely allow future computer scientists and physicists to easily test the performance of Geant4’s geometry handling. In fact, it would not be too difficult to modify the program so that other geometries can be used and loaded, such as by implementing reading from GDML files. Then, the Shooter program could be used to determine the optimal parameters for that specific geometry, for use in actual simulations.

An issue that I think is somewhat unresolved is that of the CaloVaryLoop geometry. As a test case, it is an improvement over the default CaloLoop geometry because it adds variation. However, I could not easily determine which shooting directions and sources would provide the best test cases. Moreover, Shooter could benefit from having a new test geometry with even more variation in shapes. Having only four is of course not very representative of a complex real-world model.

Regarding the variables tested, more work needs to be done to get useful results for some of the parameters. For example, testing the Max Nodes variable is almost useless with these geometries because the limit will only be reached if it is set very low, in which case the voxelization is not at all representative of what optimizations would exist in a geometry with many orders of magnitude more volumes. The same is true for the Min Nodes variables, since the voxels are all fairly similar and increasing this lower bound to a point that would cause a change in the extent of voxelization would effectively eliminate any voxels. So these variables only allow one to test geometries in ‘all or none’ scenarios. That being said, the test results I’ve presented for varying the Ratio are quite telling, since the performance impact is noticeable for different values, but isn’t ‘all or none.’ However, the optimal values here may be very different from the optimal values for a different geometry.
APPENDIX A:

Using Shooter

For a default configuration, the GNUmakefile will place the shooter executable in the following folder: ‘$G4INSTALL/Gmake-Workdir/bin/Shooter/$COMPILER/’ where $G4INSTALL is the folder containing the Geant4 installation files and $COMPILER is the c++ compiler installed on the system (ie. Linux-g++). Running shooter will produce the following output before any simulations are done:

Options (as arguments):
- **-event <number_of_events>**
  number of events for the test. Default is 1000000
- **-dirs <number_of_directions>**
  number of directions to shoot in for the test. Default is 10
  if randomization is not on, then the actual number of runs per set is given by:
  
  #dirs * #dirs - #dirs + 1, this samples angles in dphi and dcos(theta), avoiding double counting
- **-randomize <seed 1> <seed 2>**
  shoot in random directions, with the engine seeded by the required inputs
- **-testorg**
  if this option is specified, run all sets a second time with daughter organization on
- **-geom <geometry_type>**
  where <geometry_type> can be:
  - box - simple box (default)
  - calolooop - calorimeter made by a loop of placements
  - calorep - calorimeter made of replicas
  - calovaryloop – calorimeter made by a loop of placements of several different solids
- **-magn <magnetic_field_value>**
  activates magnetic field (value in tesla units). Default is OFF
- **-maxnodes <max_nodes_initial> <max_nodes_final> <max_nodes_increment>**
  runs a set of simulations for Max Nodes set to values in the given range, incremented by the given amount
- **-minnodeslv1 <min_nodes_lv1_initial> <min_nodes_lv1_final> <min_nodes_lv1_increment>**
  see -maxnodes” << G4endl
- **-minnodeslv2 <min_nodes_lv2_initial> <min_nodes_lv2_final> <min_nodes_lv2_increment>**
  see -maxnodes” << G4endl
- **-ratio <ratio_intial> <ratio_final> <ratio_increment>**
  see -maxnodes” << G4endl
As you can see, the shooter executable is usually intended to be used with many command line arguments. An example of a command that one might use to gather data on the simulation time with different ratios is:

```
zack:Linux-g++>./shooter -geom calovaryloop -event 10000 -dirs 100 -randomize 100 200 -testorg -ratio 1.5 2.1 2
```

This will generate 10000 tracks 100 times for each value of the ratio {1.5, 1.7, 1.9, 2.1}, each time shooting in a random direction, from a random source in the CaloVaryLoop geometry. The seeds for the randomization are 100 and 200, and the same set of 100 directions and sources will be used for each value of the ratio. This whole process will be repeated once more with daughter organization turned on. On average, shooting 100000 tracks will take between 1 and 10 seconds. The simulation time scales linearly with the number of tracks, so shooting 100000 tracks with the same settings will likely take between 10 and 100 seconds. You could expect to begin to see output such as the following (with initialization and readme-type material removed):

```
~~~Org on?:0:Ratio:1.5:Max Nodes:1000:Min Nodes Lv1:2:Min Nodes Lv2:3:Result for:100000:Particles,is(in s):4.413:wError:3.99274:and average time per particle:4.413e-05:for # directions:100
```

Various outputs may or may not be suppressed, but they can be easily commented out or modified for easy data extraction.
APPENDIX B:

Code of ‘shooter.cc’

// ********************************************************************
// * License and Disclaimer
// * The Geant4 software is copyright of the Copyright Holders of
// * the Geant4 Collaboration. It is provided under the terms and
// * conditions of the Geant4 Software License, included in the file
// * LICENSE and available at http://cern.ch/geant4/license. These
// * include a list of copyright holders.
// * Neither the authors of this software system, nor their employing
// * institutes, nor the agencies providing financial support for this
// * work make any representation or warranty, express or implied,
// * regarding this software system or assume any liability for its
// * use. Please see the license in the file LICENSE and URL above
// * for the full disclaimer and the limitation of liability.
// *
// * This code implementation is the result of the scientific and
// * technical work of the GEANT4 collaboration.
// * By using, copying, modifying or distributing the software (or
// * any work based on the software) you agree to acknowledge its
// * use in resulting scientific publications, and indicate your
// * acceptance of all terms of the Geant4 Software license.
// ********************************************************************
// $Id: shooter.cc,v 1.1 2007-10-11 13:01:08 gcosmo Exp $
// GEANT4 tag $Name: not supported by cvs2svn $
// shooter - perform test shots.

/*
gmake CXXFLAGS="-g -pg -a -lc_p " CPPVERBOSE=1 for the library

gmake CXXFLAGS="-g -pg -a -lc_p -static " CPPVERBOSE=1 G4TARGET=shooter
shooter
((line by line profiling in detail)
gprof -i -p -q -x -A -l `which shooter` > profile.shooter
(normal profiling)
gprof -p `which shooter` > profile.shooter
*/

#include <stdlib.h>
#include <string.h>
#include <cmath>
#include <limits>
#include "G4ios.hh"
#include "BuildBoxWorld.hh"
#include "BuildCalorimeter.hh"
#include "Shoot.hh"
#include "G4Timer.hh"
#include "G4SystemOfUnits.hh"
#include "G4GeometryManager.hh"
#include "G4PVPlacement.hh"
#include "G4PVReplica.hh"
#include "G4LogicalVolume.hh"
#include "G4Tubs.hh"
#include "voxeldefs.hh"
#include "Randomize.hh"

G4int numShoot;

G4double epsilon = 0.0001; // used when comparing doubles

const G4bool optimise = true;
const G4double x0=1.12343*cm;

G4VPhysicalVolume* BuildReplicaCal(G4Material* Air)
{
    // not used in my tests, so not reproduced here
}

int main(int argc, char *argv[])
{
    G4ThreeVector origin(0,0,0),pMX(-500,0,0);
    G4ThreeVector vx(1,0,0);
    G4ThreeVector vy(0,1,0);
    G4ThreeVector source(0,0,0);
    G4double xloc = 0, yloc = 0, zloc = 0;
    G4ThreeVector shootdir(0,0,0);

    G4VPhysicalVolume* myTopNode = 0;
    G4double Field = 0.*tesla;

    kVoxelVolumeLimits* kvvl = kVoxelVolumeLimits::GetInstance();

    // set up randomization engine for particle generation
    CLHEP::HepRandom::setTheEngine(new CLHEP::RanecuEngine);
    // set the seeds for particle generation
    long* seeds = (long*)calloc(2, sizeof(long));
    seeds[0] = 100;
    seeds[1] = 200;

    // set up voxelization parameters
    enum GeomType { BOX, CALOLOOP, CALOREP, CALOVARYLOOP } GeomType;
    G4int i;
    G4bool useField = false;

    // starting, ending, and incrementing values for voxelization parameters
    // defaults if user does not enter them as command line arguments
    G4int numruns=0;
    G4int maxnodesinit = kvvl->GetMaxNodes(), maxnodesend = maxnodesinit, maxnodesincr = maxnodesinit;
    G4int minnodeslv1init = kvvl->GetMinVolsLevel1(), minnodeslv1end = minnodeslv1init, minnodeslv1incr = minnodeslv1init;
    G4int minnodeslv2init = kvvl->GetMinVolsLevel2(), minnodeslv2end = minnodeslv2init, minnodeslv2incr = minnodeslv2init;
    G4double ratioinit = kvvl->GetMinVolsLevel3(), ratioend = ratioinit, ratioincr = ratioinit;
    G4int orglimit = 0;
for (i=1;i<argc;i++) {
    if ((i < (argc-1)) && & (strcmp(argv[i],"-event") == 0)) {
        sscanf(argv[i+1],"%d",&numShoot);
    } else if ((i < (argc-1)) && & (strcmp(argv[i],"-geom") == 0)) {
        if (strcmp(argv[i+1],"box") == 0) {
            G4bool randomdirs = false;
            GeomType = CALOLOOP;
            numShoot = 1000000;
            numruns = 10;
        /* Command line parsing */
        } else if (strcmp(argv[i+1],"caloloop") == 0) {
            GeomType = BOX;
        } else if (strcmp(argv[i+1],"calorep") == 0) {
            GeomType = CALOREP;
        } else if (strcmp(argv[i+1],"calovaryloop") == 0) {
            GeomType = CALOVARYLOOP;
        } else {
            G4cerr << argv[i+1] << " is not a known geometry (box,caloloop,calorep,calovaryloop)" << G4endl;
            exit (0);
        }
    } else if ((i < (argc-1)) && & (strcmp(argv[i],"-magn") == 0)) {
        sscanf(argv[i+1],"%lf",&Field);
        G4cout << " Mag Field = " << Field << G4endl;
    } else if ((i < (argc-3)) && & (strcmp(argv[i],"-ratio") == 0)) {
        sscanf(argv[i+1],"%lf",&ratioinit);
        sscanf(argv[i+2],"%lf",&ratioend);
        sscanf(argv[i+3],"%lf",&ratioincr);
        G4cout << " Ratio range is: [" << ratioinit << "," << ratioend << "," << ratioincr << "]" << G4endl;
    } else if ((i < (argc-3)) && & (strcmp(argv[i],"-maxnodes") == 0)) {
        sscanf(argv[i+1],"%d",&maxnodesinit);
        sscanf(argv[i+2],"%d",&maxnodesend);
        sscanf(argv[i+3],"%d",&maxnodesincr);
        G4cout << " Max Nodes range is: [" << maxnodesinit << "," << maxnodesend << "," << maxnodesincr << "]" << G4endl;
    } else if ((i < (argc-3)) && & (strcmp(argv[i],"-minnodesl1") == 0)) {
        sscanf(argv[i+1],"%d",&minnodesl1init);
        sscanf(argv[i+2],"%d",&minnodesl1end);
        sscanf(argv[i+3],"%d",&minnodesl1incr);
        G4cout << " Min Nodes Lv. 1 range is: [" << minnodesl1init << "," << minnodesl1end << "," << minnodesl1incr << "]" << G4endl;
    } else if ((i < (argc-3)) && & (strcmp(argv[i],"-minnodesl2") == 0)) {
        sscanf(argv[i+1],"%d",&minnodesl2init);
        sscanf(argv[i+2],"%d",&minnodesl2end);
        sscanf(argv[i+3],"%d",&minnodesl2incr);
        G4cout << " Min Nodes Lv. 2 range is: [" << minnodesl2init << "," << minnodesl2end << "," << minnodesl2incr << "]" << G4endl;
    }
else if ((i < (argc-1)) && (strcmp(argv[i], "-randomize") == 0)) {
    randomdirs = true;
    sscanf(argv[i+1], "%ld", &seeds0);
    sscanf(argv[i+2], "%ld", &seeds1);
    G4cout << " Shooting direction will be random." << G4endl ;
}
else if ((i < (argc-1)) && (strcmp(argv[i], "-dirs") == 0)) {
    sscanf(argv[i+1], "%d", &numruns);
    G4cout << " Number of directions to shoot in = " << numruns << G4endl ;
}
else if ( (i < argc) && (strcmp(argv[i], "-testorg") == 0)) {
    orglimit = 1;
    G4cout << " Testing with organization off and on" << G4endl ;
}
}

(CultureInfo)

G4cout << "**** Navigation Performance Tester - E.Medernach 30.10.00 ****" << G4endl;
G4cout << " Based on original benchmark test by P.Kent" << G4endl << G4endl;

G4cout << "Options (as arguments):" << G4endl
    << "-event <number_of_events>" << G4endl
    << " number of events for the test. Default is 1000000" << G4endl
    << "-dirs <number_of_directions>" << G4endl
    << " number of directions to shoot in for the test. Default is 10" << G4endl
    << " if randomization is not on, then the actual number of runs per set is given by:" << G4endl
    << " #dirs * #dirs - #dirs + 1. this samples angles in dphi and dcos(theta), avoiding double counting" << G4endl
    << "-randomize <seed 1> <seed 2>" << G4endl
    << " shoot in random directions, with the engine seeded by the required inputs" << G4endl
    << "-testorg" << G4endl
    << " if this option is specified, run all sets a second time with daughter organization on" << G4endl
    << "-geom <geometry_type>" << G4endl
    << " where <geometry_type> can be:" << G4endl
    << " box - simple box (default)" << G4endl
    << " caloloop - calorimeter made by a loop of placements" << G4endl
    << " calorep - calorimeter made of replicas" << G4endl
    << " calovaryloop - calorimeter made by a loop of placements of several different solids" << G4endl
    << "-magn <magnetic_field_value>" << G4endl
    << " activates magnetic field (value in tesla units). Default is OFF" << G4endl
    << "-maxnodes <max_nodes_initial> <max_nodes_final> <max_nodes_increment>" << G4endl
    << " runs a set of simulations for Max Nodes set to values" << G4endl
    << " in the given range, incremented by the given amount" << G4endl
    << "-minnodesl1 <min_nodesLv1_initial> <min_nodesLv1_final> <min_nodesLv1_increment>" << G4endl
    << " see -minnodes" << G4endl
    << "-minnodesl2 <min_nodesLv2_initial> <min_nodesLv2_final> <min_nodesLv2_increment>" << G4endl
    << " see -minnodes" << G4endl
    << "-ratio <ratio_initial> <ratio_final> <ratio_increment>" << G4endl
    << " see -maxnodes" << G4endl
    << G4endl;

// Build the geometry
G4cout << "Geometry type:" << G4endl;
switch (GeomType) {
    case BOX:
        G4cout << " Box only." << G4endl;
        myTopNode=BuildBoxWorld();
        break;
    case CALOLOOP:
        G4cout << " Calorimeter made of placements." << G4endl;
        myTopNode=BuildCalorimeter();
        break;
}
break;

case CALOREP:
    G4cout << " Calorimeter made of replicas." << G4endl;
    myTopNode=BuildReplicaCal();
    break;

case CALOVARYLOOP:
    G4cout << " Calorimeter made of various different placements." << G4endl;
    myTopNode=BuildVaryCalorimeter();
    break;
}

if (!myTopNode)
    exit(1);

G4GeometryManager* geoman = G4GeometryManager::GetInstance();

if (!useField)
{
    G4cout << "-- Magnetic Field is disabled !" << G4endl ;

    //Setting up variables for runs
    //----------------------------------------
    G4double avgtime, errtime;

    G4int actualnumruns = randomdirs ? numruns : numruns * numruns - numruns + 1;
    //array of simulation times for a run with a single set of parameters
    G4double* times = (G4double*)malloc(actualnumruns*sizeof(G4double));

    //----------------------------------------
    //Simulation loops - tests ranges of voxelization parameters
    //----------------------------------------

    //loop over having organization on and off
    for(G4int org = 0 ; org <= orglimit ; org++){
        kvvl->SetOrganization(org);

        //loop over Max Nodes values
        for(G4int maxnodes = maxnodesinit; maxnodes <= maxnodesend ; maxnodes+=maxnodesincr){
            kvvl->SetMaxNodes(maxnodes);

            //loop over Min Nodes Lv. 1 values
            for(G4int minnodeslv1 = minnodeslv1init; minnodeslv1 <= minnodeslv1end ; minnodeslv1+=minnodeslv1incr){
                kvvl->SetMinVolsLevel1(minnodeslv1);

            }

            //loop over Min Nodes Lv. 2 values
            for(G4int minnodeslv2 = minnodeslv2init; minnodeslv2 <= minnodeslv2end ; minnodeslv2+=minnodeslv2incr){
                kvvl->SetMinVolsLevel2(minnodeslv2);

            }

            //loop over Ratio values
            for(G4double ratio = ratioinit; ratio < ratioend + epsilon ; ratio+=ratioincr){
                kvvl->SetDefaultSmartless(ratio);

                // Resets voxels to apply the voxel parameters
                geoman->OpenGeometry();
                geoman->CloseGeometry(true);

                CLHEP::HepRandom::setTheSeeds(seeds);
                avgtime=0;
                errtime=0;

            }

        }

    }

}
G4double jlim = (randomdirs ? 1 : numruns);
for(int j = 0; j < jlim; j++){

G4int ilim = (!randomdirs && !j) ? 1 : numruns;
for(int i = 0; i < ilim; i++) // runs one time w/ i=0 if randomdirs are on

G4double phi, costheta, sintheta;

// randomly generated particle source
xloc = 1800 * CLHEP::mm * (G4UniformRand() - .5);
yloc = 2000 * CLHEP::mm * (G4UniformRand() - .5);
zloc = 2100 * CLHEP::mm * (G4UniformRand() - .5);
source = G4ThreeVector(xloc, yloc, zloc);

if(randomdirs)
{
    costheta = 2 * G4UniformRand() - 1.; // generates z component of direction between -1 and 1
    phi = .00011 * zloc + 1.3 + G4UniformRand() * .4; // generates polar angle in a rhombus skewed in z-axis
}
else
{
    costheta = 2.*i/(G4double)numruns - 1.;
    phi = 2. * j * CLHEP::pi/(G4double)numruns;
}
sintheta = sqrt(1. - costheta * costheta);
shootdir = G4ThreeVector(sintheta*cos(phi),
                         sintheta*sin(phi),
                         costheta);
G4cout << "Shooting from " << source << " along " << shootdir << G4endl;
times[i] = Shoot(numShoot, myTopNode, source, shootdir);
avgtime += times[i];
}

// calculating statistics
avgtime/=actualnumruns;
for(int i = 0; i < actualnumruns; i++)
    errtime += pow(times[i] - avgtime, 2);
errtime = sqrt(errtime/(actualnumruns-1));

G4cout << "~~~Org on?:" << org << ":Ratio:" << ratio << ":Max Nodes:" << maxnodes
      << ":Min Nodes Lv1:" << minnodesLv1 << ":Result for:" << numShoot
      << ":Particles:" << avgtime << ":wError:" << errtime
      << ":and average time per particle:" << avgtime/numShoot
      << ":for # directions:" << actualnumruns << G4endl;
}

G4GeometryManager::GetInstance()->OpenGeometry();
return EXIT_SUCCESS;