Marek Gayer

Unified Solids Documentation

We recommend reading this documentation together with sources of Unified Solids. These sources are well structured and readable and will definitely help to understand the algorithms described in this document.

Contents

Multi-Union 3
Construction 3
Voxelization of Multi-Union 3
Inside 5
DistanceToIn (p, v) 6
DistanceToOut 6
SafetyFromInside 7
SafetyFromOutside 7
Normal 7
Tessellated Solid 8
Construction 8
Voxelization of Tessellated Solid 8
Voxelizator initialization steps for Tessallated solid: 8
Inside 9
Normal 10
SafetyFromInside 10
SafetyFromOutside 11
DistanceToIn 11
DistanceToOut 11
Specific Polycone 12
UPolyconeSection 12

- Inside method algorithm 13
- InsideSection(int index, const UVector3 &p) const algorithm 13
- DistanceToIn 13
- DistanceToOut 13
- Normal 14
- SafetyFromOutside 14
- SafetyFromInside 14

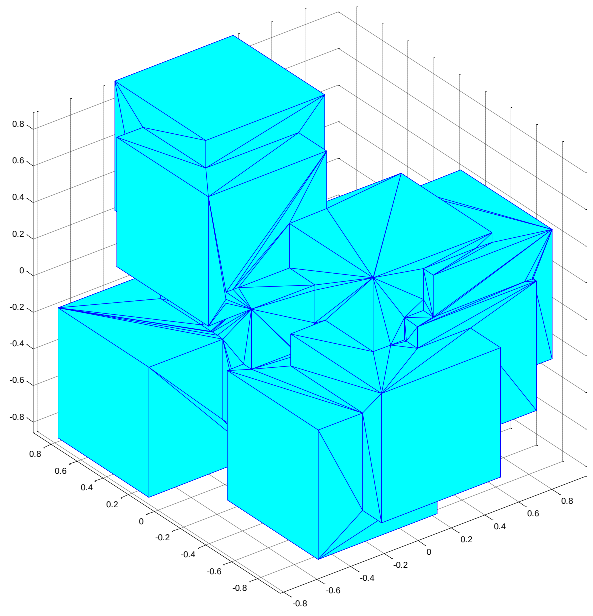
Polyhedra and Polycone (generic) – UVSCGFaceted algoritgms 15

- void UVCSGfaceted::InitVoxels(UReduciblePolygon &rz, double radius) 15
- Inside method 16
- DistanceToIn(p,v) 16
- DistanceToOut(p,v) 16
- Normal 16
- SafetyFromInside 17
- SafetyFromOutside 17

Unified SBT 18

- sbtplot (method, software1, [software2] [first] [count] [color]) 18
- sbtplotall(software1, software2, first, count) 19
- sbtplotallone(software1, software2, first, count) 19
- sbtplot3d(method, software1, software2, first, count) 19
- sbtgenpolycones.m and sbtgenpolyhedra.m 20
- sbtscale 21
- sbtscalability(n) / sbtperfall(n) 21
- sbtvectors(method, name, nameValues1, nameValues2, first, count, color) 22
- sbtperf(scale) 24
- sbtdifferences.m, sbtdot.m, sbtplotpart.m, sbtpoints.m 25

Multi-Union



A solid based on voxelization technique. Quite good introduction, especially for an overview of voxelization, how the parts of solids are put into boundaries, and how these boundaries forms the grid and how the bitmasks are created might be found also in the French report of Jean-Marie Guyader. But since then lot of things changed, including also names of classes, methods, fields. A very important field of this solid is UVoxelizer field voxel, which manages voxelization and is shared between *UMultiUnion* and *UTessellatedSolid*.

Construction

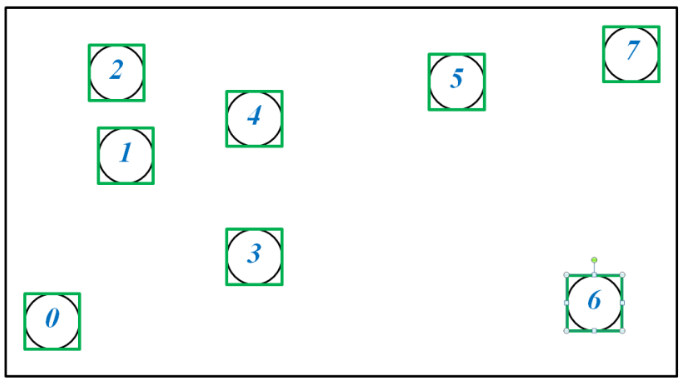
The multi-union is created by its constructor. *AddNode* method than adds it's components. It accepts reference to solid and its transformation (*UTransform3D* class). When all nodes are added, it is necessary to call method *Voxelize* (note this could be renamed to e.g. *SetSolidClosed* to be consistent with tessellated solid). Than it is possible to use the solid as a normal solid.

Voxelization of Multi-Union

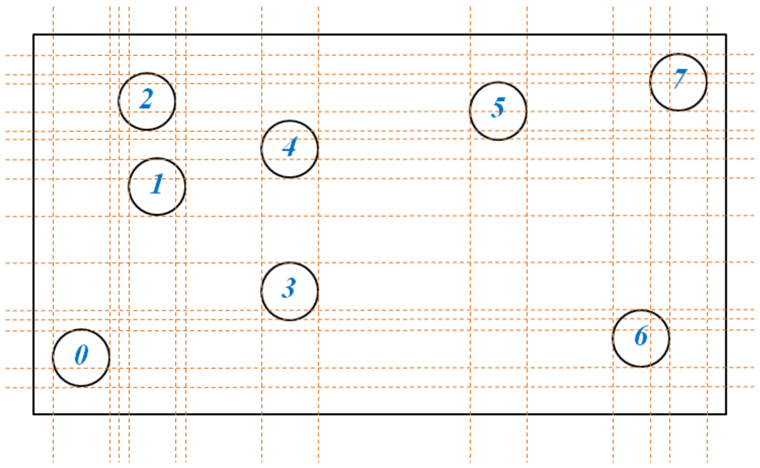
1. Step – create bounding boxes with positions and half-lengths of each solids. Tolerance is added to these. The corresponding method which does that is *UVoxelizer::BuildVoxelLimits*. These boxes are stored in *std::vector<UVoxelBox> fBoxes*.

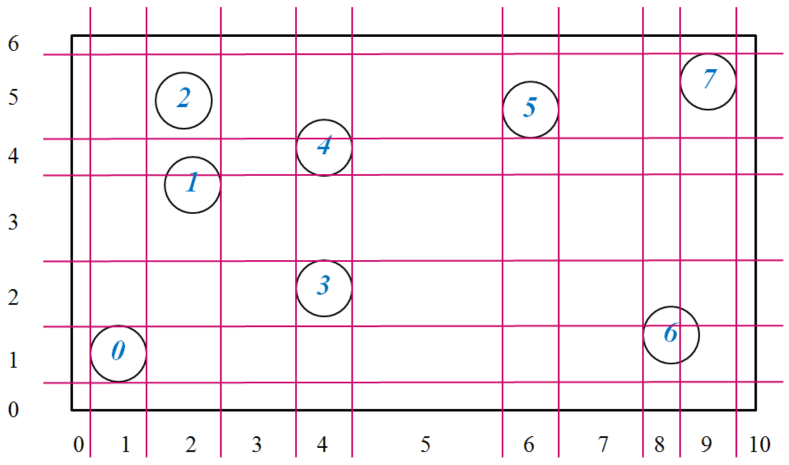
```
struct UVoxelBox
{
      UVector3 hlen; // half length of the box
      UVector3 pos; // position of the box
};
```

This picture illustrates what was done at this step:



2. Step – From boxes mentioned above, sorted boundaries for x, y, z coordinates are created. These boundaries are reduced, for example in case the difference between boundaries would be very small (tolerance / 100). At this step, we also make sure that we do not have more than some very large number and more than sufficient number of boundaries (e.g. 100.000). There could be more reducing to be done, but in case of multi-union we currently did not find it as a priority or that it would help us much. So only this initial reducing of boundaries is made. Corresponding method is *BuildBoundaries.* This picture illustrates what was done at this step, before and after the reduction:

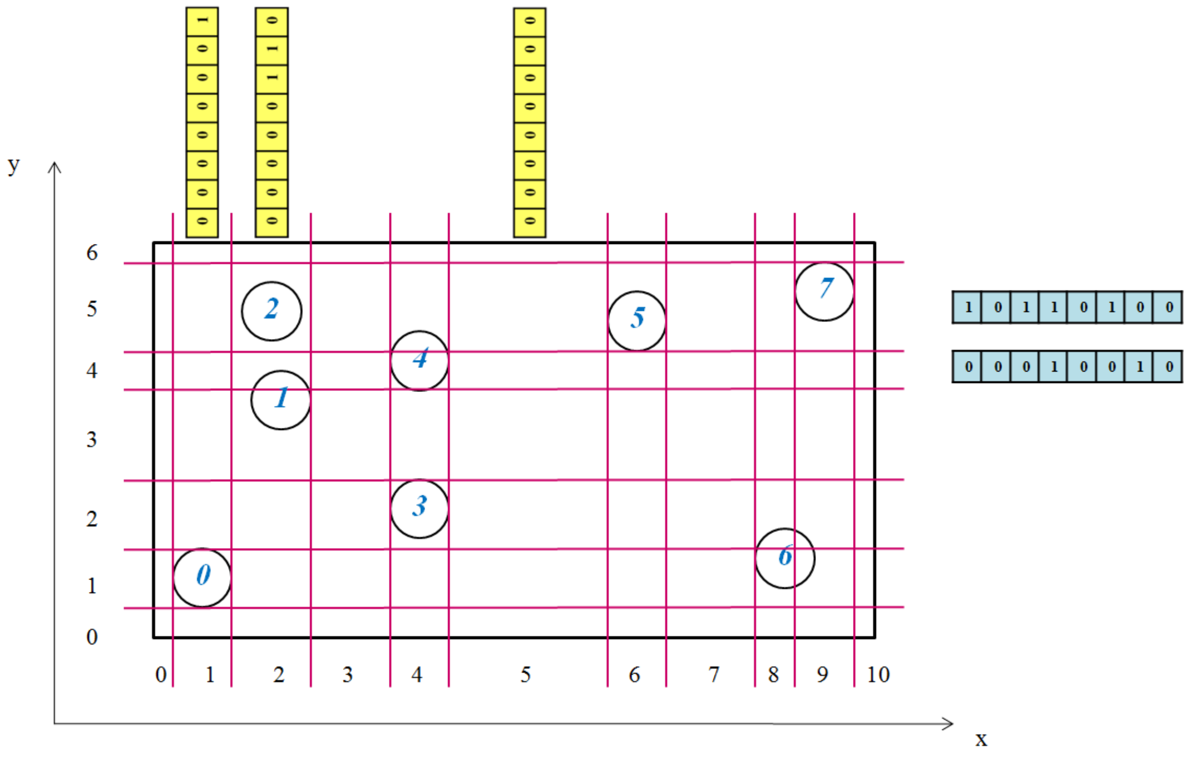




3. Step – We build bitmasks which will assign for ranges created in previous steps bitmasks. If in the range is solid with its number as n, we will set n-th bit in this case. Note that for each of the x, y and z direction we have just one bitmask array. This is done because storing separate array for each of the range would bring more memory requirements, as well as performance penalties.

   To make it fast we go as follows:

   For each of the boxes, we determine where it's minimal position (e.g. left for x-axis) in the ranges. We use **binary search** to quickly find an index of the corresponding segment. We set the appropriate bit depending on the index (multiplied by the total number of solids) and on the *n* of the solid forming the union. The picture corresponding to what was done at this step:



After this, the multi-union is prepared for voxelized algorithms in methods such as Inside.

Inside

Voxelized method is presented. Note that there also exists method *InsideNoVoxels*, which uses the old classical algorithm. It is useful especially for debugging purposes and for comparison of performance.

1. Candidates numbers of the voxel in which the queried point is located are retrieved (via *GetCandidatesVoxelArray*)
2. The point is transformed using the transformation corresponding to candidate indexes

3. Appropriate solid is referenced based on candidate index. We call *Inside* on that solid. If *eEnside* is returned we return with e*Inside*
4. In case, we are on a surface, we store point and solid in an *std::vector* array
5. When all candidates were processed, we check surface points array in an attempt to find two solids which would give normals just going in opposite directions. This would mean we would return *eInside*, since the point is between two surfaces
6. Finally, after processing all candidates, if no surfaces were found, we are outside, otherwise we are on surface

DistanceToIn (p, v)

Voxelized version presented. Code is brief and readable with logical parts moved to separate methods.

1. We will first find distance to the first voxel (we move to the bounding box of voxels)
2. *DistanceToFirst* method does this
3. This can return infinity; in this case we can return with infinity
4. We move the point *p* by given direction *v* with the computed shift
5. The result of this method is minimal distance computed repeatedly by *DistanceToFirst* method
6. Indexes of current voxel for x, y, z coordinates are obtained using binary search by *BinarySearch* method
7. A loop for “flight” through voxels, trying to find a first component that would be collisioned.
8. For this first list of candidates in current voxel is obtained *GetCandidatesVoxelArray*
9. Special method *DistanceToInCandidates* with the list of candidates is invoked. This goes through the list of candidates in the current voxel calling *DistanceToIn* for all the given candidates, also using its spatial transformations. Exclusion bitmask which is also passed to this method is used to make sure that computation is fast, by not examining solids which were already examined by this method.
10. Minimum distance is being kept. If such distance is less than *shift* of the point, we can return with such distance
11. Distance (*shift*) needed to move the point to the border is determined using special method *voxels.DistanceToNext*. If the returned value is infinity, it means we moved out of the voxelized area. At this point it would mean we did not find any solid which would be found collided by the ray, so we can return infinity (the cycle will break as well).
12. Again, if already found minimal distance is not greater than *shift* we can early exit with the collected minimal distance
13. Coordinates of current voxel are updated. If we would find we are at the end of voxelized area, it would mean we can return the minimal collected distance, otherwise we continue at the step 8.

DistanceToOut

1. List of candidates is obtained via *GetCandidatesVoxelArray*
2. In case the returned list is empty we return 0, because it means we are outside
3. We will go through the list of candidates indexes, obtaining the solid and transformation for these

4. *DistanceToOut* is called for points that are not outside. The largest distance, it's corresponding candidate index and the corresponding normal parameter are being kept
5. If some of the points were found not outside from the previous step:
    a. Normal output parameter is filled by using global transformation
    b. Accumulating distance with maximum distance from previous step
    c. Shifting the point with that maximum distance
    d. Check if we are in another solid, by calling Inside excluding the current candidate
    e. Returning accumulated distance if it will be found we are outside
    f. Get new candidates and go to step 3
6. Return the accumulated distance

SafetyFromInside

1. Candidates from the current voxel are obtained (because we are Inside there would be always something)
2. Transformed solid based on a candidates index is obtained
3. Inside is called is called for that solid
4. If we are Inside of such solid, we update the minimal safety with the one we will receive by calling *SafetyFromInside* of such solid

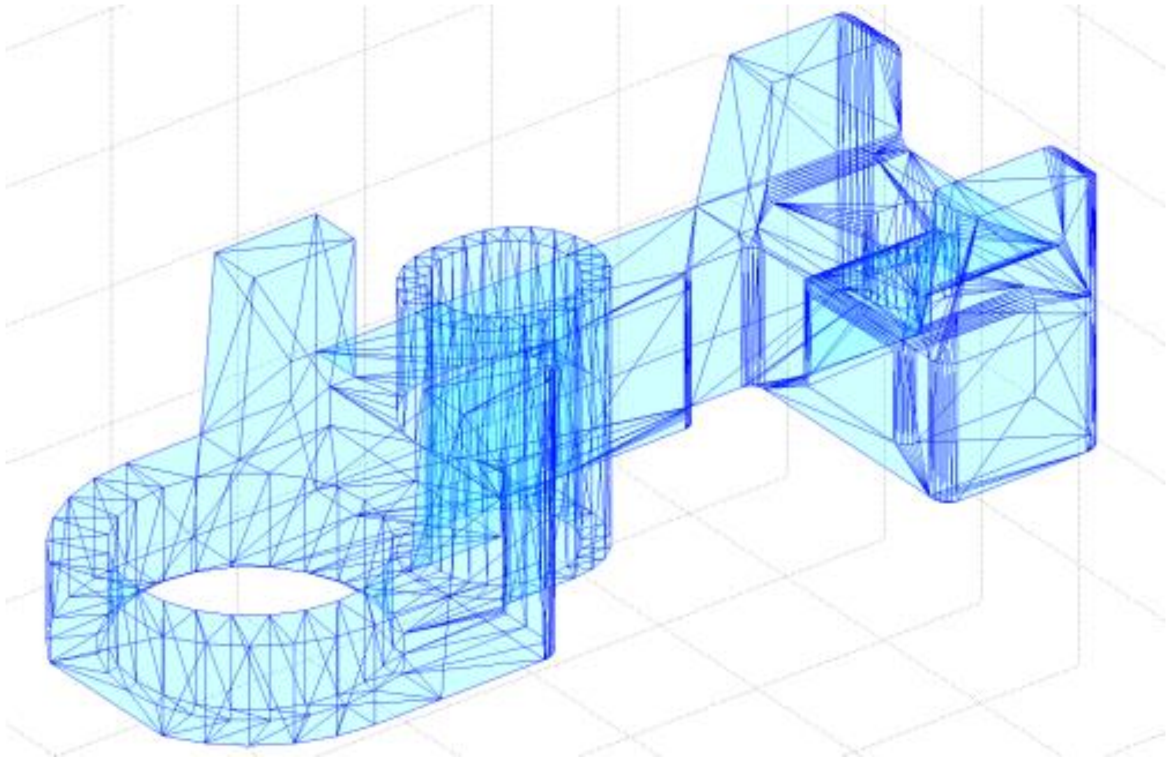
SafetyFromOutside

1. All solids components are checked:
    a. If the distance to bounding box or one of its x,y,z part of a given solid component is less than the minimal safety obtained so far, we can skip checking *SafetyFromOutside* for that solid
    b. Otherwise we have to call *SafetyFromOutside* with transformed point
    c. If we received safety smaller than current minimum, update the minimal safety
2. Returned the minimal safety

Normal

1. We determine if there are any candidates in the current location of point (this way we will also make sure we are at voxelized area)
2. If there are such candidates:
    a. We traverse through the candidates asking for *Inside* status
    b. If we are on surface (this would be the most typical case), we can return the normal of that solid (transformed from local coordinates)
    c. In cases we are not, we are collecting the smallest safety and index of the candidate
    d. This index than would be used, if no solids was found where given point would be on surface. We would then call *Normal* method for such solid
3. If there were no such candidates, we will find a solid in union with the smallest safety (special method for this *SafetyFromOutsideNumberNode* is used)
4. We will return normal for solid returned from previous steps. Transformations to local point to be examined as well as transformation of resulted normal to global coordinates of multi-union is used

Tessellated Solid



- Made from connected triangular and quadrangular facets forming solid
- Old implementation was slow, no spatial optimization
- We use spatial division of facets into 3D grid forming voxels (i.e. we are "voxelizing")
- Voxelizer is based on bitmasks logical and operations during initialization and on pre-calculated list of facets candidates during runtime

Construction

- The constructions complies with the way how it is already described in the documentation for multi-union. Method *SetSolidClosed* invokes the voxelization

Voxelization of Tessellated Solid

- Voxelization is similar as for the multi-union
- But is more complex and needs some additional steps not used for multi-union
- A very important difference is that during the runtime, pre-calculated lists of candidates are being used instead of bitmasks (which are still used but only during initialization)
- The voxelize method accepts *std::vector* of facets
- Voxelization is not used for number of triangles less than 10

Voxelizator initialization steps for Tessallated solid:

1. Voxel limits – based on creating bounding boxes of facets in the similar way how it was done for the **multi-union step 1**.
2. Boundaries are created in the same way as for the **multi-union step 2.**

3. Building bitmasks starts, but parameter NULL is given for bitmasks, which means we will not be storing bitmasks, instead only count of candidates in each of the segments will be evaluated. These counts are used in later steps for reducing (merging) number of ranges
4. Reduction ratios for *x,y* and *z* axis are being calculated. Either from maximum number of voxels, which user can set or from ratios (user currently cannot use these)
5. The voxels are reduced, based on the computed ratios. There are two algorithms that can be used *BuildReduceVoxels* and *BuildReduceVoxels2*.
    a. First algorithm merges two ranges with lowest number of candidates
    b. Second algorithm merges starting from left to right, until expected average number of candidates is reached
6. After the reduction, bitmasks are built in the same way as for the **multi-union, step 3.**
7. Secondary voxel representation, so called mini voxels are being created. These have minimal density and are used for some of methods (*SafetyFromInside, SafetyFromOutside*)
8. List of candidates are being evaluated, together with an array – a bitmask, which holds the information weather there are any candidates in a given voxel. When C++ will finally offer portable hash map which Geant4 would be allowed to use, than instead of list of candidates (*std::map<int, std::vector<int> > fCandidates*) and binary mask we could just use the hashmap. It would make code more brief and clear, while still very fast.
9. For empty voxels, we pre-calculate weather these are outside or inside. This is based on painter's algorithm using *std::stack* structure for that. The motivation for this is to make it faster and avoid recursion, which would fail for larger structures because of stack overflow.

    After this, the Tessellated Solid is ready for algorithms for methods such as *Inside*.

Inside

Voxelized method is presented. Note that there also exists method *InsideNoVoxels*, which uses the old classical algorithm. It is useful especially for cases when voxelization is disabled (e.g. the number of facets forming the solid is too low or in cases user forced to not to use it by setting number of voxels to zero) for debugging purposes, and for comparison of performance.

1. If we are outside of extent, we return *eOutside*
2. Based on the given point *p* location, we determine coordinates of voxel in which we are located
3. Candidates of given voxel are fetched (using array of empty bits and pre-calculated candidates set in the constructor)
4. If size of these candidates are zero, we will use pre-calculated insides made in constructor. We will receive the corresponding index for the voxel, and use it to access to Boolean variable with these data and based on that return either *eInside* or *eOutside.*
5. For the facets candidates in the current voxel, we will check whether the point *p* would not be found on a surface of one of the facets. If so, we could return with *eSurface* status.
6. What happens next is adaptation and modification of original tessellated algorithm. It is based on shooting random rays. For example, if the ray does not hit any surface it would mean that we are outside.

7. The original algorithm was described as follows: *“The following is something of an adaptation of the method implemented by Rickard Holmberg augmented with information from Schneider & Eberly, "Geometric Tools for Computer Graphics," pp700-701, 2003. In essence, we're trying to determine whether we're inside the volume by projecting a few rays and determining if the first surface crossed is has a normal vector between 0 to pi/2 (out-going) or pi/2 to pi (in-going). We should also avoid rays which are nearly within the plane of the tessellated surface, and therefore produce rays randomly. For the moment, this is a bit over-engineered (belt-braces-and-ducttape).”*
8. We have to find such a ray with direction where the vector is not nearly parallel to the surface of any facet since this causes ambiguities. The usual case is that the angles should be sufficiently different, and there are 20 random directions to select from.
9. We modified the shooting of rays in the way that we only check those facets, which would be candidates in corresponding voxels. We use traversal through voxelized area in the similar way as in *DistanceToIn* and *DistanceToOut.*
10. For these limited facets in the voxel, we are trying to find an intersection between ray and facets, for entering and leaving case. If intersection is found, distances are returned. These values are than used to determine whether we were inside or outside, and returning *eOutside* or *eInside.*
11. For this intersection is being asked for the facet via *facet.Intersect* method, separately for
12. If there are no candidates in the voxel, we use pre-calculated arrays of bits which indicate whether the whole voxel is inside or outside.

Normal

1. For the facets in current voxels, we are computing distance to the facet via *VUFacet::Distance*. If it is smaller than tolerance, we found the facet for which the point is on surface and we call *VUFacet::GetSurfaceNormal* to get the vector normal and we return true and the returned normal
2. Otherwise, we continue checking other facets
3. If no facet is found where the point is on surface (this should not happen, since we expect user will usually provide point on surface), we call *UTessellatedSolid::MinDistanceFacets*:
    a. This function will find shortest distances to all the voxels for the point, sorts them, and then will gradually finds smallest distance for the facets inside them. There is an early exit condition when found distance to facet is smaller than shortest distance to next voxel.
    b. Minimal distance and facet is returned from *MinDistanceFacets*
4. For the facet with minimal distance, we call *VUFacet::MinDistanceFacets*

SafetyFromInside

1. If we are outside of extent, we return 0, which means we were outside
2. For voxelized case, we return value of function *MinDistanceFacet,* already described in our description of *Normal*
3. For non-voxelized case, for all the facets *facet.Distance* is called, and the smallest value is in the end returned

SafetyFromOutside

1. Inaccurate version returns safety to bounding box
2. For non-voxelized case, all facets are checked with *facet.Distance*, minimal distance is returned in the end
3. For voxelized case, we first determine if we are inside of extent of the solid. If so, and we would find that we are in the voxel with 0 faces, we check bitmask where we have stored weather the whole voxel is inside our outside. If we are inside, we return 0
4. Otherwise, we return value from function *MinDistanceFacet*, which we already described

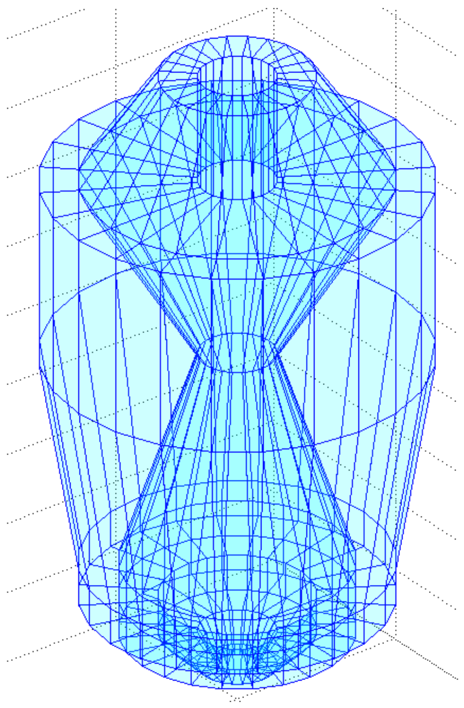
DistanceToIn

1. Distance (shift) from point to the border of first voxel is determined (*UVoxelizer::DistanceToFirst*)
2. If shift is infinity, return infinity
3. Point is shifted with the shift as *current point*
4. Current voxel indices are determined using *UVoxelizer::GetVoxel*. These indices are used to update voxel
5. List of candidates of the current voxel are referenced using *UVoxelizer::GetCandidates*.
6. Minimal distance is set to infinity
7. In case their list is not empty, we call *DistanceToInCandidates*, which will find smallest distance for all the candidates, based on calling *VUFacet::Intersect*. If the returned value is smaller, we update our minimal distance
8. We traverse through voxels as unless *UVoxelizer::DistanceToNext* returns value larger than minimal distance

DistanceToOut

1. Minimal distance is set to infinity
2. If by quick check voxel does not contain point (*UVoxelizer::Contains*) we return 0
3. We get coordinates of the current voxel from point coordinates
4. We get reference to list of candidates of the current voxel
5. In case of list is not empty, we call *DistanceToOutCandidates,* which will find & update smallest distance for all the candidates, based on calling *VUFacet::Intersect*. It will also fill normal parameter for eventual new smallest distance facet and updates found facet index.
6. If the returned value is smaller than distance to next voxel (*shift*) we can break loop
7. We update the distance to next voxel via *UVoxelizer::DistanceToNext*
8. If such shift would return infinity, we break loop as well
9. If we did not find any candidate, we call Normal method to fill normal parameter
10. Otherwise we will fill convex variable by finding facet in list of so called extreme facets

Ordinary Polycone



- Similar to multi-union in a way that core data-structure is an array of UCons/UTubes
- Important concept of section, with just one UCons/UTubes
- Logic of method is usually moved to section segment, which calls version of navigation method (e.g. *Inside, DistanceToIn, DistanceToOut, Normal*) for given section solid
- But does not have overhead of full voxelizer, therefore it's much faster
- Measurements show much better performance than both Geant4/ROOT
- Half of lines of code needed in comparison with ROOT for core methods and more readable
- Type of section (Tubs/Cons) is determined in constructor
- Effort is made to pre-calculate everything in the constructor when possible, such as *shift* in UPolyconeSection

UPolyconeSection

- Important data-structure to keep data for each of the section in organized and clear way.
- The data structure is filled at the constructor

```
  struct UPolyconeSection
  {
    VUSolid *solid;// true if all points in section are concave in regards
to whole polycone, will be determined
    double shift; // this is added to z to the point for given section
    bool tubular; // is it UTubs or UCons?
//    double left, right;
    bool convex; // TURE if all points in section are concave in regards
to whole polycone, will be determined, currently not implemented
  };
   std::vector<UPolyconeSection> fSections;
   std::vector<double> fZs; // z coordinates of given sections (including
left and right borders or whole polycone)
```

- *fSections* is a placeholder for UPolyconeSection, as well as *fZs*; is filled in constructor

Inside method algorithm

The code is very brief and clear and does the following:

1. First find out if we are outside of bounding box (with shifted z in respect to the box), if so early exit with *Outside*
2. We will find in which z-section of polycone we are via *GetSection* method with z-coordinate, which makes binary search (making sure a valid section is always returned)
3. Calling *InsideSection* for given section. If *Inside* early exit with that
4. For points around boundary within tolerance, we have to check also previous / next section for Inside method.
    a. For case, both points on two sections are on surface, check normal, if they goes against, we are still Inside
5. Otherwise, if *InsideSection* reported Surface we are on surface, otherwise outside

InsideSection(int index, const UVector3 &p) const algorithm

1. Find if point *p* is *Inside*, *Surface* or *Outside* with given section
2. The point *p* is shifted, with the shift for given section stored in *fSections*
3. Original implementation called *Inside* method of given solid for the given section
4. New implementation inlines the code, so the performance could be better
5. Minimum and maximum radius of cone / tube at given r is obtained
6. We check if radius is outside radiuses, we return outside in this case
7. If around theses radiuses, we return surface
8. For case of full phi section, the point is than inside, unless it would be found around z borders (left or right), which would return surface
9. For phi section, the angle is determined using atan2 and we find if it is in the phi region, we can be *Inside* or *Surface* at this phase
10. It can be surface at this stage, under the condition that either it's around z borders or around radial tolerance from Phi

DistanceToIn

1. First we check if the ray would hit bounding box and/or enclosing cylinder (experiments might be made which of the options is faster, currently both are present). If it would not hit, we are outside
2. We shift the point with the value obtained from hitting the bounding box; important since it increases performance of this algorithm
3. We get the current section of shifted point with *GetSection*
4. We start to call *DistanceToIn* for the solid in the section, with shift of given section
5. If concrete value is given, we can add this value to the value of shift and return it
6. If infinity is returned, we will continue checking next solid, unless z direction is close to zero or we get away of range of sections we have. The increment is 1 or -1, depending on sign of z

DistanceToOut

1. We get current z-section of point p via *GetSection*

2. We start to accumulate distance to out result from 0
3. We check if we are outside in the current section we return the accumulated distance
4. We call *DistanceToOut* for the solid of current section
5. If zero is returned as result, it means we are already out, we return accumulated distance
6. We increase accumulated distance by the result of *DistanceToOut*
7. Point is shifted by accumulated distance, using direction *v*
8. We will continue checking next solid found in array of sections, unless we run out of allowed range. The increment of corresponding index is 1 or -1, depending on sign of z component of point

Normal

1. Current section is obtained (using *GetSection* in the same way as for *Inside*)
2. If we are close to border, also neighbouring section is obtained, similarly as in Inside
3. If not close to border, we return the value of Normal for given section solid
4. Otherwise, we have to treat surfaces in the similar way as for case of *Inside*
5. False is being set in case when point is found not on surface, but Normal of given section is called to give some vector as result, though it might not be valid

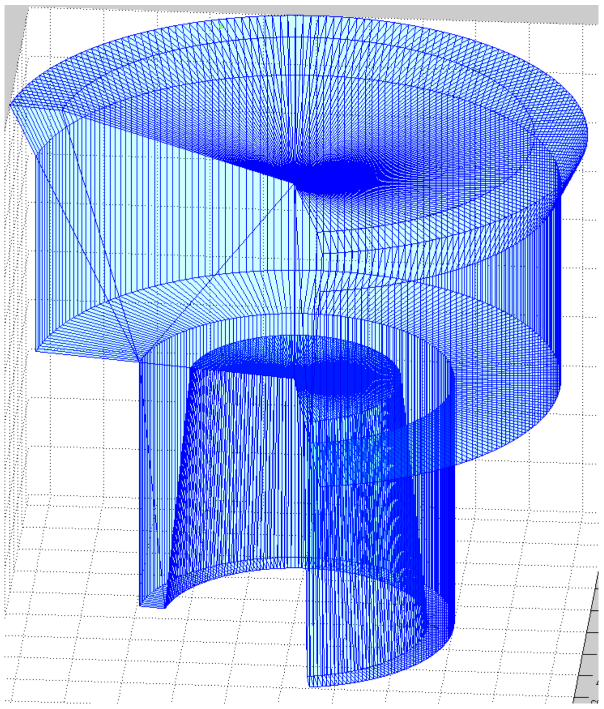
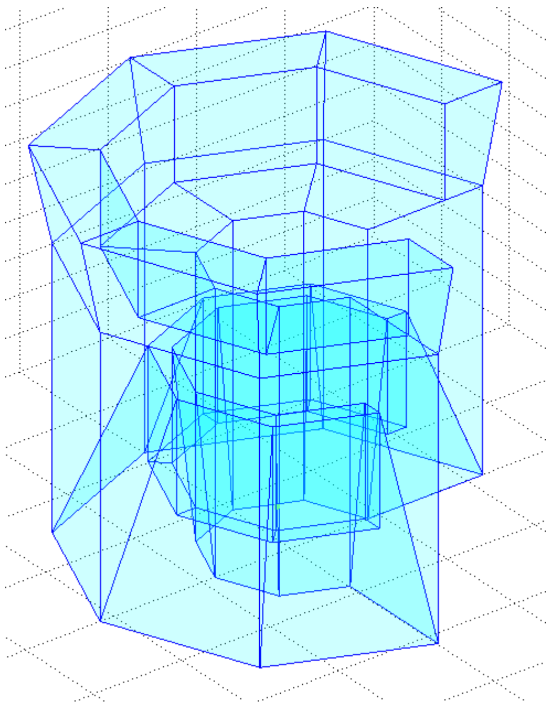
SafetyFromOutside

1. For inaccurate case, we are content with the value of bounding box which is returned
2. For accurate cases, we first get the current section solid *SafetyfromOutside*
3. Next, all sections left and right from current sections are checked, if a lower value of *SafetyFromOutside* can be found. A check whether the minimal safety is smaller than z distance of investigated section from z distance of current section (left or right border respectively), this would mean we can skip checking more
4. Minimal safety is returned

SafetyFromInside

1. Using binary search we will find current z section, if outside we return 0
2. We are calling *SafetyFromInside*, if zero returned early exit
3. The next steps are same as for *SafetyFromOutside*
4. Next, all sections left and right from current sections are checked, if a lower value of *SafetyFromOutside* (note: Safety from Inside cannot be called in this context, because point is not inside, we have to call *SafetyFromOutside* for given section) can be found. A check whether the minimal safety is smaller than z distance of investigated section from z distance of current section (left or right border respectively), this would mean we can skip checking more
5. Minimal safety is returned

Polycone (generic) and Polyhedra – UVSCGFaceted algoritgms

Model is based on facets. Due to its universality it's used **both f**or Polyhedra and for generic polycone. Navigator methods are same for **both** (except *Inside* and *DistanceToIn*, which thanks to overriding first check enclosing cylinder before passing back to UVSCGfaceted implementation)**.** The constructors of these shapes are quite similar than those used in Geant4. Important addition is calling method *InitVoxels* which we will describe bellow.

Most important fields:

```
    std::vector<double> fZs; // z coordinates of given sections
    std::vector<std::vector<int> > fCandidates; // precalculated
candidates for each of the section
    int fMaxSection; // maximum index number of sections of the solid
(i.e. their number - 1). regular polyhedra with z = 1,2,3 section has 2
sections numbered 0 and 1, therefore the fMaxSection will be 1 (that is 2
- 1 = 1)
    mutable UBox fBox; // bounding box of the polyhedra, used in some
methods
    double fBoxShift; // z-shift which is added during evaluation, because
bounding box center does not have to be at (0,0,0)
    bool fNoVoxels; // if set to true, no voxelized algorithms will be
used
```

These store the datastructures necessary for 1D voxelization, i.e. section optimization algorithms.

void UVCSGfaceted::InitVoxels(UReduciblePolygon &rz, double radius)

1. This function is called from UPolyhedra or UPolycone, after the shape is created in *Create* method of these solids. R-Z vertices are copied to local variables.

2. The Z coordinates are sorted and only unique are kept to create sections. In each section, faces which are relevant to this section are detected using *FindCandidates* method and these are stored in the candidates lists. These lists are important for the algorithms which will follow.
3. Bounding box is set up based on smallest and largest z coordinates and radius of shape

Inside method

1. Current section based on binary search from z coordinate is obtained via *GetSection*
2. Loop over index array of candidates faces of current section is being performed
3. Bitmasks are being used to mark faces which were already checked to prevent being checked again
4. Candidate face reference is obtained using index
5. *UVCSGface::Inside* is called. This method returns *VUSolid::EnumInside*, but also distance of the point from the facet. Only smallest value will be in the end taken into account
6. If returned value is surface, method returns surface immediately
7. If the distance is less than the smallest so far, we update smallest
8. We update the bitmasks by the index of the candidate
9. The algorithm than continues with the steps described earlier, by checking sections in the left and right of the current section. The shift on the z-axis is compared to the smallest already found distance of point by obtained from the Inside method. If this z-shift is larger, we can stop checking subsequent left and right sections.

DistanceToIn(p,v)

1. We start calling *DistanceToIn* of bounding box. If box returns infinity, we return infinity
2. We use distance from box to shift the z coordinate of the point closer to the structure
3. We get the current section of the shifted z via *GetSection*
4. We loop through all the candidates in the given sections. Same as with method *Inside*, we also use the bitmasks to avoid checking faces again
5. *UVCSGface::Distance* is called. We take into account it's returned values only if the distance is smaller than the best so far
6. Based on whether the z coordinate of v has + or – sign, we continue checking the sections either on left or right from the current section. If the *v.z* is very close to zero, we do not check more section.
7. We stop checking more sections if shift needed to reach next sections is greater than the best distance obtained from faces so far
8. In the end of the method a check is made if the point is not on the surface

DistanceToOut(p,v)

1. This function is very similar to *DistanceToIn*, with two major differences:
2. We do not shift the point in the beginning as *DistanceToIn* does
3. The conditions in the end of the method to check id the point was already on surface are different

Normal

1. We obtain the current section via *GetSection*
2. We loop trough the candidates, only for the current section

3. We store the *UVCSG::face* result in case returned distance is less than the best so far
4. In the end of method, we store the best answer

SafetyFromInside

1. The current section is obtained using binary search. If we are outside of valid index range we return 0
2. For current segment, minimal safety is obtained through *SafetyFromInsideSection.* This function calls *face.Safety* for each of the faces based on candidates indexes for given section. It also uses bitmasks, which are passed and filled all the time this function is invoked
3. If the current section returned value close to zero, return 0
4. *SafetyFromInsideSection* is called for the sections to the right and then to the left, but only until the minimal safety is greater than the z shift to the border of next section
5. Finally, if the accumulated safety is less than half tolerance return 0, otherwise return the accumulated tolerance

SafetyFromOutside

1. For inaccurate version, estimate is providing by returning *SafetyFromOutside* for bounding box.
2. For accurate version, the original code, based by checking all facets is used. Tatiana could optimize this

Unified SBT

The tests are defined using Geant4 macros. Large collection of *.sbt files provides detailed example of usage of these macros. The folder with data files is created automatically, in the folder where */performance/errorFileName* is located. There are also values.sbt and performance.sbt, which defines ratio of points located inside, outside and on surface, for all the methods measured. Results with output values of each method are stored in datasets. Most important parts of code are located in SBTperformance.cc

The post-processing is based in collection of MATLAB scripts, with file names starting with "sbt"

sbtplot (method, software1, [software2] [first] [count] [color])

Makes 2D plots of points, either from Geant4, Root or USolids software. When providing *software2* parameter, difference is plotted. When running, detailed information is given in the MATLAB console, allowing to copy and paste point and direction vectors. First and count allows to specify a region from which points should start. If count is not given, maximum number of points remaining to the end are assumed. The current directory must be set to the one where SBT datasets are stored.

Example: *sbplot(Inside, Geant4, None, 1000);*

Note: Inside is a function just returning 'Inside', so it's not necessary to use quotes. There are more similar functions like this, see USolids\bridges\G4\SBT\matlab folder.

To omit software2 parameter, None or '' can be passed

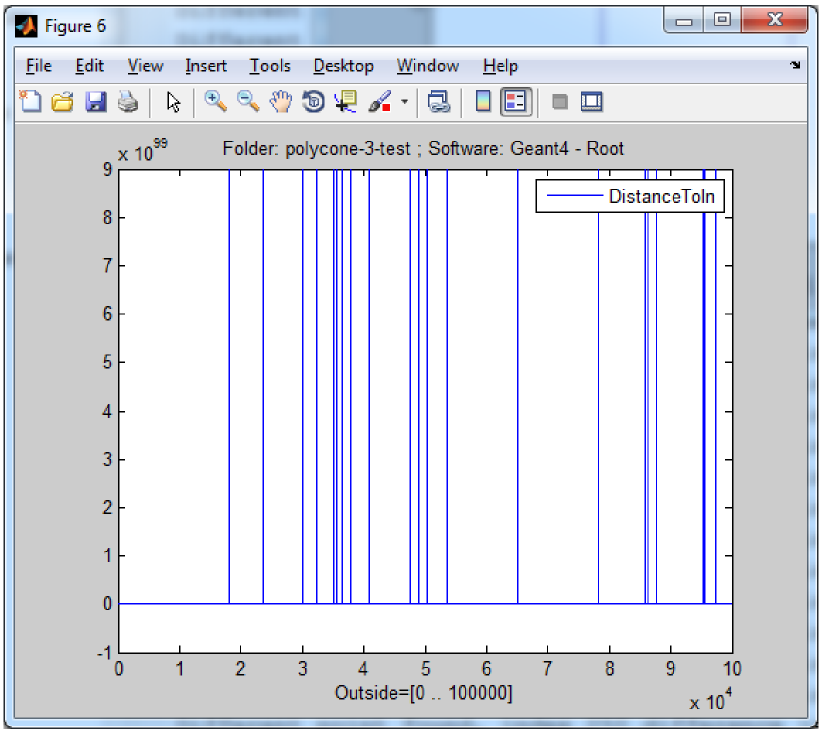


Figure 6
File Edit View Insert Tools Desktop Window Help
Folder: polycone-3-test ; Software: Geant4 - Root
x 10^{99}
DistanceToIn
Outside=[0 .. 100000]
x 10^4

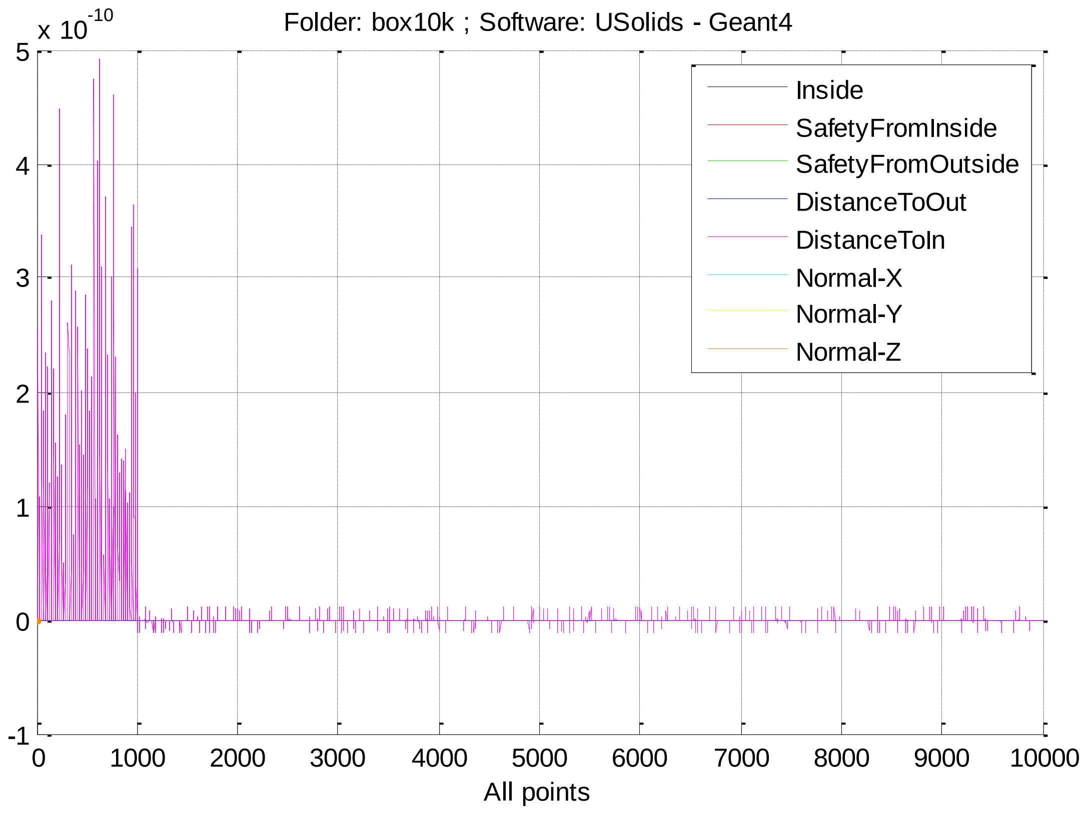
A typical plot obtained using sbtplot

sbtplotall(software1, software2, first, count)

Plots all values or differences if parameter software2 is given. It will test methods *Inside*, *SafetyFromInside, SafetyFromOutside, DistanceToOut, DistanceToIn, Normal.*

sbtplotallone(software1, software2, first, count)

Same as *sbtplotall*, but it puts everything in one plot.

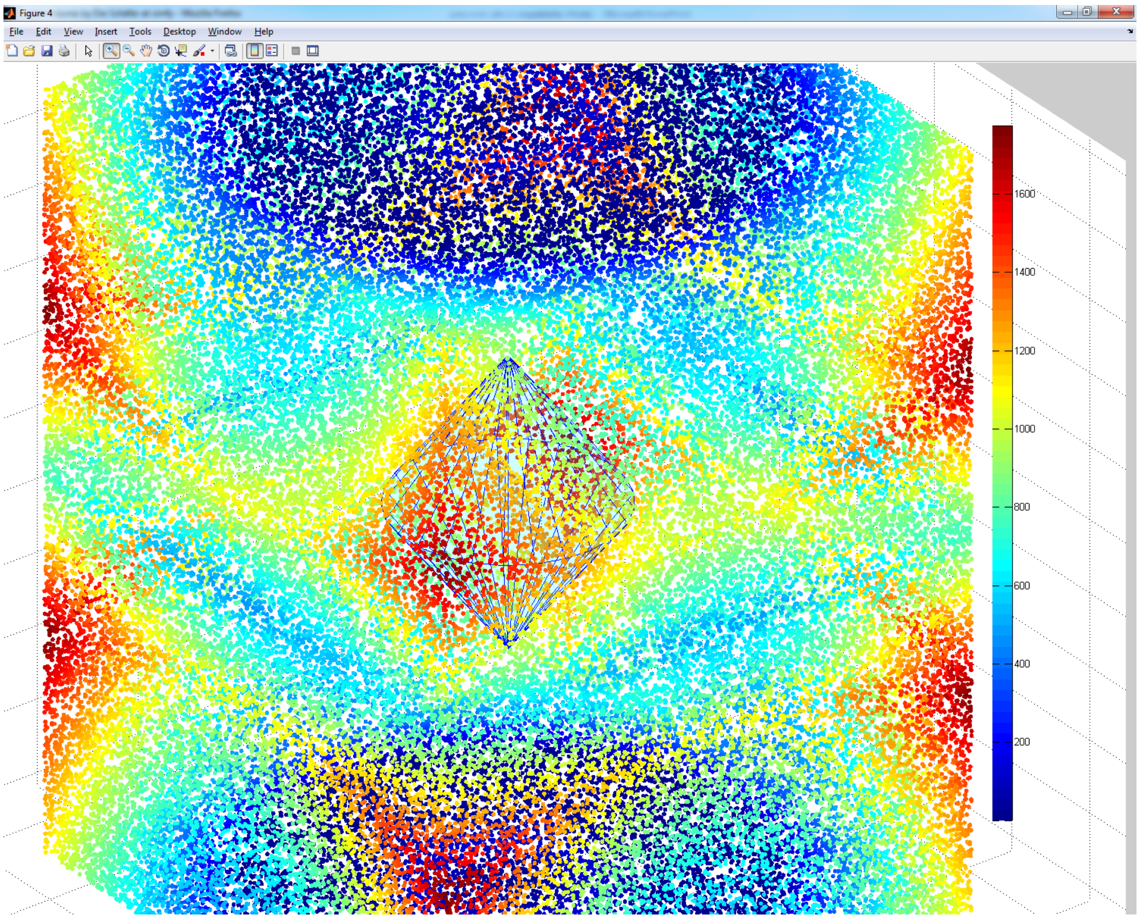


A typical plot obtained using *sbtplotallone*

sbtplot3d(method, software1, software2, first, count)

Similar to sbtplot, but differences to console are not such detailed, and it makes 3D plot instead of 2D one. It will visualize points and polyhedra of a shape, which it gets from SBT datasets.

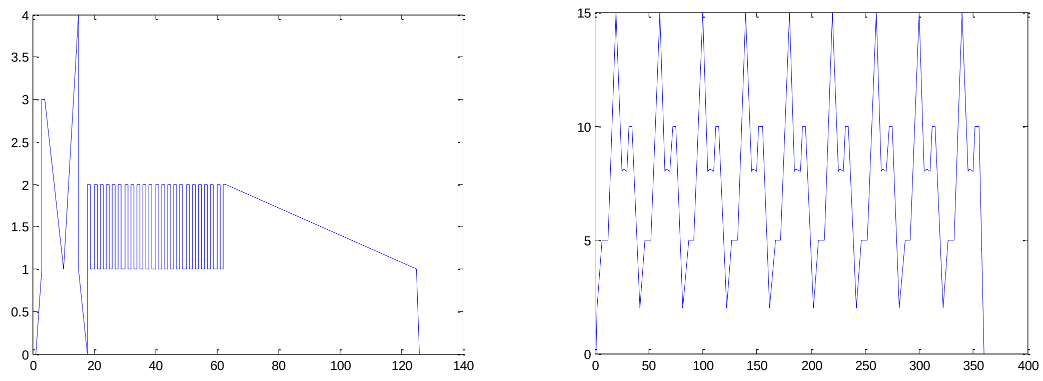
Example: *sbtplot3d(SafetyFromOutside,Geant4,Root,20001);*



A plot obtained using sbtplot3d, notice the polycone shape partially covered by point

sbtgenpolycones.m and sbtgenpolyhedra.m

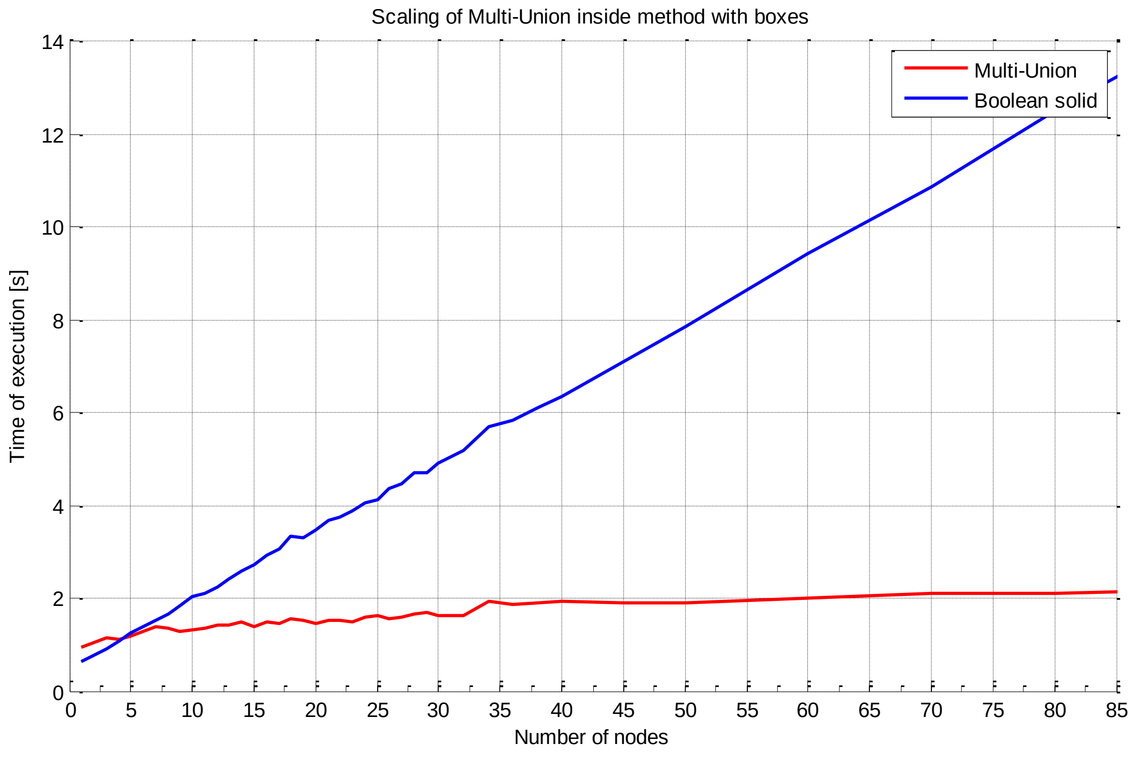
These scripts creates content for .mac files for polycone and polyhedral (will make several shapes which would differ in number of z-sections. It was used to generate data for scalability. It will also make plots, so the user can see what was generated.



Plots obtained by sbtgenpolycones and sbtgenpolyhedra

sbtscale

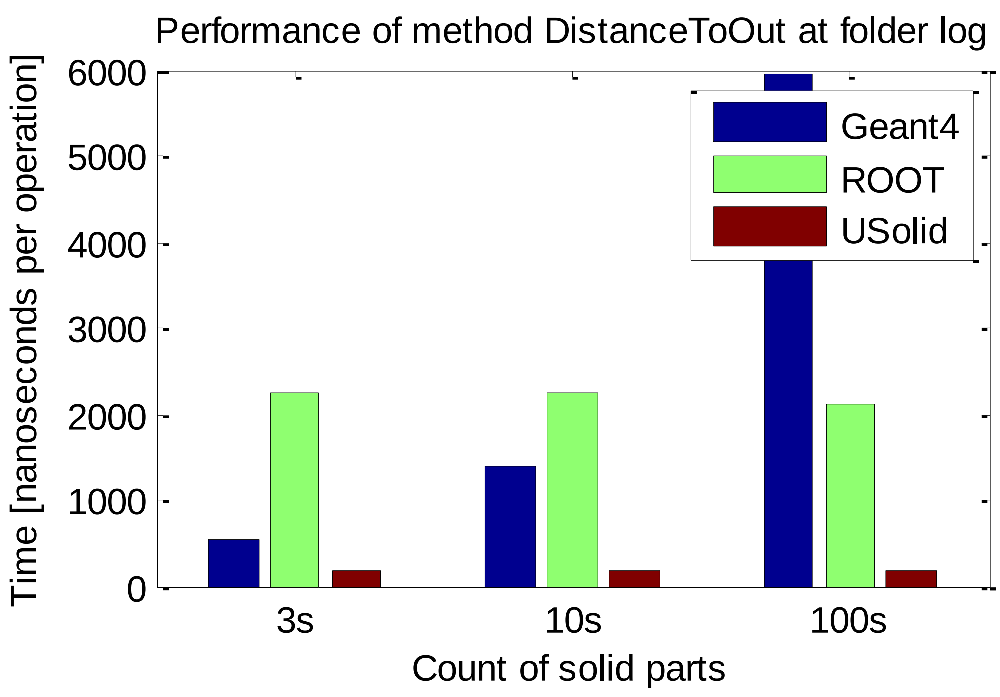
Used solely for comparison of scalability of method Inside for Multi-Union from data needs times.dat and nodes.dat generated by C++ code when concrete algorithm is chosen (no voxelization, with voxels). These need to be renamed properly to times*.dat and nodes*.dat (see sbtscale.m for more details).



A plot from sbtscale

sbtscalability(n) / sbtperfall(n)

Makes scalability graphs by using several folders with datasets generated using USBT. By editing this script allows user to specify folders which would be passed to core of this method, which is sbtperfall which would be called several times. This script makes scalability pictures. There is one parameter, n which specify which part of the folder name, separated by the dash (“-“) should be used for labelling the x-axis of the plot.



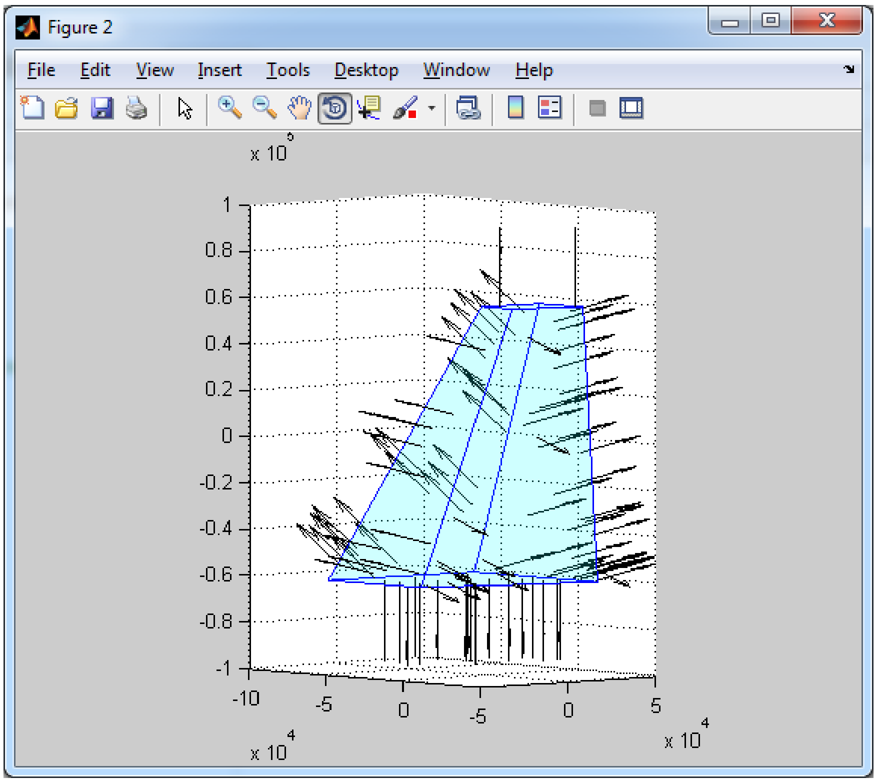
A plot obtained from sbtperfall. Sbtscalability would visualize such picture for several methods that the user will specify there

sbtvectors(method, name, nameValues1, nameValues2, first, count, color)

Perhaps the most tricky script. For given method, it will visualize **vector data-sets with file name starting with *method* and appended by *name***. *nameValues1* and *nameValues2* allow to specify concrete data sets which would contain values or differences which would be mapped on the vector (this is currently not typically used, and can be skipped by using None). First and count allows to focus on region or allow to focus e.g. on one vector. This command is typically used together with **sbtpolyhedra or sbtplot3d**, which will visualize the shape and make figure. This can be used e.g. 2 times with different colours, this way e.g. root and Geant4 normals can be visualized. Because sbtvectors does not open its own window, counting that this command is typically used with other command, it may be necessary to add *figure;* in the beginning!

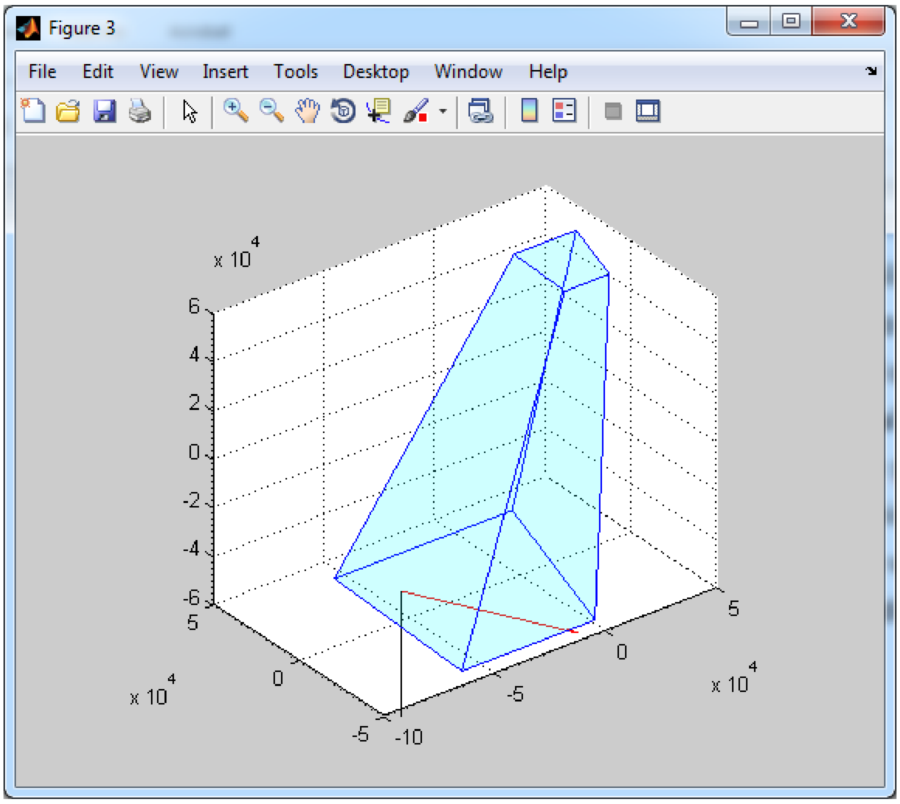
Example 1: visualization of Geant4 vectors (first 100) follows:

figure; sbtpolyhedra(Normal); sbtvectors(Normal, Geant4, None, None, 1, 100);



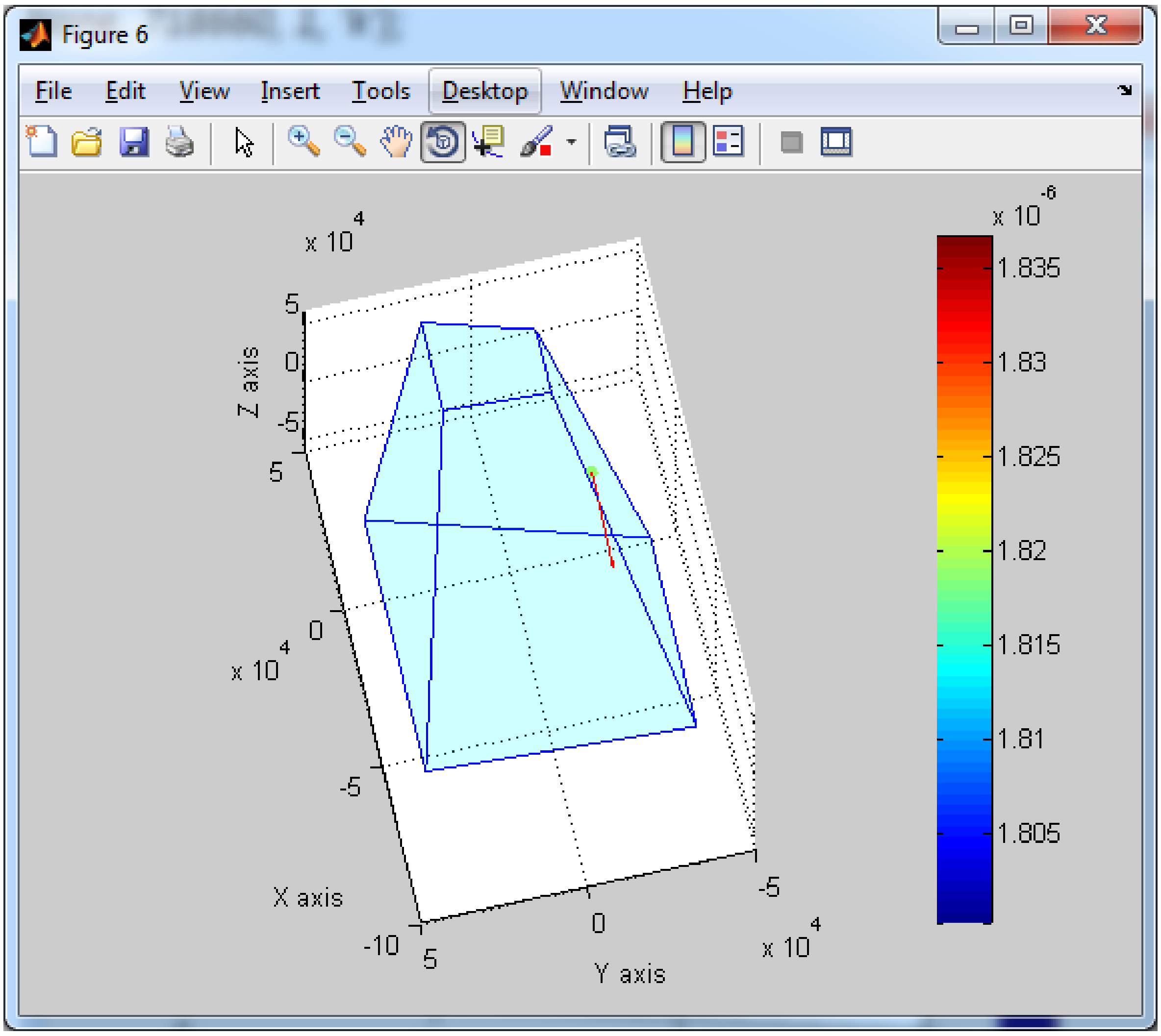
Example 2: by using the following command with trapezoid datasets and knowing that index 352 contain erroneous point (can be obtained using *sbtplot*), you could give following visual follows:

figure; sbtpolyhedra(Normal); sbtvectors(Normal, Geant4, None, None, 352, 1, 'k'); sbtvectors(Normal, Root, None, None, 352, 1, 'r');



Example 3: Visualization of *DistanceToOut* (case where difference around 10-6 was found between Geant4 and ROOT). Note that green point bellow represents the value on the scale-bar right follows:

sbtplot3d(DistanceToOut, Geant4, Root, 718660, 1); sbtvectors(DistanceToOut, 'Directions', None, None, 718660, 1, 'r');

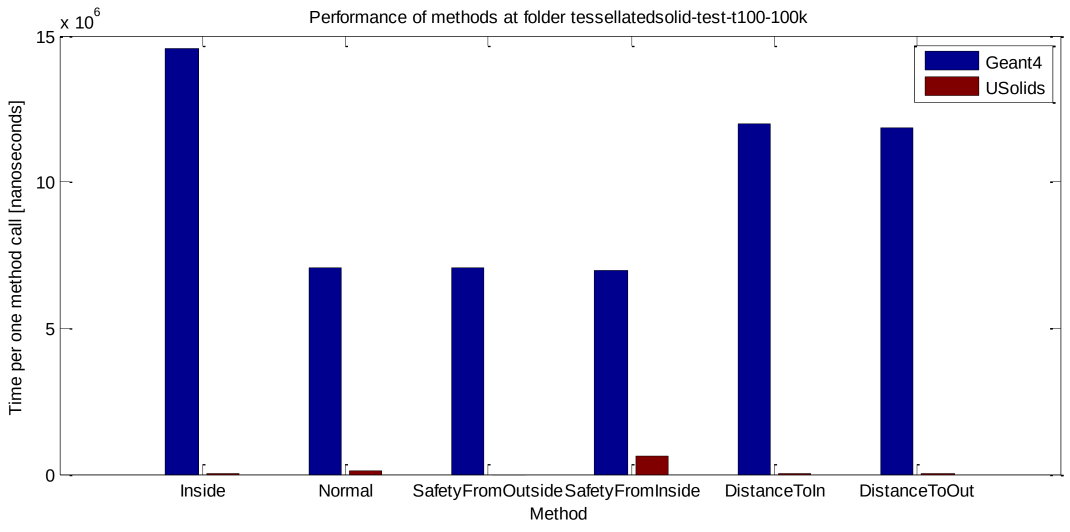


sbtperf(scale)

One of most easy script with only one parameter. Without any parameter, it will put the performance plot. With parameter, it will set the MATLAB YScale parameter (*set(gca,'YScale', scale)).* We have this parameter especially for easy logarithmic scale).

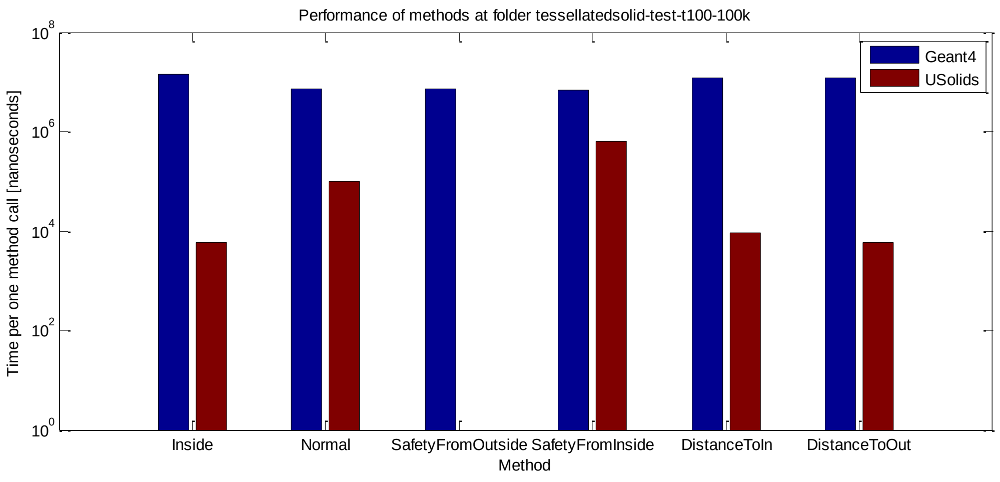
Example with LHCb 164k foil tessellated solid:

sbtperf;



Example with LHCb 164k foil tessellated solid follows:

*sbtperf(**'log'**);*



sbtdifferences.m, sbtdot.m, sbtplotpart.m, sbtpoints.m

These files are used internally by the scripts above. These are not meant to be called directly.