

Software and Robotic Integration

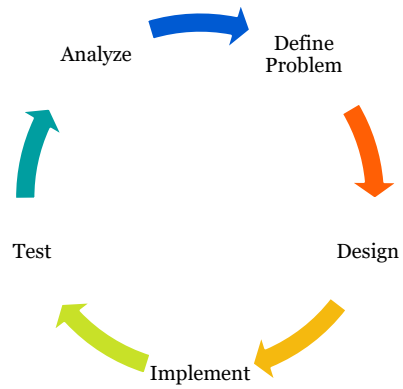
Good Practices for Code Design

Rachel Sparks, Ph.D.
Rachel.sparks@kcl.ac.uk
Lecturer in Surgical & Interventional Engineering
School of Biomedical Engineering & Imaging Sciences

Code Design

- Systematic approach to design & implementing code
 - There are many possible methods – Agile, Waterfall, Scrum, Lean, Extreme
 - Some differences, but we are going to focus on the similarities
- Common trends
 - Decouple design (*how* code achieve aim) from implementation (specifics of *what* code does)

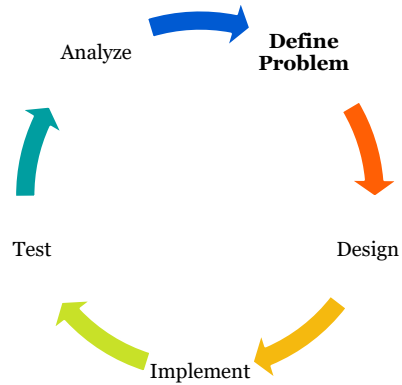
Code Design Cycle



Have them define the problem theoretically

Code Design Cycle

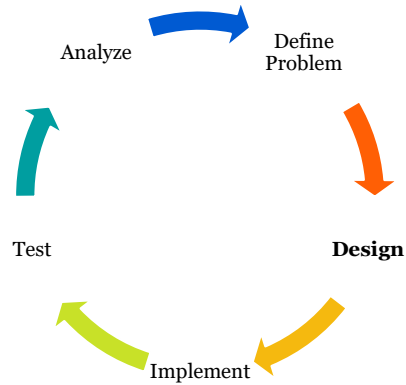
- Define Problem
 - Why is our code being created?
 - Corresponds to theoretical definition
- At this point don't think about code, inputs or outputs
 - Use mathematics or English to define the main idea
 - Should be able to do in one sentence
- In school this typically defines your assignment
- Industry this is done by your boss/client



The “why” we are coding -> “why” is this code going to solve a real world problem/automate an important task etc.

Code Design Cycle

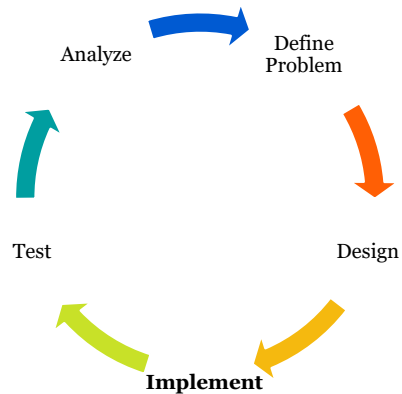
- Design
 - Structure the <problem> into smaller tasks?
 - Corresponds to Pseudo-Code
- At this point don't think how to code
 - Imagine you have a friend that only uses FORTRAN how would you describe this problem?
 - Define inputs and outputs
 - Define information flow in the program



The “how” -> how are we going to solve the problem, does not need to be line by line but with enough granular detail we know which individual components we need and how they should be interacting

Code Design Cycle

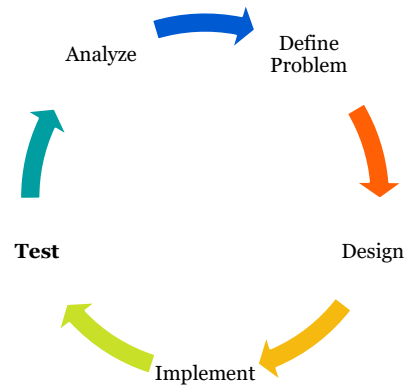
- Implement
 - Translate Pseudo-Code to actual code
- Key here is to start small and build up
 - Start with what you know how to do
 - What (if any) code have we already created that might help



“What” -> once we know how we want to solve our problem we need to define exactly what the code will do to achieve our design.

Code Design Cycle

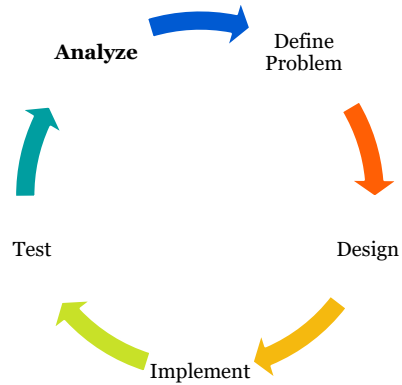
- Test
 - Check does our code work as expected?
 - Early warning your code is not correct
- Key is to keep it simple
 - Minimum working example
 - Easy to check – we already know the result(s)



Is it working -> always try to define a very simple test that you can easily check. For instance give the program one positive example (code should work as expected and give a known output), one negative example (code should fail or this is an edge case where the result should be a known, but invalid output), and a few nonsense examples (for instance an empty input, a datatype you are not expecting, etc.)

Code Design Cycle

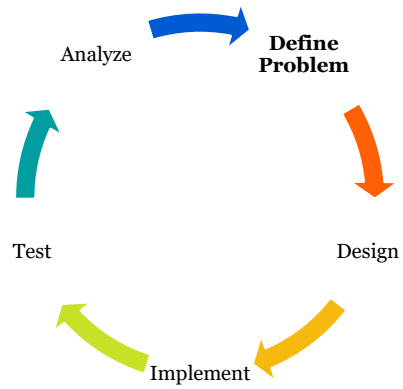
- Analyze
 - Have we have solved the problem?
 - Corresponds to Validation
- Assess how the code works on real data
 - Define criteria of success – we may not always know the right answer
 - Allows us to define expected performance



This can be where we bring in edge cases – where the output is not known ahead of time but we want to see if the algorithm would work under a special scenario or on real world data (that might have noise or other “problems” not introduced during testing)

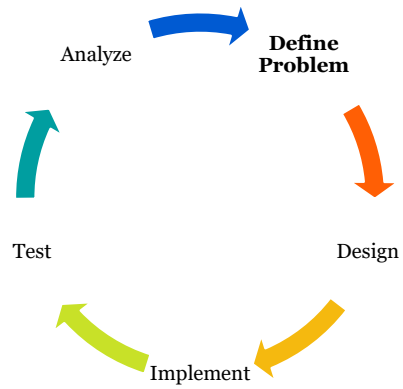
Code Design Cycle

- Define Problem
 - Rescope if your analysis shows your implementation is not good enough
 - Move onto another interesting problem if you have solved it



Code Design Cycle

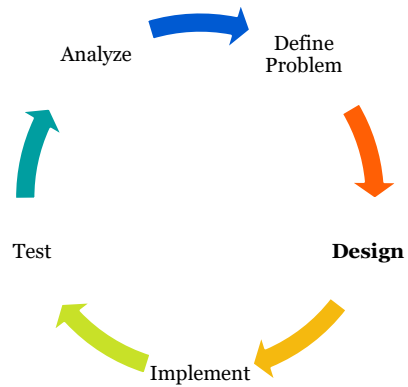
- Define Problem
 - I want an algorithm that will find all the points inside a structure
 - Find all point $p \in P$ such that $\text{Outside}(p, B) = \emptyset$



Have them define the problem theoretically

Code Design Cycle

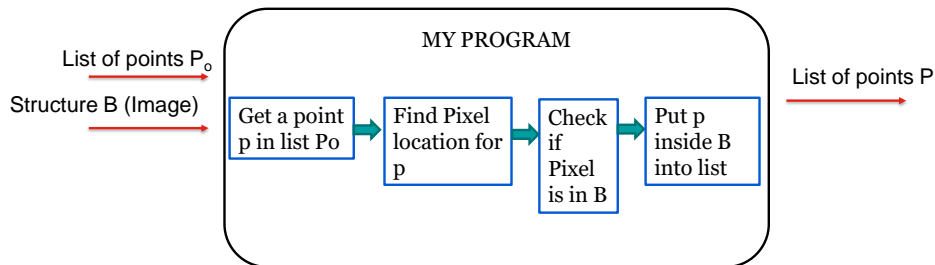
- Design
 - I want an algorithm that will find all the points inside a structure
- Usually inputs/outputs are defined first



Have them define the problem theoretically

Code Design Cycle

- Design
 - I want an algorithm that will find all the points inside a structure
- Usually inputs/outputs are defined first



What can we further define in the inputs

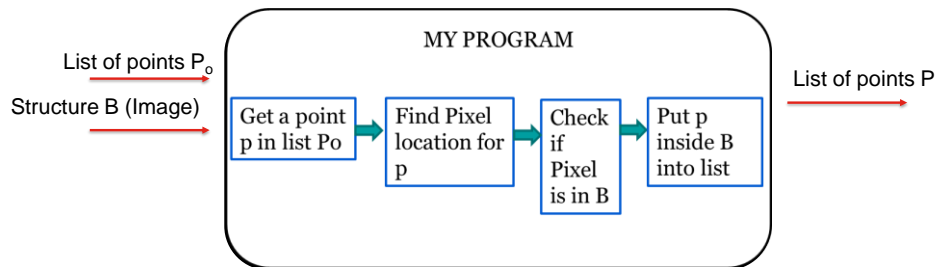
Think about the elements and how they need to be connected

Does not have to be linear just think about the minimum step you can take for a specific input or desired output

Then think about how to line up the boxes– don't worry about having to "add" boxes if you need to make a link!

Code Design Cycle

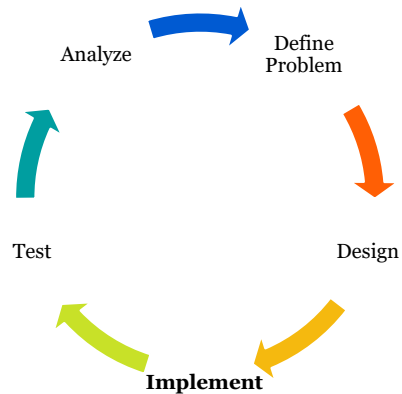
- Design
 - Take the elements you have designed and identified how to turn them into pseudo code



What can we further define in the inputs

Code Design Cycle

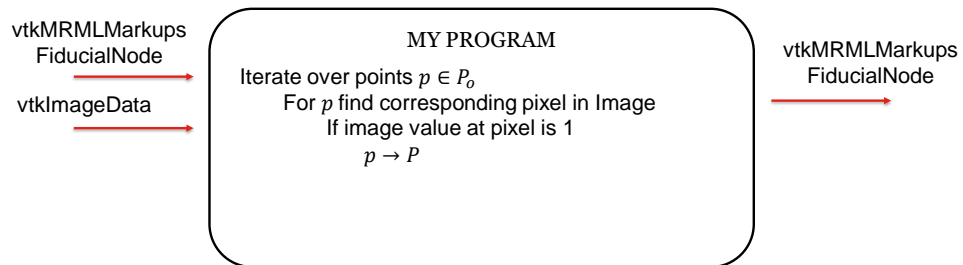
- Implement
 - Translate Pseudo-Code to actual code
- Key here is to start small and build up
 - Start with what you know how to do
 - What (if any) code have we already created that might help
 - For now skip what we don't know how to do



Have them define the problem theoretically

Code Design Cycle

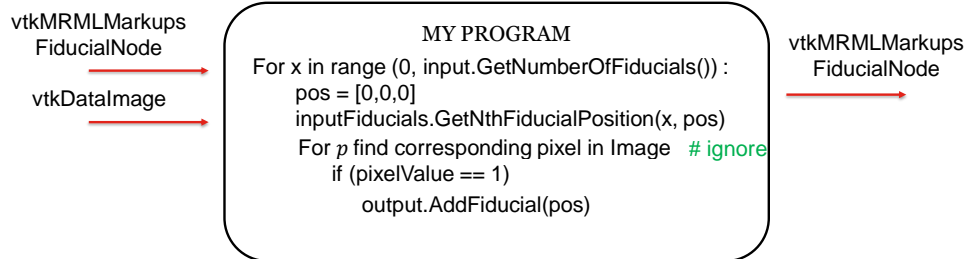
- Implement
 - Inputs and Outputs are probably defined by our code base



What can we further define in the inputs

Code Design Cycle

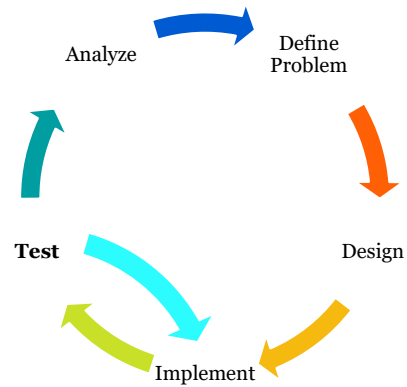
- Implement
 - Inputs and Outputs are probably defined by our code base
 - What do you know how to do
 - We have done a fair amount of code – but haven't tested if anything works yet



Start adding in simple things we know how to do ignoring those things we don't know how to do

Code Design Cycle

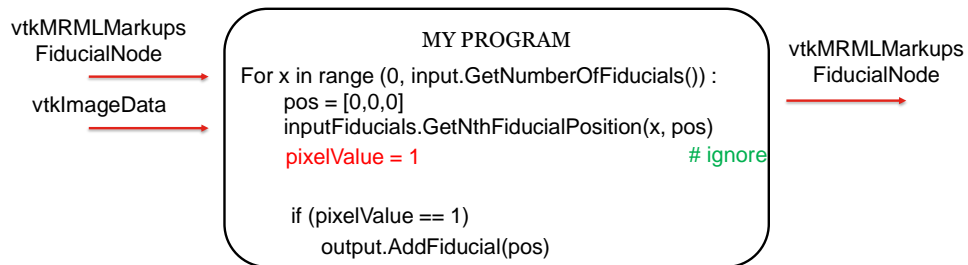
- Test
 - Check does our code work as expected?
 - Early warning your code is not correct
- Your test should inform how to fix your implementation



What can we test and how?

Code Design Cycle

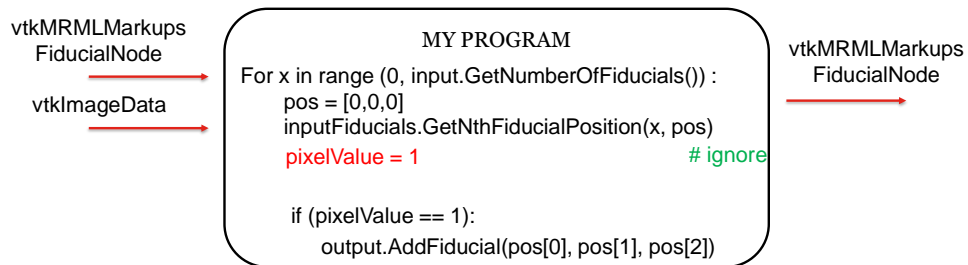
- Test
 - Don't need to finish to start testing
 - If we run incomplete it will complain pixelValue is not defined – so lets define it



Fill in misc values/variables that we don't know how to properly complete yet for the purpose of testing

Code Design Cycle

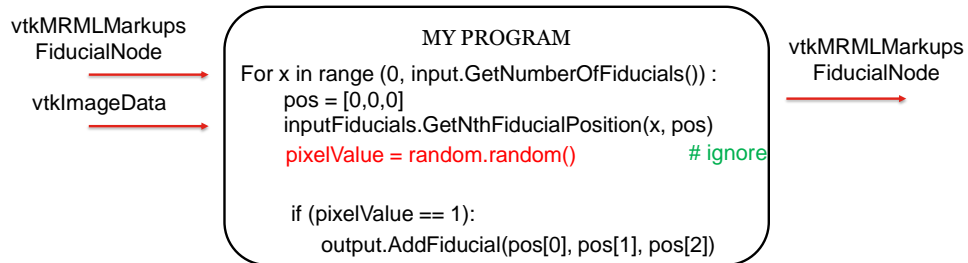
- Test
 - Don't need to finish to start testing
 - If we run incomplete it will complain pixelValue is not defined – so lets define it



Fix any mistakes so that we return what we would expect

Code Design Cycle

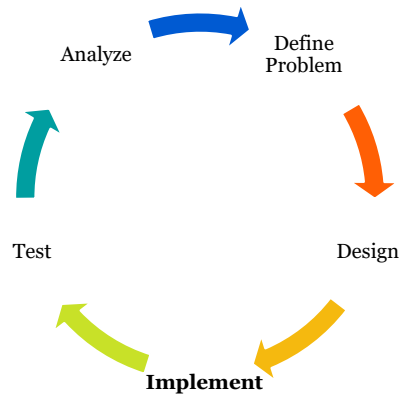
- Test
 - Don't need to finish to start testing
 - If we run incomplete it will complain pixelValue is not defined – or a slightly better test



Maybe try creating a slightly better check – just to see that our if statement is working correctly.

Code Design Cycle

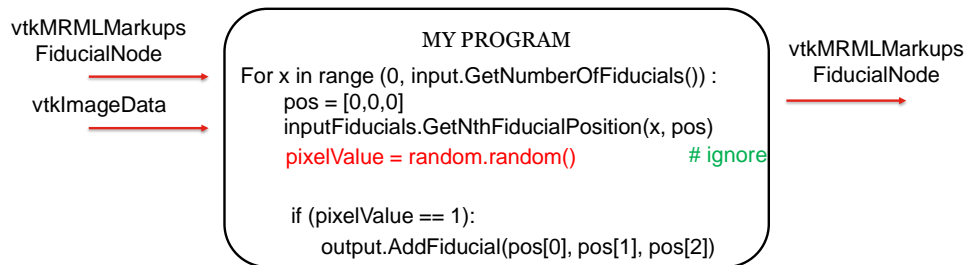
- Implement
 - Translate Pseudo-Code to actual code
- Key here is to start small and build up
 - Now we only have to finish up the things we don't know how to do



Have them define the problem theoretically

Code Design Cycle

- Implement
 - We need pixelValue from our vtkDataImage



Finally we can focus on filling in the line of code we have chosen to ignore.

vtkImageData

There are many ways to do this

- Documentation: <https://vtk.org/doc/nightly/html/classvtkImageData.html>
- Inline help

```
>>> help(vtk.vtkImageData)
Help on class vtkImageData:

class vtkImageData(vtkDataSet)
|   vtkImageData - topologically and geometrically regular array of data
|
|   Superclass: vtkDataSet
|
|   vtkImageData is a data object that is a concrete implementation of
|   vtkDataSet. vtkImageData represents a geometric structure that is a
|   topological and geometrical regular array of points. Examples include
|   volumes (voxel data) and pixmaps.
|
```

- Tab complete to list functions

```
>>> vtk.vtkImageData()
|   Decrease the reference count
|   AddObserver()
|   has the same effect as
|   reference count
|   AllocateCellGhostArray()
|   AllocatePointGhostArray()
|   AllocateScalars()
|   AttributeTypes()
|   BoundingBox()
|
```

First determine where you should look to find the functions/calls

vtkImageData

There are many ways to do this

- Documentation: <https://vtk.org/doc/nightly/html/classvtkImageData.html>
- Inline help
- Tab complete to list functions
- Other online program forums: <https://programtalk.com/python-examples/vtk.vtkImageData/>; <https://stackoverflow.com/questions/tagged/vtk>

We are still going to need a bit of reasoning/trial and error

- Find/Get usually retrieve things
- Set usually sets things

Use some knowledge/logic to help figure out where to start your investigation

vtkImageData

Read to identify likely candidates

```
>>> help(vtk.vtkImageData().FindPoint)    >>> help(vtk.vtkImageData().GetPoint)
Help on built-in function FindPoint:      Help on built-in function GetPoint:

FindPoint(...)                            GetPoint(...)
V.FindPoint(float, float, float) -> int   V.GetPoint(int) -> (float, float, float)
C++: virtual vtkIdType FindPoint(double x, C++: double *GetPoint(vtkIdType ptId)
double y, double z)                      override;
V.FindPoint([float, float, float]) -> int V.GetPoint(int, [float, float, float])
C++: vtkIdType FindPoint(double x[3])      C++: void GetPoint(vtkIdType id, double
override;                                x[3]) override;
Standard vtkDataSet API methods. See      Standard vtkDataSet API methods. See
vtkDataSet for more                      vtkDataSet for more
information.                             information.
```

Help paired with your knowledge of how the code is named can help identify possible functions to use

Code Design Cycle

- Implement
 - We need pixelValue from our vtkDataImage

```
ind = inputVolume.GetImageData().FindPoint(pos)
# a bit of math to transform the ind into i,j,k position
dim = inputVolume.GetImageData().GetDimensions()
xInd = ind % dim[0]
yInd = (ind % (dim[0] * dim[1]) ) / dim[0]
zInd = ind / (dim[0] * dim[1])
pixelValue = inputVolume.GetImageData().GetScalarComponentAsDouble (xInd,yInd,zInd,0)

if (pixelValue == 1)
    output.AddFiducial(pos[0], pos[1], pos[2])
```

After some trial and error we can figure out how to translate our point into a pixel

Class Design

In general define classes where possible to

- More closely follow the pseudo code flow charts
- Help isolate code logic into classes – when you reuse it you do not need to remember implementation details every time
- Allows easier testing – test function not every single call
- Remember to document the point of classes

Knowing how to break code into classes involves a bit of practice

- If you find you are constantly creating new classes with minor variations consider trying to make the function more general
- If you find multiple classes have the same documentation for many classes consider consolidating
- If your class is only called once consider removing

Code Refactoring

Don't be afraid to rewrite code.

- Once you have implemented your entire algorithm you may realise there are unnecessary calls or multiple similar lines of code
- Similar code may indicate you need to create a class
- If after implementation you think you have a simpler way to achieve the same results try it
 - Fewer lines of code can be easier to debug
 - More readable code can be easier to use and debug later

Software and Robotic Integration

Hard Constraints Part 1

Rachel Sparks, Ph.D.
Rachel.sparks@kcl.ac.uk
Lecturer in Surgical & Interventional Engineering
School of Biomedical Engineering & Imaging Sciences

Overview of Path Planning

Moving objects through space, including robots and other medical tools, should take into account:

- Physical constraints
- Patient safety
- Medical staff safety

Path planning is how to ensure the robot/tool moves without violating the laws of physics (or poking out an eye)

Overview of Path Planning

Informally, path planning is often described by hard constraints

- Cannot go through <bone, surgical tables, surgeon>
- Avoid <vessels, lung>
- Target <tumour>; Insert into <kidney>

These can also be inequalities

- Drill bits slip at angles above <degree> to <bone>
- Due to breathing motion we avoid <vessels> by up to <distance> to ensure safe targeting

Overview of Path Planning

Path planning also involves soft constraints

- Minimize <length of tool> in <tissue>
- Maximize <distance> to <vessel>
- Reduce <time> to <position robot>

These will be discussed in more detail next week

Path Planning Formalization

Path planning involves

- Potential paths (states) a robot can take $S(t)$
 - A state $s(t) \in S(t)$ is a function that defines the location in the world the object occupies
 - t is time, parameterizing changes in robot state over time

States can be either discrete variables, e.g. the robot can move 1mm in the x, y, or z direction, or a continuous function based on some input parameters, e.g. my robot can move to any point x within a cube defined by $[x_{\min}, x_{\max}]$. If it is continuous we can assume there are a set of parameters θ that define the position of the robot making $S(t) \rightarrow S(t, \theta)$

Path Planning Formalization

Path planning involves

- Potential paths (states) a robot can take $S(t)$
- An image scene B that contains a set of objects $l \in L$
 - Many image segmentation algorithms (see Week 1 Lecture)
 - The scene is represented as $f_B(c, l) = [0, 1]$
 - Each pixel c has a binary value for all $l \in L$

Strictly speaking the objects can be time varying too! This makes our life a lot more complicated, so we are going to ignore it at the moment

Path Planning Formalization

Path planning involves

- Potential paths (states) a robot can take $S(t)$
- An image scene B that contains a set of objects $l \in L$
- Hard constraints that must be met:
 - Cannot go through <bone>

$$S_{good}(t) \subseteq S(t) \text{ s.t. } Intersect(s(t), f_B(c, l_{bone}) = 1) = 0 : \forall s(t) \in S_{good}(t)$$
 - Target <tumour>

$$S_{good}(t) \subseteq S(t) \text{ s.t. } Intersect(s(t), f_B(c, l_{tumour}) = 1) \neq 0 : \forall s(t) \in S_{good}(t)$$
 - Path must have an angle with the skull below 35°

$$S_{good}(t) \subseteq S(t) \text{ s.t. } f_a(s(t), f(c, l_{skull}) = 1) < 35^\circ : \forall s(t) \in S_{good}(t)$$

• ~~Soft constraints that should be optimized for:~~

Strictly speaking the objects can be time varying too! This makes our life a lot more complicated, so we are going to ignoring it at the moment

Path Planning Formalization

Path planning involves

- Potential paths (states) a robot can take $S(t)$
- An image scene B that contains a set of objects $l \in L$
- Hard constraints that must be met:

We want the following algorithm

```
For  $s(t) \in S(t)$ 
  For  $c \in B$ 
    if (Criteria 1 & Criteria 2 & Criteria 3)
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

What is the O?

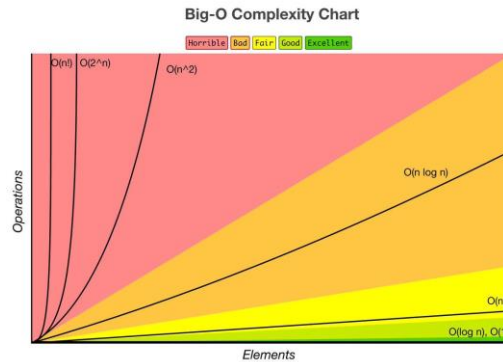
What is the Big O notation for this most simply of formulas? $O(N_s N_c N_{\text{criteria}})$
in general the number of elements in b is going to be the rate limiting step; although
in some complex robotic configurations it may be the number of states

Big “O” Notation

Big “O” gives a sense of the theoretical worst case scenario

- Can help identify software performance early independent of implementation details
- Useful to select best algorithm before implementation

Some things won't change your Big “O” but will help your algorithm go faster on average



Path Planning Formalization

We want to design the following algorithm

```
For  $s(t) \in S(t)$ 
  For  $c \in B$ 
    if (Criteria 1 & Criteria 2 & Criteria 3)
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

This algorithm is

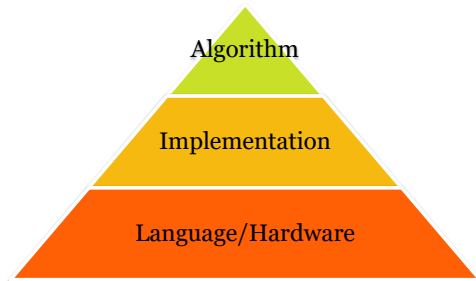
- Incredibly slow and computationally intensive but...
- (As long as Criteria are formulated correctly) gives the complete & correct result
- This makes it great for testing/verification

Everything else today is about what we can do to make this algorithm “better”

Better can be – faster, less computationally intense, reducing the search space (i.e. only exploring subsets of the data).

Algorithm Development – Avoid Over Design

1. Is a better architecture even necessary?
 - a. Only to prove a theoretical concept?
 - b. Only ever going to run on small N?
 - c. Another step in the workflow that is much slower? Re-design the “slowest” steps first
2. Re-design the right thing – try to design “top down”



Path Planning Algorithm Design

We want to design the following algorithm

```
For  $s(t) \in S(t)$ 
  For  $c \in B$ 
    if (Criteria 1 & Criteria 2 & Criteria 3)
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

What can we make better here?

Path Planning Algorithm Design

We want to design the following algorithm

```
For  $s(t) \in S(t)$ 
  For  $c \in B$ 
    if (!Criteria 1)
      return
    else if (!Criteria 2)
      return
    else if (Criteria 3)
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

$O(N_s N_b 3)$ but will on average exit earlier now – once a single hard constraint is violated.

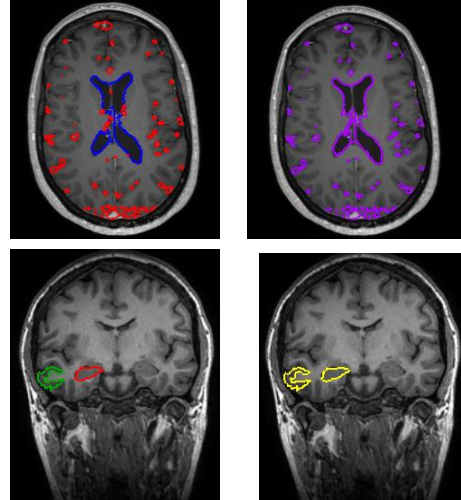
Path Planning Algorithm Design

Consider the details of Criteria and B

- If two criteria are avoid ventricles ($f_B(c, l_{vents}) = 0$) and avoid blood vessels ($f_B(c, l_{vessels}) = 0$)
- Equivalent to avoid all critical structures ($l_{crit} = l_{vents} \cup l_{vessels}$)

Be careful not all criteria can be easily combine

- Place tool through hippocampus ($f_B(c, l_{hippo}) = 1$) and middle temporal gyrus ($f_B(c, l_{mtg}) = 1$)
- NOT equivalent to through ($l_{combo} = l_{hippo} \cup l_{mtg}$)



Path Planning Algorithm Design

We want to design the following algorithm

```
For  $s(t) \in S(t)$ 
  For  $c \in B$ 
    if (!Criteria 13)
      return
    else if (Criteria 2)
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

$O(N_s N_b 2)$ now

- Reduced run time by 1/3 just by combining a few things!
- Best algorithm design

A Note on Image-based Constraints

Collisions/Intersection

- Detecting if you are inside or outside an object is easy
- Treat image as a lookup table and test if label is 0 or 1

Distance

- Detecting how far you are from an object is more complex than collisions
- Need to compute distance from binary mask
 - Simplest to count number of pixels ([Chamfer Distance](#)) will be inaccurate up to the width of a pixel
 - More accurate is to use other distance functions ([Danielsson](#) , [Maurer](#))

Angle

- Angle of collision with surface is very complex
- Need to estimate local curvature ([Anti-Aliasing Binary Image Filter](#), [Active Contour](#))

A Note on Image-based Constraints



Arrange criteria to reject easiest to compute

- Collisions/Intersection
- Distance
- Angle

Will help increase average speed time no change in O .

Path Planning Implementation

We want to design the following algorithm

For $s(t) \in S(t)$		How robot position is computed
For $c \in B$		How image element is defined

```
    if (!Criteria 13)
        return
    else if (Criteria 2)
         $S_{\text{good}}(t) \leftarrow s(t)$ 
```

When considering optimization typically cardinality of B is larger than $S(t)$

We have two elements to design – the robot and the images

Hence B is where we should focus our efforts as it is going to give us bigger gains.

Image Iteration

What is $c \in B$ doing

- Element wise iteration over the array of voxels
- Starts at the corner and visits sequentially

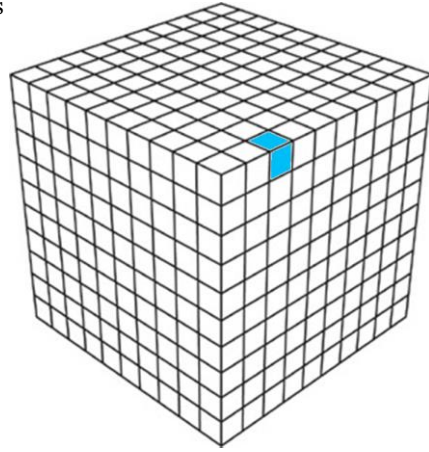


Image Partitioning – Octree Construction

Split B into halves of pixels

- Split is spatial (Left-Right, Anterior-Posterior)
- Now 8 small images
- Split those into halves
- Now 64 images
- Split those into halves
-

Image divisions grow by 8^n

- By 9th split there are over 100 million elements
- Or enough nodes to fill a 512 x 512 x 512 image

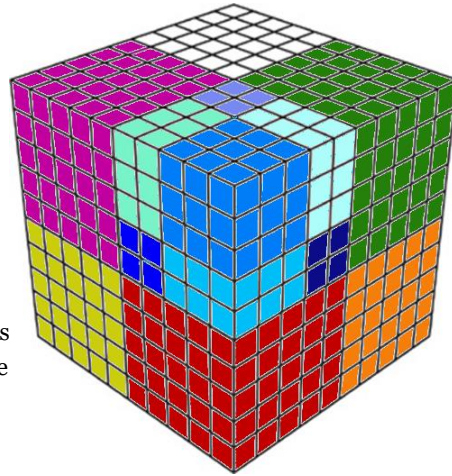


Image Partitioning – Octree Traversal

Start at top most layer

- Do any of the 8 divisions need to be visited?
- Visit only those divisions
 - Do any of their 8 divisions need to be visited?
 - Visit only those division?
 -

$O(\log(N_B))$ traversal time

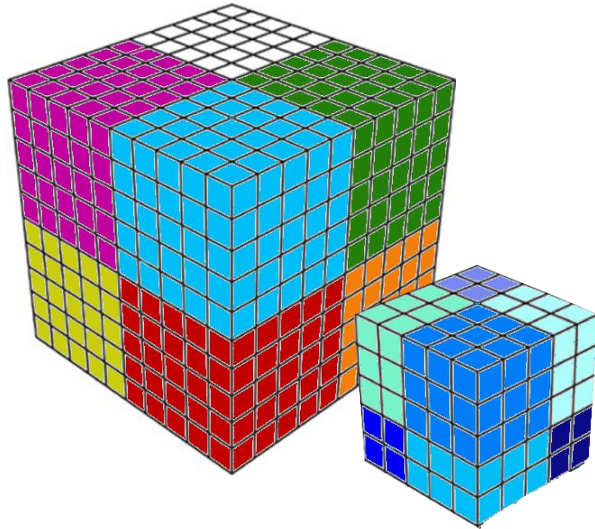


Image Partitioning – Octree Traversal

Enables quick iteration and ignoring of uninteresting portions of our image space

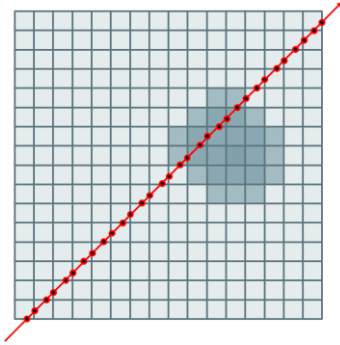
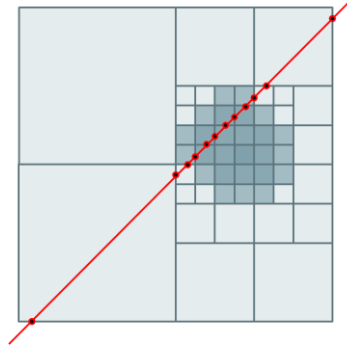


Image Iteration



Octree Iteration

Image Partitioning – Octree Traversal

Easiest to think of an octree as a tree

- “Root” is the top of the tree (i.e. the image)
- Each node may have
 - Children (smaller partitions of image)
 - A parent (a larger partition of image)

Can use recursion for the algorithm

- At each node decide which node to visit next
- When you are at a node with no children (“leaf node”) perform some check & return

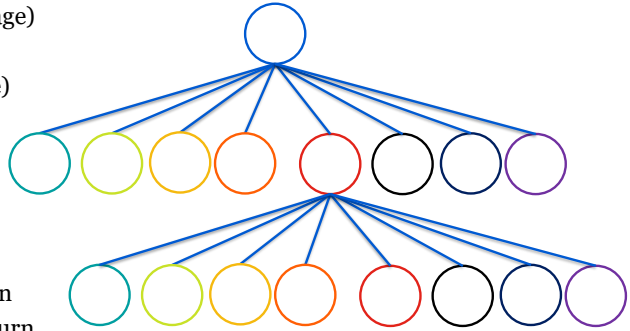


Image Partitioning – Octree Traversal

Easiest to think of an octree as a tree

- “Root” is the top of the tree (i.e. the image)
- Each node may have
 - Children (smaller partitions of image)
 - A parent (a larger partition of image)

```
ClimbThroughTree(Node n, p)
    if !n.HasChildren()
        return collide(n,p)
    else
        Node child = n.GetChild(p)
        ClimbThrougTree(child, p)
```

Can use recursion for the algorithm

- At each node decide which node to visit next
- When you are at a node with no children (“leaf node”) perform some check & return

Image Partitioning – Octree Traversal

Octrees make use of spatial relationships in the image to partition data

- Top down – uses information of the entire image to construct the tree
- Good for detecting collisions when an object is relatively small
- Poor for objects that are dispersed throughout the image

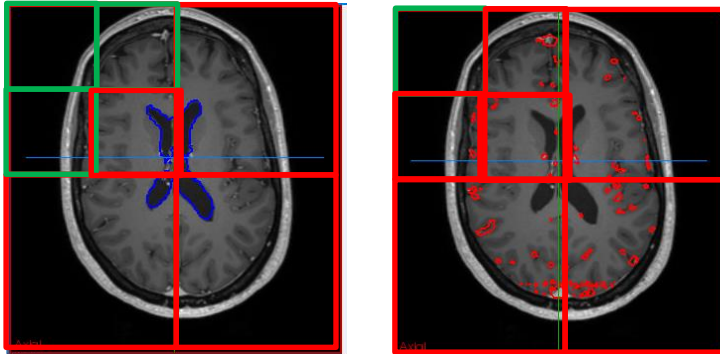


Image Partitioning – Bounding Volume Hierarchy

Another approach is “bottom up”

- Merge neighboring pixels together if they share a label
- Non-cubes are annoying to deal with so only do for “boxes”
- These are our child nodes

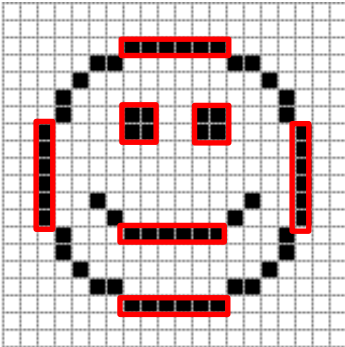


Image Partitioning – Bounding Volume Hierarchy

Another approach is “bottom up”

- Merge closest two children nodes....
- Until no more nodes remain

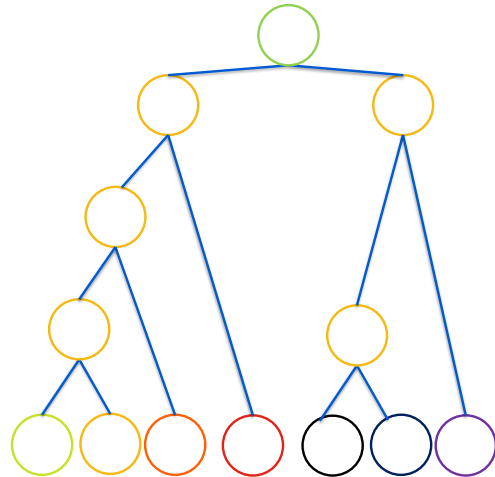
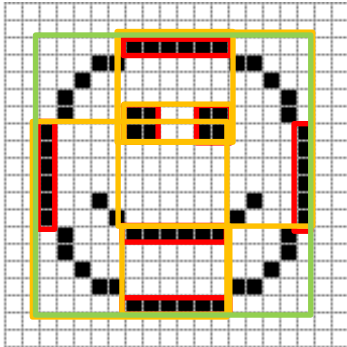


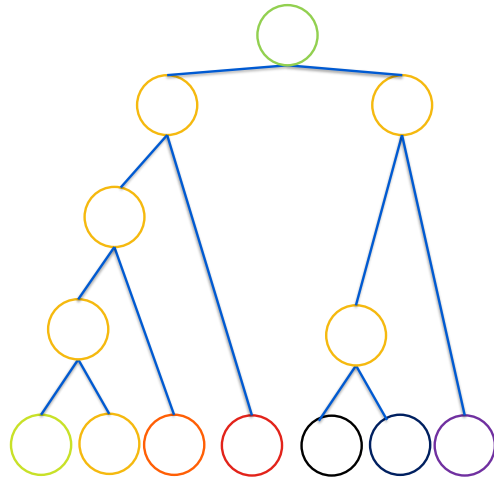
Image Partitioning – Bounding Volume Hierarchy

Another approach is “bottom up”

- Merge closest two children nodes....
- Until no more nodes remain

Just like the octree we can use recursion

- At each node decide which node to visit next
- When you are at a node with no children (“leaf node”) perform some check & return



Tree comparison

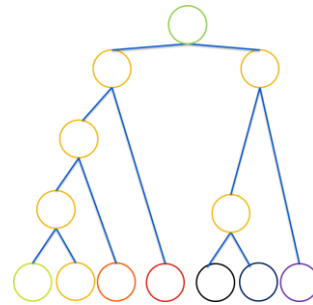
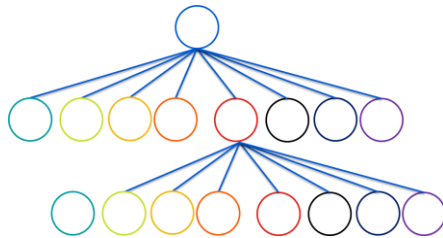
What is the correct tree for your data?

Tree depth

- Octrees are guaranteed balanced – small $\log_8(N_B)$
- BVH can be unbalanced – up to $|l|(|l| - 1)$

Number of nodes

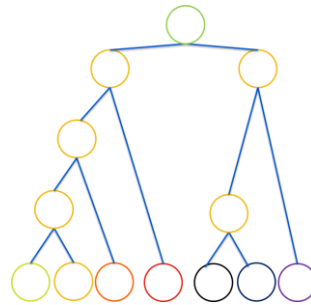
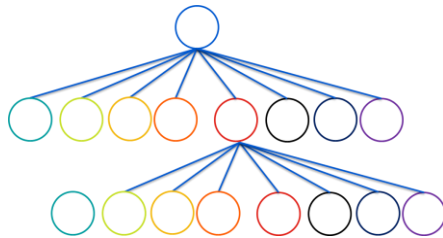
- Octree number of nodes based on total pixel value
- BVH based on total number of labels present



Tree comparison

What is the correct tree for your data?

- Octrees are good for
 - Spatially balanced, sparse data (one solid object)
 - Guaranteed worst case run time $O(\log(N_B))$
- BVHs are good for
 - Spatially unbalanced, sparse data (tendrils)
 - Guaranteed worst case run line $O(l^2)$



Implementation Notes & Inspiration

ITK Classes


- [Image Iterators](#) – Get all pixels
- [Conditional Image Iterators](#) – Get all pixels that meet a criteria (say having 1 as the label?)
- [Line Iterator](#) – Get all pixels along a line
- [Octree](#) – Image octree
- No Bounding Volume ☹

VTK Classes

- [Image Iterators](#) – Get all pixels
- [Octree](#) – Image octree (plus some extensions)
- [Oriented Bounding Box](#) – Only works on mesh data....

Path Planning Implementation

We want to design the following algorithm

```
For  $s(t) \in S(t)$   
  For  $c \in B$   Iterate over node  $\in OT(B)$   
    if (!Criteria 13)  
      return  
    else if (Criteria 2)  
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

Now $O(N_s \log(N_b)2)$

We have two elements to design – the robot and the images

Hence B is where we should focus our efforts as it is going to give us bigger gains.


Software and Robotic Integration

Hard Constraints Part 2

Rachel Sparks, Ph.D.
Rachel.sparks@kcl.ac.uk
Lecturer in Surgical & Interventional Engineering
School of Biomedical Engineering & Imaging Sciences

Path Planning Implementation

We want to design the following algorithm

```
For  $s(t) \in S(t)$   
  For  $c \in B$   Iterate over node  $\in OT(B)$   
    if (!Criteria 13)  
      return  
    else if (Criteria 2)  
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

Was $O(N_s N_b^3)$ now $O(N_s \log(N_b)^2)$

We have two elements to design – the robot and the images

Hence B is where we should focus our efforts as it is going to give us bigger gains.

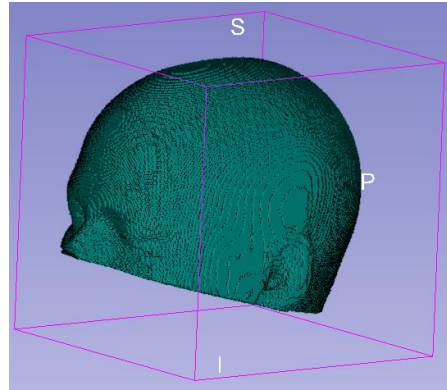
Defining Objects

Up to now we have been using images as look up tables $f_B(c, l) = [0, 1]$

- Size dependent on number of pixels
- Build up a representation based on little cubes

Is an image the best representation? We have alternatives

- Surfaces (meshes)
- Points



Using images as a LUT directly is the simplest approach. However, be aware of the stair casing effect – where your scene representation is essentially built up of “cubes” (voxels). This leads to inaccuracies in collision detection that can be on the order of +/- voxel size.

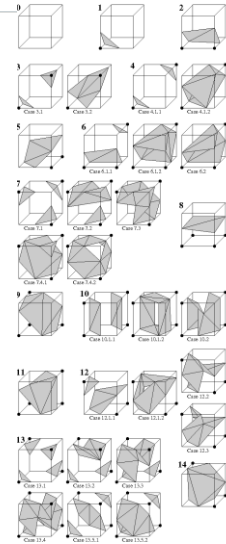
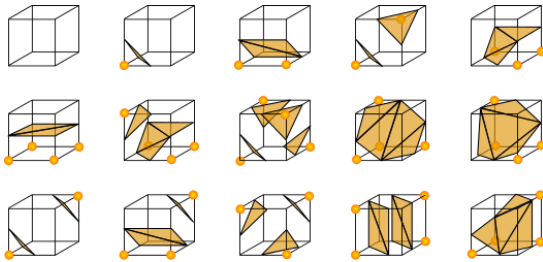
Image To Surface

Marching Cubes: convert image to surface

- Treat each voxel independently to identify rule

Marching Cubes 33

- Identified some rules that are dependent on local information
- Requires some neighbourhood iteration to ensure continuity



At the cost of increased memory we can compute surfaces -> sets of triangles that represent the surface of objects. These can be easier to work with, and may be easier to “smooth” to get a realistic object boundary definition

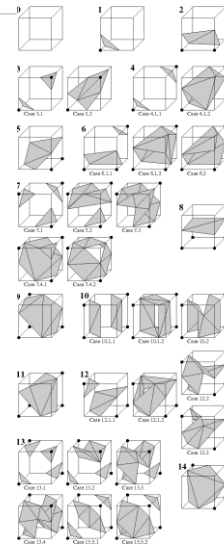
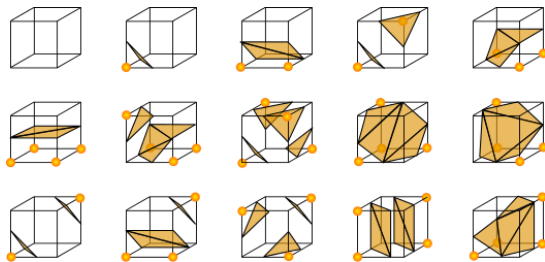
Image To Surface

Marching Cubes: convert image to surface

Marching Cubes 33

Implementation Details:

- [vtkMarchingCubes](#)
- [skimage.measure.marching_cubes_lewiner](#)



At the cost of increased memory we can compute surfaces -> sets of triangles that represent the surface of objects. These can be easier to work with, and may be easier to “smooth” to get a realistic object boundary definition

Image To Surface

Meshes may be able to represent objects more sparsely and accurately

- Take only boundaries into account – may be sparser data representation
- No more cubes – now we can represent using continuous triangles
 - Can enforce smoothness and continuity constraints
 - Distance and angles are easier to compute

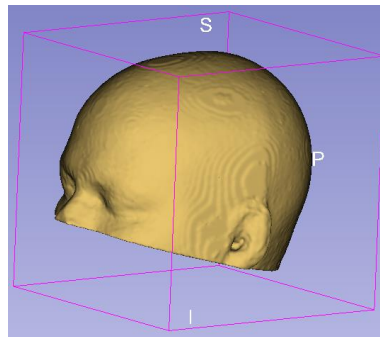
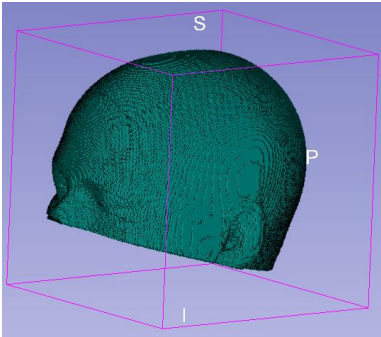
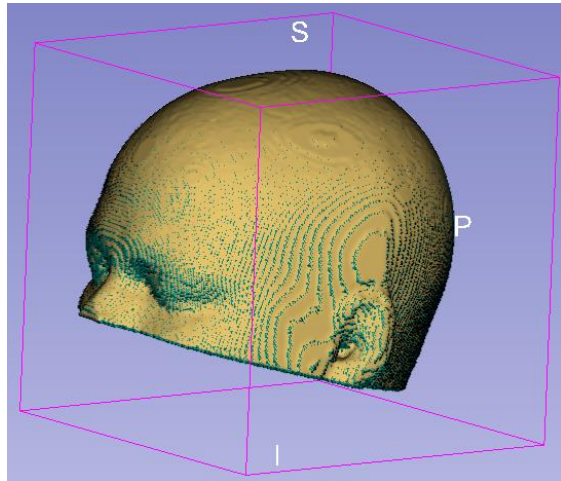


Image To Surface

Image rendering (green) and surface (yellow)
Direct comparison

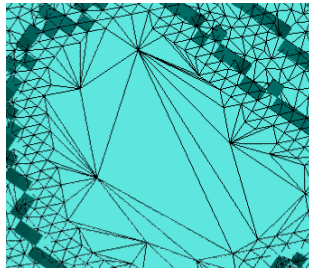
Mask has : 14417920 pixels
Surface has: 99543 polygons



Surface Traversal

Similar to image traversal

- Octree traversal to split triangles based on spatial location
- Same properties as image octree except
 - Be aware size of triangle matter – triangles can end up in more than one Octree leaf
 - Badly formed triangles can cause problems
 - Leaf node need to check triangle collision not box (slightly more complex)



Surface Traversal

Similar to image traversal

- Bounding Volume traversal modified to work better with triangles
 - Triangles are relatively planar
 - Bounding boxes can be oriented to align along major triangle axis
 - Leaf node need to check triangle collision – oriented bounding boxes help minimize need for this check (vtk [Oriented Bounding Box](#))

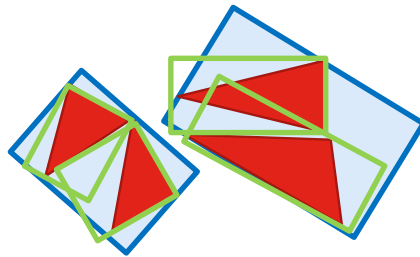


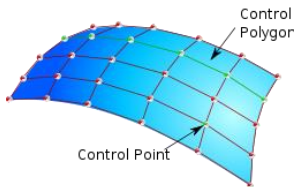
Image to Points

Implicit surface

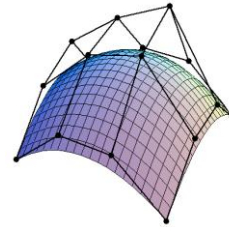
$$f_{surface} = \sum \phi(p_{query}) = 0$$

ϕ is some function (typically a polynomial) that takes uses a set of points $p \in P$ to define a surface

For a new point $f_{surface}$ is <0 for points inside the object, >0 for points outside the object.



Example NURBS (Non-uniform rational B-spline)



Bezier Surface – polynomial representation

Lightweight both in memory (only need to store points) and high in accuracy we can define “implicit” surfaces using sets of points to define an iso surface. The biggest downside is algorithmically this is challenging and can be computationally heavy to define an isosurface if you are using a large set of spline functions

NURBS collision detection: F. Page and F. Guibault, "Collision detection algorithm for NURBS surfaces in interactive applications," *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*, Montreal, Quebec, Canada, 2003, pp. 1417-1420 vol.2.

T. Sederberg and R. Farouki, "Approximation by interval Bezier curves" in *IEEE Computer Graphics and Applications*, vol. 12, no. 05, pp. 87,88,89,90,91,92,93,94,95, 1992.

Image to Points

Implicit surface

$$f_{surface} = \sum \phi(p_{query}) = 0$$

ϕ is some function (typically a polynomial) that takes uses a set of points $p \in P$ to define a surface

For a new point $f_{surface}$ is <0 for points inside the object, >0 for points outside the object.

Implementation Details

- [Bezier Curve](#)
- [Spline](#)

Lightweight both in memory (only need to store points) and high in accuracy we can define “implicit” surfaces using sets of points to define an iso surface. The biggest downside is algorithmically this is challenging and can be computationally heavy to define an isosurface if you are using a large set of spline functions

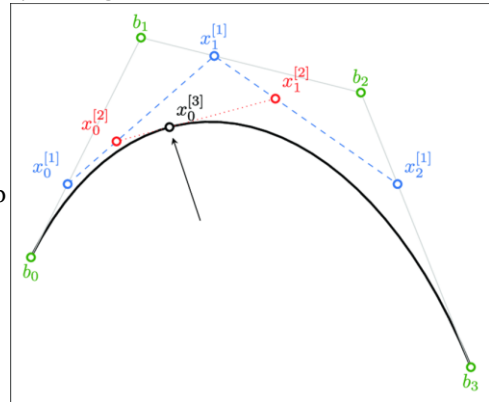
NURBS collision detection: F. Page and F. Guibault, "Collision detection algorithm for NURBS surfaces in interactive applications," *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*, Montreal, Quebec, Canada, 2003, pp. 1417-1420 vol.2.

T. Sederberg and R. Farouki, "Approximation by interval Bezier curves" in *IEEE Computer Graphics and Applications*, vol. 12, no. 05, pp. 87,88,89,90,91,92,93,94,95, 1992.

Image to Points

Implicit surfaces can be defined using piecewise approximations
(De Casteljau's algorithm)

- Recursively split the patch into two segments by finding
 - Appropriate Split point
 - New control points
- The set of control points can be used to define a bounding box for each curve
- As these surfaces are continuous functions algorithmically need to define discrete step (i.e. precision of system)



Wolfgang Boehm, Andreas Müller. On de Casteljau's algorithm. Computer Aided Geometric Design 16(7): 587-605, 1999.

Implicit Surface Traversal

As in image and surface

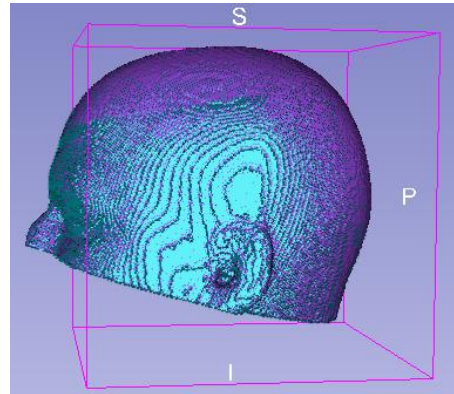
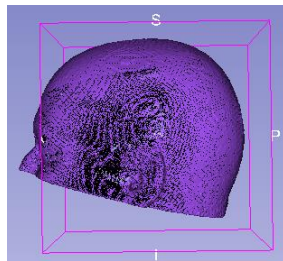
- Tree traversal can be used to identify region of curve to consider
- De Casteljau's algorithm can be used to iteratively define a bounding volume split
- For both octree and bounding volume each leaf node must contain enough points to correctly define surface patch in the region covered by the surface node
- Accurate but computational complex

Klein, J. and Zachmann, G. (2004), Point Cloud Collision Detection. Computer Graphics Forum, 23: 567-576. doi:[10.1111/j.1467-8659.2004.00788.x](https://doi.org/10.1111/j.1467-8659.2004.00788.x)

Point Cloud Collision Detection

Define object by a densely populated point cloud (Klein and Zachmann 2004)

- Define a collision as being with a set radius (defined by sampling distance between points)
- Quicker than implicit surfaces
- Much simpler and computationally quick

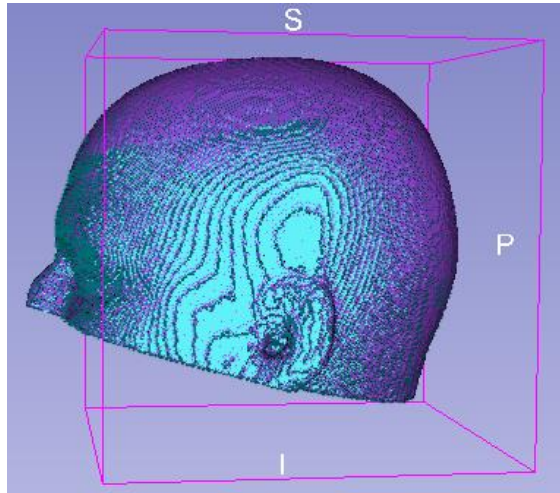


Klein, J. and Zachmann, G. (2004), Point Cloud Collision Detection. Computer Graphics Forum, 23: 567-576. doi:[10.1111/j.1467-8659.2004.00788.x](https://doi.org/10.1111/j.1467-8659.2004.00788.x)

Image To Surface

Image rendering (green) and surface (yellow)
Direct comparison

Mask has : 14417920 pixels
Surface has: 99543 polygons
Point count: 255787



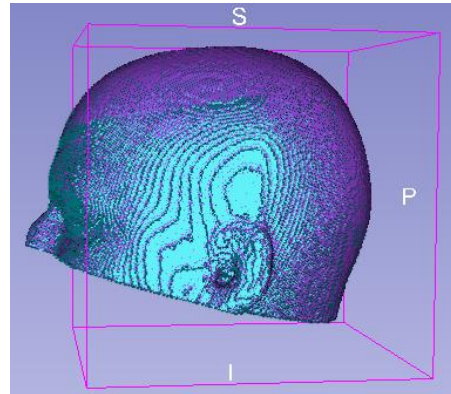
Point Cloud Collision Detection

Define object by a densely populated point cloud (Klein and Zachmann 2004)

- Define a collision as being with a set radius (defined by sampling distance between points)
- Quicker than implicit surfaces
- Much simpler and computationally quick

As with other object types

- Octrees can be used
- As points have no orientation/volume bounding volumes typically not used
- **kdTree** are tree with a binary partitioning of the points sets (similar to bounding volume)



Klein, J. and Zachmann, G. (2004), Point Cloud Collision Detection. Computer Graphics Forum, 23: 567-576. doi:[10.1111/j.1467-8659.2004.00788.x](https://doi.org/10.1111/j.1467-8659.2004.00788.x)

Path Planning Implementation

We want to design the following algorithm

```
For  $s(t) \in S(t)$ 
  For  $c \in B$ 
    if (!Criteria 13)
      return
    else if (Criteria 2)
       $S_{\text{good}}(t) \leftarrow s(t)$ 
```

We have two elements to design – the robot and the images

Hence B is where we should focus our efforts as it is going to give us bigger gains.

Defining States

$s(t)$ the state of a robot can be represented by a position $q(t)$

- 3D surface
- Parametric Line (+thickness)

Alternatively we can define $q(t) = f_q(\theta_i(t)) : i = [1, \dots, I]$

- $\theta_i(t)$ set of controllable tool parameters
- $f_q(\cdot)$ maps tool parameters $\theta_i(t)$ to tool position $q(t)$

Linking state to movement parameters allows for

- Robust representation of how the robot moves
- Easier to determine how to control robot
- Can help reduce optimisation space: we need only solve for $\theta_i(t)$

Simple lines and surfaces can help make a problem easy for a path planning perspective but may obscure the relationship between how to control the robot and where the position of the robot should be

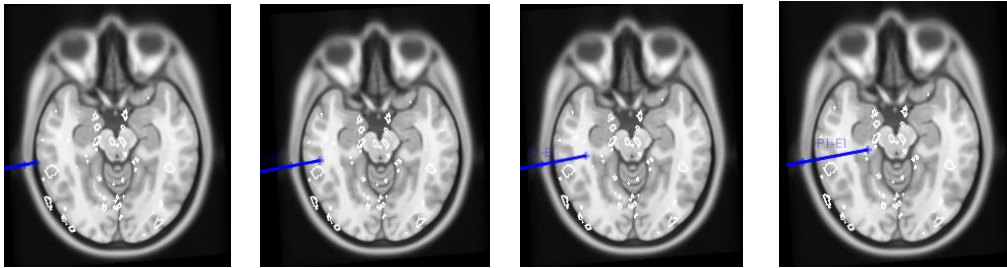
Defining States – Straight Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

For a straight tool $q(t) = [p(0), (p(t_{max}) - p(0))t]$

- $p(0)$ – Starting point
- $p(t_{max})$ – Ending point
- Don't forget some thickness term r

Note: can drop t ; $q(t_{max})$ contains information about all time points



Defining States – Straight Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

For a straight tool $q(t) = [p(0), (p(t_{max}) - p(0))t]$

- $p(0)$ – Starting point
- $p(t_{max})$ – Ending point
- Don't forget some thickness term r

Note: can drop t ; $q(t_{max})$ contains information about all time points

- Enumerating all possible states is quick and easy
- Once states enumerated reject based on criteria

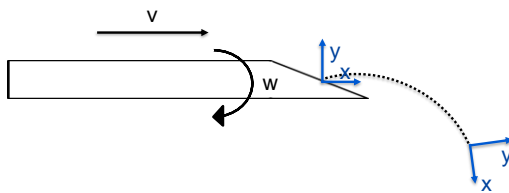
Defining States – Steerable Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

- At a steady insertion velocity v the needle with bend with a radius r
- Steering by turning the needle controls the direction of the bend (parameterised by θ, ϕ)

Simplest solution is assume position is a discrete update:

$$q(t) = q(t-1) + \begin{bmatrix} r \cos(\theta) (1 - \cos(\phi)) \\ r \sin(\theta) (1 - \cos(\phi)) \\ r \sin(\phi) \end{bmatrix}$$



The state depends on the previous state so discretizing along time can be a useful way to reduce the search space -> basically find viable solutions at increments of t starting early to exclude infeasible trajectories early

Webster, R. J., Kim, J. S., Cowan, N. J., Chirikjian, G. S., & Okamura, A. M. (2006). Nonholonomic Modeling of Needle Steering. *The International Journal of Robotics Research*, 25(5–6), 509–525. <https://doi.org/10.1177/0278364906065388>

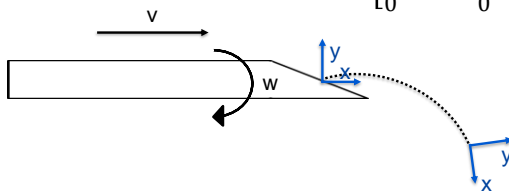
Patil, S., & Alterovitz, R. (2010). Interactive Motion Planning for Steerable Needles in 3D Environments with Obstacles. *Proceedings of the ... IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics. IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics*, 893–899. doi:10.1109/BIOROB.2010.5625965

Defining States – Steerable Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

- At a steady insertion velocity v the needle with bend with a radius r
- Steering by turning the needle controls the direction of the bend (parameterised by θ, ϕ)
- More complete solution is given by the instantaneous velocity:

$$\dot{g}(t) = g(t) \begin{bmatrix} 0 & -w(t) & 0 & 0 \\ w(t) & 0 & -v(t)/r & 0 \\ 0 & v(t)/r & 0 & v(t) \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



First three columns take into account material frame rotation and position

Defining States – Steerable Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

- At a steady insertion velocity v the needle with bend with a radius r
- Steering by turning the needle controls the direction of the bend (parameterised by θ, ϕ)

Simplest solution is assume position is a discrete update:

$$q(t) = q(t-1) + \begin{bmatrix} r \cos(\theta) (1 - \cos(\phi)) \\ r \sin(\theta) (1 - \cos(\phi)) \\ r \sin(\phi) \end{bmatrix}$$

Due to the iterative nature of the problem

- Best to reject positions at each step of t
- Avoid enumerate $q(t)$ for solutions that are infeasible early

The state depends on the previous state so discretizing along time can be a useful way to reduce the search space -> basically find viable solutions at increments of t starting early to exclude infeasible trajectories early

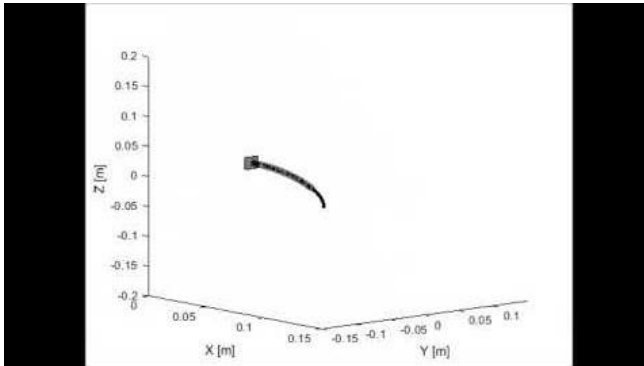
Webster, R. J., Kim, J. S., Cowan, N. J., Chirikjian, G. S., & Okamura, A. M. (2006). Nonholonomic Modeling of Needle Steering. *The International Journal of Robotics Research*, 25(5–6), 509–525. <https://doi.org/10.1177/0278364906065388>

Patil, S., & Alterovitz, R. (2010). Interactive Motion Planning for Steerable Needles in 3D Environments with Obstacles. *Proceedings of the ... IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics. IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics*, 893–899. doi:10.1109/BIOROB.2010.5625965

Defining States – Flexible Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

- Highly dependent on the robot
- Example concentric tube:



States can vary along the trajectory based on theta and phi!

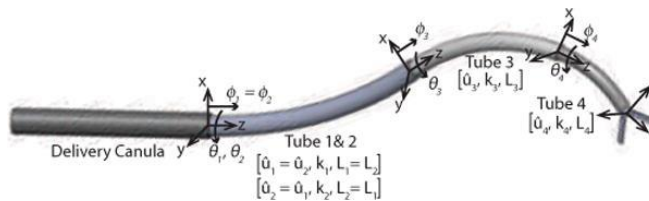
At least it is a piecewise Bezier function (each segment can be solved independently)

Defining States – Flexible Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

Position dependent on

- Relative length, size, stiffness, and bend of each tube
- Best modelled as a piecewise curvature (actually very similar to Bezier curves)



States can vary along the trajectory based on theta and phi!

At least it is a piecewise Bezier function (each segment can be solved independently)

C. Bergeles and P. E. Dupont, "Planning stable paths for concentric tube robots," *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, 2013, pp. 3077-3082.

Defining States – Flexible Tools

$s(t)$ the state of a robot can be represented by a position $q(t)$

Position dependent on

- Relative length, size, stiffness, and bend of each tube
- Best modelled as a piecewise curvature (actually very similar to Bezier curves)

In terms of optimisation this is the worst of all worlds

- Highly complex mathematics to describe relative locations
- Position is not easy to predict from previous state (non-linear relationships between tubes)
- Position is time varying
- Best to use optimisation algorithm to reduce the number of states to consider

More on how to perform optimisation next week.

States can vary along the trajectory based on theta and phi!

At least it is a piecewise Bezier function (each segment can be solved independently)

C. Bergeles and P. E. Dupont, "Planning stable paths for concentric tube robots," *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, 2013, pp. 3077-3082.

Acceleration Tips and Tricks

Ensure algorithm and implementation are as optimized as possible before considering these other options.

They are time consuming to implement. If algorithm and implementation are poor performance gains will be small

- Try to create parallel running threads
- Consider a better programming language
- Take advantage of hardware acceleration

Parallelisation

For path planning

- Possible robotic states $s(t) \in S(t)$ can be considered as independent
- The problem can be considered embarrassingly parallel

Subsets of $S(t)$ can be identified and run on parallel threads of the same function

- Can maximize computational resources and minimize runtime
- Needs a bit of programming to ensure each call is run on a separate processor core

Be aware other portions of your algorithm (element iteration, image manipulation) may also be multi-threaded

- In general parallelisation of the lower functions will give you better performance
- Can be useful especially for large computing resources (i.e. many cores)

Programming Language

The choice of programming language can have large impacts on runtime

- Lower level languages are usually faster for simple computations
- Learning curve for lower level languages is higher
- Longer development times for lower level languages

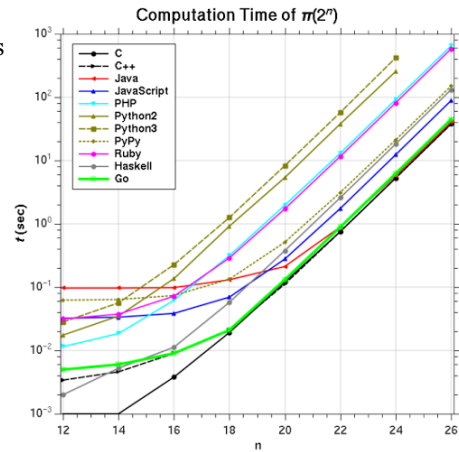


Figure is from () showing run time to compute digits of pi for a large set of programming languages

Hardware Acceleration

What hardware you use matters

- CPU computations are slow
- GPU computations are faster
- Embedded/On chip computations are fastest

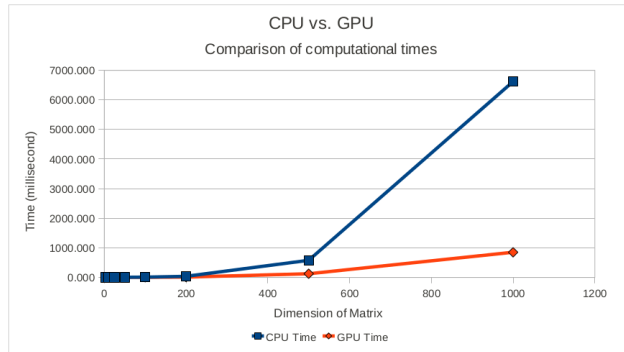


Figure is of element wise matrix multiplication