


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.

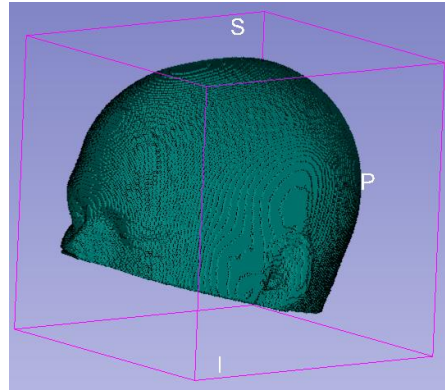
Defining Objects

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 \pm voxel size.

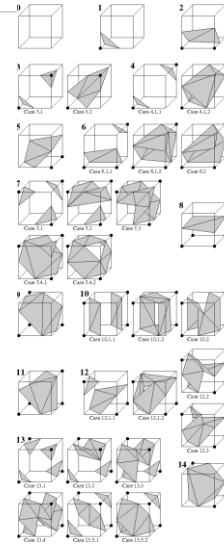
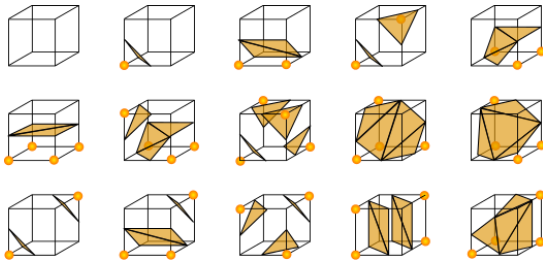
Image To Surface

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

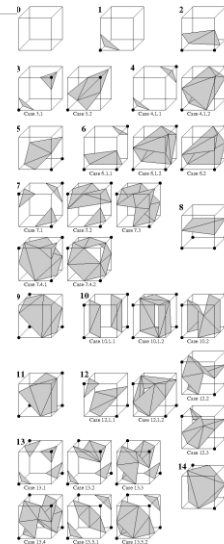
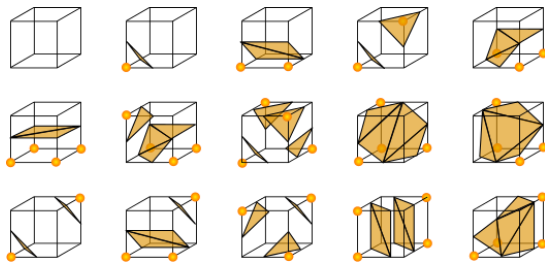
Image To Surface

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

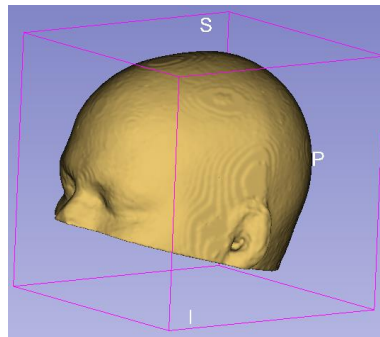
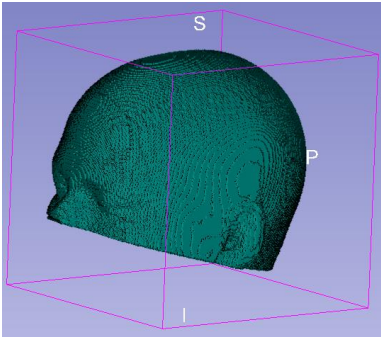
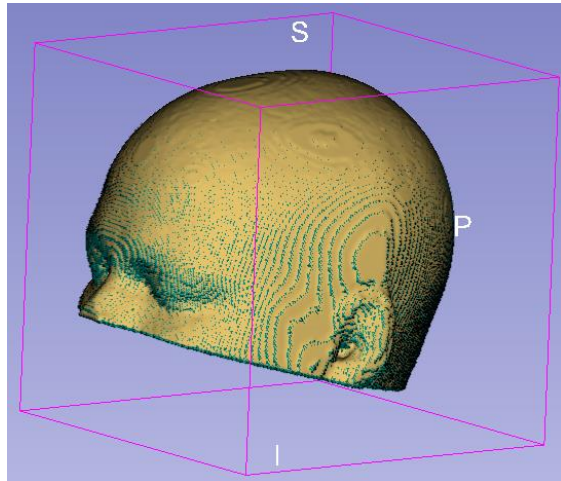


Image To Surface

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

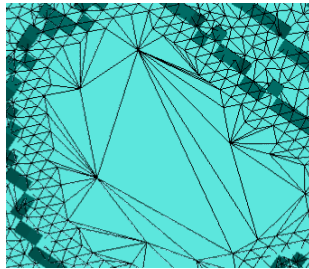
Mask has : 14417920 pixels
Surface has: 99543 polygons



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](#))

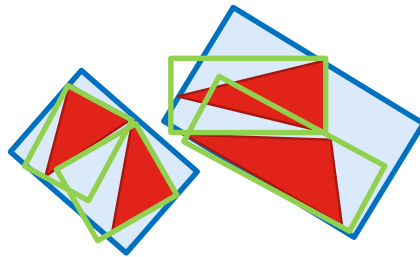


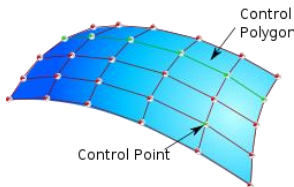
Image to Points

Implicit surface

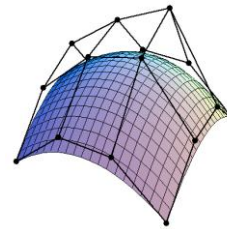
$$f_{surface} = \sum \phi(p_{query}) = 0$$

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

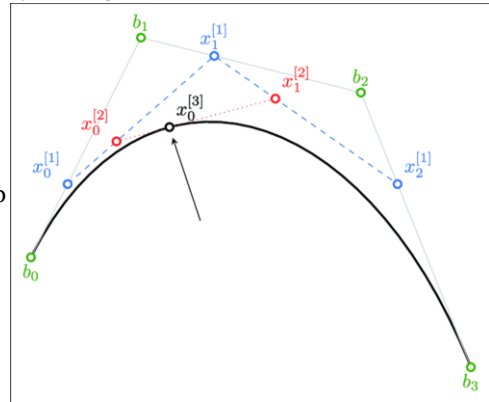
NURBS collision detection: F. Page and F. Guibault, "Collision detection algorithm for NURBS surfaces in interactive applications," *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*, Montreal, Quebec, Canada, 2003, pp.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

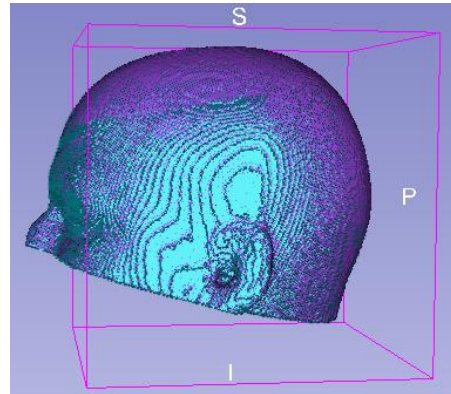
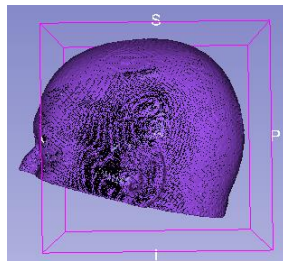
- Tree traversal can be used to identify region of curve to consider
- De Casteljau's algorithm can be used to iteratively define a bounding volume split
- For both octree and bounding volume each leaf node must contain enough points to correctly define surface patch in the region covered by the surface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

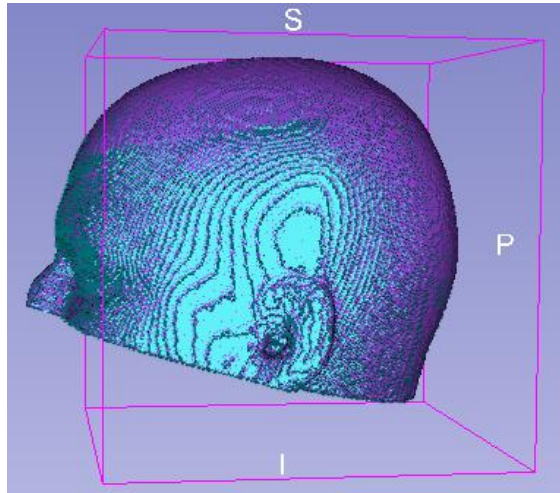


Klein, J. and Zachmann, G. (2004), Point Cloud Collision Detection. Computer Graphics Forum, 23: 567-576. doi:[10.1111/j.1467-8659.2004.00788.x](https://doi.org/10.1111/j.1467-8659.2004.00788.x)

Image To Surface

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

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



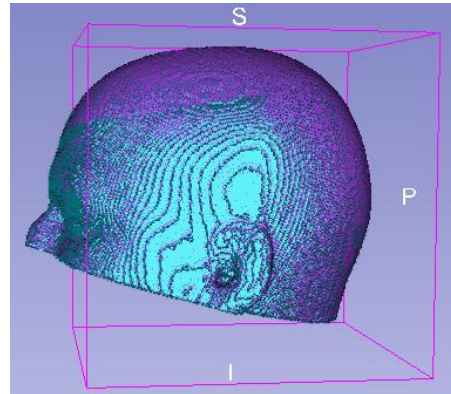
Point Cloud Collision Detection

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

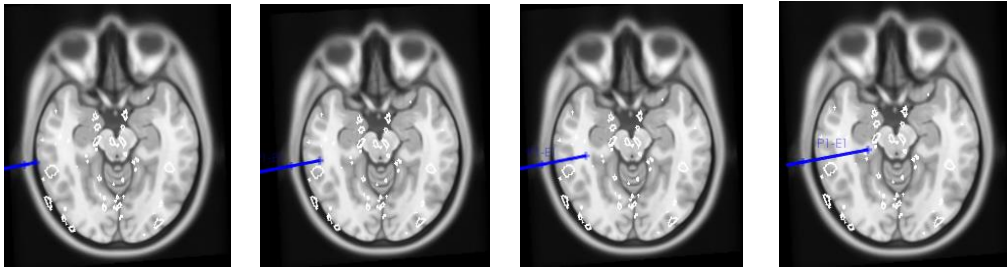
Defining States – Straight Tools

$s(t)$ 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

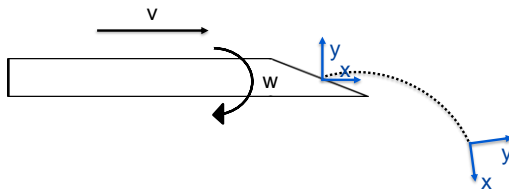
Defining States – Steerable Tools

$s(t)$ 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>

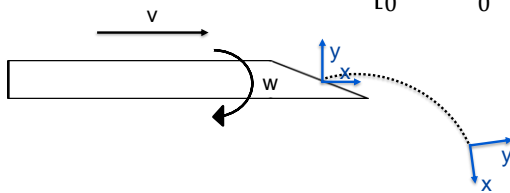
Patil, S., & Alterovitz, R. (2010). Interactive Motion Planning for Steerable Needles in 3D Environments with Obstacles. *Proceedings of the ... IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics. IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics*,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

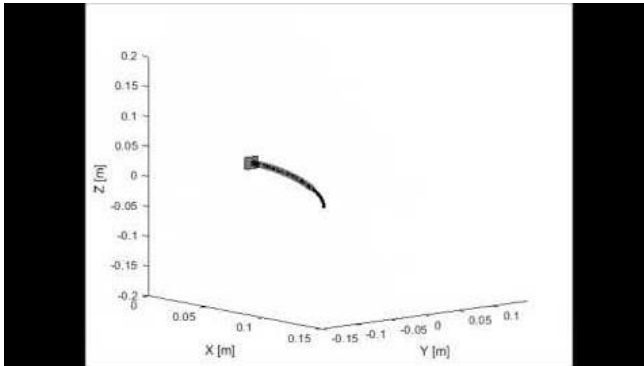
Webster, R. J., Kim, J. S., Cowan, N. J., Chirikjian, G. S., & Okamura, A. M. (2006). Nonholonomic Modeling of Needle Steering. *The International Journal of Robotics Research*, 25(5–6), 509–525. <https://doi.org/10.1177/0278364906065388>

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!

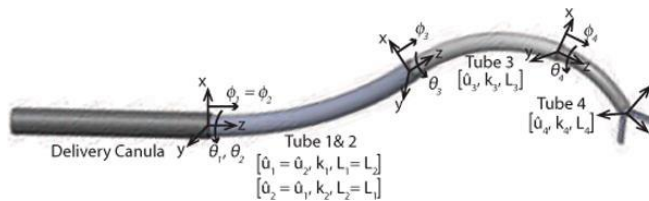
At least it is a piecewise Bezier function (each segment can be solved independently)

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

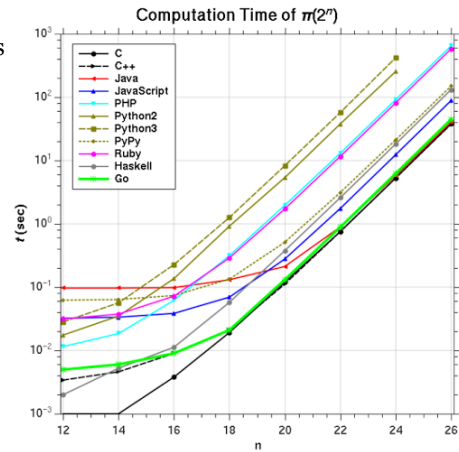


Figure is from () showing run time to compute digits of pi for a large set of programming languages

Hardware Acceleration

What hardware you use matters

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

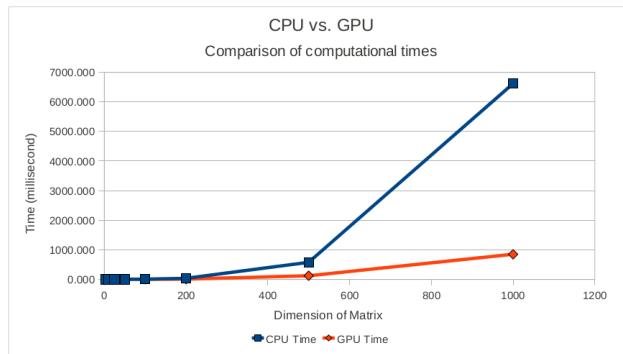


Figure is of element wise matrix multiplication