## Curve Representations

We begin with parametric and implicit curve representations. This is analogous to a coefficient and point representation of a polynomial respectively, in that parametric representations consist of a set of parameters which can each be modulated to fit the curve shape, and implicit curves consist of a finite set of points that make up the curve (which will be expanded upon in the relevant section). Parametric curves are used extensively to represent smooth curves with a small memory footprint. For example, the Bezier curve is utilised within digital typography to generate smooth glyphs composed of curved sections on large displays, such as the English letter ‘d’.

Implicit representations of geometry are used extensively in video game mechanics. This is because they can be used for collision detection between some pair of shapes, such as two spheres and their associated implicit functions, by directly solving a set of simultaneous equations where both functions evaluate to a particular value (in most cases zero). A very popular collision detection algorithm which is also used in Blender animation is known as AABB (axis-aligned bounding box) collision detection and relies heavily on implicit representation, and can be extended to more than just a pair of colliding objects. The parametric form is insufficient for this problem due to a lack of such a finite representation, having an infinite number of potential intersections to check.

In the ‘Curve Representations’ section, we were tasked to implement code to showcase the differences between implicit and parametric curve representations and their use cases. As described prior, implicit representations are preferable for collision detection, as they are defined with a finite set of parameters. To demonstrate this, an implicit representation of a circle was implemented by the following formula: (LATEX…)

Then, using **numpy’s** built in *linspace* and *meshgrid* functions, we generated a graph of 2000 evenly spaced points on the interval [-10, 10] on both the x & y axes for which the implicit circles could be evaluated. To find the ‘merged representation’ which will also include our intersections, we simply took the product of the two implicit circles which would then include the points of intersection.

Overall, it was demonstrated the parametric forms performed better for shape merging, maintaining a continuous figure-eight, whereas small precision errors in the implicit product led to some invalid values being produced at the intersections (which hence were not plotted). It was also demonstrated through this that implicit representations would be more effective for intersection checking.

Also, as mentioned prior, parametric representations are beneficial in the context of generating smooth and cyclical curves, which can be difficult to produce implicitly. To demonstrate this, an attempt at generating a cycloid curve (commonly used for 2-dimensional wheel motion animations) was made for both a parametric and implicit representation. The cycloid path was simple to produce with the following parametric equations (LATEX).

The implicit representation, however, was impossible to produce accurately. It could not maintain periodicity, and required a repeated copying over of particular curve segments in order to generate the cycloid.

## Cubic and Quartic Bezier Curves

The notebook then progresses to particular parametric curve representations, Bezier curves (cubic and quartic) and Hermite curves. As related prior, Bezier curves are used extensively in smooth curve generation, and another application area is in animation pathing. For example, if a VFX designer wishes to simulate a virtual camera travelling through space, a lack of continuity and ‘discrete jumps’ in the curve will be noticeable and unnatural, which is resolved by the use of the Bezier curve representation. Hermite curve representations are applied widely and are particularly useful where the user needs to specify the slope information directly by modifying the tangent vectors at each endpoint of the curve. A particular case where Hermite curves are used over Bezier curves is in CAD (computer-aided design) applications, since curve segments can be modified independently, enabling finer local control of the curve.

We then continued onto Cubic & Quartic Bezier Curves, where our task was simply to implement and display both types of curves. In order to understand how to create cubic and quartic cases, we began by looking at the general construction of a Bezier curve to an arbitrary degree.

Initially, a parameter array of points is produced using numpy’s *linspace.* This parameter array determines the resolution and smoothness of the curve. Then, a set of control points are initialised which will be used to effectively modulate the overall shape of the curve. The control points include the start of the curve, intermediary points and the endpoint of the curve. Then we iterate through our parameter array. For each unique parameter, we define the Bernstein basis polynomial, with the following formula (LATEX- ALSO MENTION USE OF SCIPY COMBINATORIAL FUNCTION).

This polynomial value is computed for every control point with the parameter and is used to create a running sum which evaluates to the final (x, y) position on the curve for that parameter. This in effect ‘pulls’ each parameter toward the control points. Once this process has been completed for every parameter, we have our final output curve.

Now, in order to define the cubic and quartic cases singularly, we complete the same process, except we define the Bernstein basis polynomials for degrees n=3 and n=4 explicitly. These are the explicit formulae (LATEX).

Finally, we must also ensure that we pass an appropriate number of control points to the Bezier curve algorithm, where control points should be equivalent to one more than the degree of the polynomial.

Below are some example cubic and quartic curves with various (randomly sampled?) control points to demonstrate the Bezier formalism in action (LATEX).

## Hermite Curves

The next section focused on implementation of Hermite Curves. Hermite Curves are defined in terms of four different components (start, endpoint, start tangent vector, end tangent vector). The start and endpoints defined the curve boundaries, and the tangent vectors are used to modulate the shape of the curve.

Similarly to Bezier curve implementation (since Hermite curves are also parametric), we first define a parametric array consisting of values *t*, in this case on the interval [0, 1] with some number of points defined (in our experimentation n=100).

We then define the four Hermite basis functions, which are used in this instance as we focused on cubic splines. These basis functions are themselves cubic polynomials defined as follows (LATEX).

The reason for using these basis functions is simple to derive, first starting with the general definition of a cubic polynomial (LATEX).

Then we apply the following constraints ( t = 0 => P(O) = P1, t = 1 => P(1) = P2, P’(0) = m1, P’(1) = m2) and solve the resultant system of equations (PLACE AS LATEX EQUATIONS).

A weighted sum of these basis functions with the endpoints and tangent vectors are then computed to calculate the point on the curve for each unique parameter. This is given by the following formula (LATEX).

In order to experimentally test the Hermite curve implementation, and demonstrate their flexibility by tangent adjustment, four subplots (with their curves) were generated. These subplots included:

* The original tangents.
* Exaggerated tangents (with the effect of steepening the curve).
* Vertical start and horizontal endpoint tangents.
* Symmetric (parallel) tangents.

(SUBPLOTS BELOW IN LATEX).

## Surface Extraction & Marching Cubes

We then progress to surface extraction, which generally means to obtain surface geometry. This will be discussed in more formal detail in the theory section. Surface extraction is applied extensively in context of various medical imaging techniques. For example, CT (computed tomography) is able to support analysis of particular anatomical structures (i.e. differentiating bone from soft tissues) by obtaining 2D slices at different density values (a form of surface extraction known as iso-surfacing).

A particular algorithm known as Marching Cubes is then implemented which is used regularly to provide slicing methodology noted above. The technical details of the algorithm and our implementation is discussed in the theory section also.

In surface extraction, we were first tasked to implement a simple function which could extract surfaces from a *scalar field*. A scalar field is a function which assigns singular values to points in a region of space, in our case to a particular geometry we are attempting to model and analyse.

In order to extract an objects surface, we need to first decide a particular surface function, which decides the points which sit on the surface. This effectively partitions the set of all points in a region of space into the interior (where the surface function evaluates to a sub-zero value), boundary (evaluating to zero), and exterior (surface function evaluates to a positive value).

In this task, we decided to use the surface function for a torus, which is formally defined as (… - WE CAN INSERT THE LATEX FOR THIS LATER).

Once this surface function has been defined, we then need to generate the region of space to partition, for this we utilised built-in functions provided by *numpy*, in particular *linspace* and *meshgrid*. A separate linear space is produced for the x, y and z axes respectively. The mesh grid function is then used in order to combine these axes to produce a final 300x300x300 voxel region to evaluate our surface function within.

Then, for the simple case, we only needed to evaluate the *zero-level set* of the surface function, that is where it evaluates to zero. However, since some values will be arbitrarily close to zero but not exactly zero, it was required we defined a reasonable tolerance range (in this case any value < 0.0001 is accepted as in the zero-level set). Finally, numpy’s built-in functions *isclose* and *where* allowed to select scalars in the surface function with a value close to < 0.0001.

Include 3-dimensional surface plots below in the final document…

In the second part of this task, we were asked to implement the Marching Cubes Algorithm as a more efficient and well-faithed representation of a surface.

The term ‘marching’ refers to the process of iterating over each individual voxel in generating the surface information. A triply nested loop is used to extract each voxel at a particular (i, j, k) position in a 3-dimensional scalar field parametrised by our *resolution*, which will also determine the fineness of the surface representation.

For each voxel, the coordinates of its centre and its corresponding vertices are extracted using this information as well as the spacing measurement between centre points, which is simple to compute from the resolution of the field. For example, the bottom-left-front vertex from the centre can be computed by subtracting the spacing from all three centre coordinates and halving them (the intuition for halving would be that the surface of each voxel sits on the midpoint boundary between it and its neighbouring voxel).

Next, whether the voxel is contained in the volume of interest or not is determined by comparing the iso-value (in our case zero since we define the surface on the zero-level set) with the density of the voxel. If the density is greater than the iso-value, it means that the voxel is outside of our volume of interest and we discount it in actual extraction.

However, if our voxel is in our volume of interest, we now extract the surface by triangulating the voxel depending on a set of possible configurations. These configurations are whether each individual vertex of the voxel is inside or outside the iso-surface (full or empty respectively). In the three-dimensional case, there are 8 vertices, each of which can either be in the empty or full state, generating 2^8 = 256 unique triangulations. Once these cases have been generated, they will serve as a lookup table for selecting the appropriate triangulation to apply to a voxel.

In order to apply the triangulation, a trilinear interpolation is used. Interpolation is completed for each individual edge of each individual triangle produced as a result of the previous step. Firstly, the absolute differences between the densities of the vertices constituting the edge are checked, if it is arbitrarily close to zero (in our case <0.0000001), then we simply return the first point in the pair (as there is no gradient to interpolate). Next, we compute t, a parameter that represents how far along the line segment between p1 and p2 we need to interpolate to reach the iso-value (iso) (based on the following formula LATEX).

We then perform a weighted interpolation over all three axes for our vertices in terms of the generated parameter t.

One note for a further improvement is that not all triangulation cases need to be stored explicitly, as the 256 configurations can be reduced to 15 unique cases as a result of rotational symmetries. However, the constant space complexity difference with ~200 case reduction is negligible and performance losses are only noticeable with particularly small volumes.

## Subdivision Surfaces and Catmull-Clark

The final section considers a technique known as ‘subdivision surfaces’. Generally speaking, this refers to the complexification of mesh geometry through the repeated subdivision of surface information into a collection of smaller but still topologically valid shapes, most commonly triangles or quadrilaterals. These techniques are used extensively in the development of complex mesh geometry, such as Pixar animated characters and simulations of astrophysical objects, since a simpler ‘control’ mesh can be repeatedly refined to a target mesh by application of these techniques. This reduces complexity and effort spent in the design phase. The bonus task then directed us to implement a well-known and commonly implemented surface subdivision algorithm known as Catmull-Clark subdivision. Again, the technical details of the algorithm are also discussed in the theory section.

In subdivision surfaces, our first task was to use a pre-built library to implement subdivision, utilising this as a demonstration for how a more complex animated character (complex mesh geometry) can be generated from a simpler, low-polygonal control mesh. For this we utilised the *trimesh* library, which provided functionality for loading pre-defined control meshes, as well as an in-built implementation of the Catmull-Clark subdivision algorithm.  
  
Firstly, a low-polygonal control mesh of a rabbit is loaded from an ‘.obj’ file (more detail needed).

The initial mesh is then shown (DISPLAY INITIAL RABBIT MESH BELOW).

Following this, the *remesh.subdivide\_loop* method provides the functionality for continually subdividing the rabbit mesh surface using Catmull-Clark. Multiple rounds of subdivision and the subsequent results for the mesh are displayed below, and these demonstrated the improvement of the visual quality and smoothness of the rabbit model, to a level of granularity that would not be perceptible for the casual viewer of an animated film which may utilise such a model as a background object (i.e. a digital scene displaying nature with rabbits interspersed in the grass). (DISPLAY SMOOTHED MESHES BELOW).

Finally, as a bonus task, we were directed to implement the Catmull Clark subdivision algorithm without using external libraries such as *trimesh*, however, the *numpy* library was used extensively for efficiency.

The Catmull Clark algorithm first begins by computing a set of face points, which are in essence the mean average over the set of vertices that contribute to a face. This was simple to implement as the *subdivide* method expected to have an instance attribute named *self.faces* which was constituted of a list of sublists, each containing the contained vertices as references to another instance attribute named *self.vertices*. The vertices could be extracted by these references (indexes) and then averaged using numpy’s built in mean functionality (with axis=0). In this same section of the code, a vertex -> face mapping was produced to simplify later calculations.

Next, the algorithm will produce edge points. It first generates all possible edges using the face information, assuming that the vertices in each face are stored in *counter-clockwise winding order*, which is the case in the initialisation of the example unit cube provided. It does this by simply pairing each adjacent set of vertices (and makes use of the modulus function in order to pair the last and first element of each face).

Now, provided we have generated the edges correctly, we will have two distinct cases to handle for edge points, where the edge is *bordering a* ***hole,*** and where it is not. A hole is simply referring to a gap in the mesh surface. Edge detection for holes is simple through determining the number of faces the edge belongs to. If this is only one face, then the edge is bordering a hole, otherwise it is not. In the case where the edge is bordering a hole, we generate the edge point as the mean of the vertices constituting that edge (the edge centre). Otherwise, we generate the edge point by computing the mean of the face points of each face containing the edge, alongside the edge centre (mean of vertices constituting the edge).

Next, we compute the locations of the new vertices of the subdivided mesh. This again is partitioned into the case of vertices bordering versus not bordering a hole. A vertex is considered to be bordering a hole if the number of faces containing the vertex is *not equivalent* to the number of edges containing the vertex. In the case where the vertex is bordering a hole, the new position is simple to compute by summing each old position (for each axis) by the average of the edge centres which are bordering a hole, and dividing this by two to generate a mean position. The computation for vertices not bordering a hole is more involved and is determined by the following formula (LATEX).

Finally, each face of the original mesh is replaced by a quadrilateral face, but both cases (triangular or quadrilateral faces in the original mesh) must be dealt with separately. Three new quadrilateral faces replace the original triangular face, and four new quadrilateral faces replace the original quadrilateral face. The formulae for these (in terms of edge segments, face points and edge points are provided below:

The effectiveness of the algorithm was tested using three repeated applications of subdivision on the unit cube, as this was simple to compare against other visual examples of a correct Catmull-Clark implementation. This yielded, as to be expected, a continuous refinement of the original cube into an evenly subdivided spherical shape.