CS 779
Splines and Their Uses in Computer Graphics

Stephen Mann

Winter 2020

Contents

Preface

These are course notes for the University of Waterloo course Computer Science 779 (Splines and Their Uses in Computer Graphics). They are a companion to my monograph "A Blossoming Development of Splines." The monograph contains basic material about splines, while these course notes contain additional material not covered in the monograph, as well as the details.

Stephen Mann
Waterloo
December 2009

Chapter 1

Administration

1.1 People

Instructors:	Stephen Mann	DC 2106 x34526	smann@uwaterloo.ca
	Office Hours	TBA	

1.2 Resources

Texts:

A Blossoming Approach to Splines,
by Stephen Mann, Morgan-Claypool, 2006

Curves and Surfaces for CAGD,
by Gerald Farin, fifth edition, Academic Press, 2001

Course Homepage: http://www.student.cs.uwaterloo.ca/~cs779/

References: *Pyramid Algorithms,*
by Ron Goldman, Morgan Kaufmann, 2003

Curves and Surfaces in Geoemtric Modeling,
by Jean Gallier, Morgan Kaufmann, 2000

A Practical Guide to Splines, Carl de Boor.

An Introduction to Splines,
Richard Bartels, John Beatty, and Brian Barsky.

Fundamentals of Computer Aided Geometric Design,
Josef Hoschek and Dieter Lasser.

Blossoming: A connect the dots approach to Splines,
Lyle Ramshaw.

The last is a DEC SRC technical report. It has been scanned and is available on the web `http://research.compaq.com/SRC/publications/src-rr.html` as SRC-019. I have also made a local copy:

`http://www.cgl.uwaterloo.ca/~smann/Papers/SRC-019.pdf`

1.3 Background

To take this course, you should have a reasonable background in mathematics. In particular, I will assume you know the basics of linear algebra, and have the mathematical sophistication to understand new ideas when they are introduced.

The assignments will be C programming using fltk for the user interface. You are free to use Java instead, but sample code will only be provided for fltk/C++. While you can easily learn the required fltk needed for the assignments (we don't use much of it), you will have troubles if you do not know C or C++.

One extra caution: it should be possible to do the programming assignments for curves on linux, Windows, Mac, or Android although the provided code is primarily for linux and Android. However, for the surface portion of the course at the end, I provide a polygon viewer that only works on linux. This viewer is available for download to linux machines, and is on the machines in MC 3007 (the undergrad graphics lab) which you'll be able to login to. While you aren't required to use my polygon viewer, if you do not have a way to view polygons (with surface normals!) on your machine, then you'll have great difficulties unless you have some linux background and can use the machines in MC 3007..

1.4 Course Goals

The main goal of the course is to give students an understanding of splines, including Bézier and B-spline curves and surfaces. A further goal is to do so in a geometric way, and to develop your geometric intuition and reasoning.

1.5 Marking

The grade will be based on homeworks (60%) and a final project (40%). The homeworks will be part theoretical and part programming. The purpose of your project will be to learn something and show what you learned.

Note that 20% of your mark on the homeworks and on the final project will be based on the quality, clarity, and conciseness of your exposition.

1.5.1 Audit Credit

If you wish to take the course for audit credit, you will be expected to do the non-starred, non-programming questions on the assignments but you do not have to do a final project. For assignments that are only programming questions and/or starred questions, students taking the course for audit credit do not have to submit anything.

1.6 Assignments

There are eight assignments. While the points for each problem are noted on the assignment, 20% of the mark will be awarded for presentation. Thus, your writeups should be clear, concise, and easy to understand.

Note that it is NOT the case that each assignment is worth 7.5% of your grade; instead, the sum of the points on all five assignments is worth 60% of your grade.

Late assignments are accepted up until the lecture in which I discuss the assignment (usually the lecture following the due date). However, a penalty of 1 point per day late will be assessed.

The assignments and their due dates will be posted on the course webpage.

1.7 Project

The course notes describe the final project. You will either give a demonstration or a talk for your project, either of which will be open to the entire class. The project is worth 40% of your grade; 5% will be based on a project proposal presentation you give about your project to the class, and 35% is for the project itself.

The project proposal presentations will be in mid- to late-March. The structure of your proposal should be similar to the following:

- Introduction (problem statement, etc.)
- Discussion (mathematical details about the problem, etc.)
- Proposal (what you intend to do, what you expect to learn).

1.8 Exams

There are no exams for this course.

1.9 Documentation

Documentation for various things can be found in the course web page.

1.10 Data

Data files needed for the assignments can be found in the course web page.

1.11 Project

Your final project in this course will be on a CAGD topic of your choosing. Your project is due by the last day of examinations (although I might extend this a bit). You will need to arrange a time with the instructor for giving your presentation.

You have two choices: you can choose an implementation project or a written project. Both will have a presentation component open to the rest of the class. For a written project, this involves giving a talk on the subject you research. For an implementation, this probably consists of a demonstration of your project, although for some implementations, a talk might be more appropriate.

The following are ideas for projects:

- Surface
  - Subdivision Surfaces
  - Polynomial Least interpolation
  - C^1 surfaces via the hyperbolic plane ("Topological Design of Sculptured Surfaces," Ferguson, Rockwood, Cox, SIGGRAPH 1992)
  - Functions over the Sphere (Schumaker, CAGD June 1996)
  - N-side patches via Base Points (Warren)
  - S-patches (Loop, ACM TOG, July 1989)
  - Sabin patches (plus 1 level of subdivision) (Sabin, Eurographics 1983(?))
  - Zheng-Ball patches (plus 1 level of subdivision)
- User Interfaces
  - Surface Pasting (Bartels et al)
  - Free-form deformations (SIGGRAPH, Sederberg 86, Hsu 1992)
  - Hierarchical B-splines (Bartels, Forsey)
- Other
  - Wavelets
- Hard stuff
  - T-NURBS, etc., (Sederberg, SIGGRAPH 2003)
  - A-patches (Bajaj, ACM TOG April 1995)
  - Triangular B-splines (Seidel)

The point of the project is for you to learn a CAGD topic on your own. Thus, if I'm satisfied with what you learned, you will receive an 65 for your project. To get above an 65, you will need to go beyond learning something in the literature. To get a mark above a 90 on your project, you would need to do some research.

A paper project would require looking at several papers, and synthesizing the ideas in these papers. In particular, you would want to address the following questions:

- How do the ideas in the papers compare?
- Which is better?
- What overlap do the papers have?
- Can ideas from multiple sources be combined to produce something better?
- What are the next research steps?

To get a project mark above 90, you would need to actually do some of the next research steps.

For an implementation project, things that will push your project mark above 65 include

- Running experiments to test the ideas that you studied.
- Use your project to build nice models, etc.
- Implement and merge several ideas. Note: The ideas should be significantly different to get marks for this, although you could get experiment marks for implementing two similar ideas and comparing them via experimentation.

To get a project mark above 90, you would have to implement something new (which probably means deriving something new).

Note also that you will not get any project marks for learning any CS488/688 material.

1.11.1 Project proposal

You will give a short talk about your proposed project. You have three objectives for your presentation:

1. Presentation of the high level ideas. What is the general topic you will be working on? It is not expected that you will give many details about it (you won't have learned it yet and you won't have time in this presentation), but you should be able to convey the general idea.
2. What you plan to do for your project — the basics that you will implement (or study) and ideas for extras.
3. Staying within your time limit. There is a 1 point per minute penalty for going overtime. You should rehearse your presentation to get within the time limit.

If you need any audio-visual support for your presentation, you should let the instructor know at least 1 day (and preferrably more) in advance.

1.11.2 What to hand in — implementation project

You give me (at your demo) a 1 page summary of your project. I will use this page to make notes during your demo to decide what you should get points for. So on this one page, you should tell me what you learned, and tell me what you feel I should give you marks for. "Point form" is preferred here. You should also give me the URL for project your web page (described in the next paragraph).

You should also write a short web page (about 1–3 pages when printed) describing your project with pictures. Further, if you do any experiments, give a short writeup of what the experiment is and of the results of the experiment. Be sure to include a bibliography listing the main sources for your project.

You may combine the 1 page summary that you give me at your demo with your web page, and base your web page around this summary.

1.11.3 What to hand in — writing project

Your writeup should be 5–10 pages, with an extra sheet saying what you learned and what you deserve marks for. In your presentation, you do not need to say what you deserve marks for; rather, you should present the highlights of the paper you have written.

Chapter 2

Polynomial Curves and Surfaces

See “A Blossoming Development of Splines”

Chapter 3

Debugging

Debugging splines is pretty much like debugging anything: you trace the execution of your code on known input and check that you're getting the desired output. Indexing and off-by-one errors causes many problems, so tracing loop indices is also helpful. A few observations can be made as to some simple checks that catch many of the errors.

3.1 Bézier Curves

de Casteljau's algorithm is reasonably easy to debug. The main concern is going through loops too many or too few times. The simplest checks involve using a curve with control points of (0,0), (2,0), (4,0), (6,0) and evaluating at $t = 0.5$. At the intermediate steps, you should get the points (1,0), (3,0), (5,0), and then (2,0),(4,0), and finally (3,0), since your blending weights should be 0.5,0.5 when $t = 0.5$.

A second check is to evaluate the curve (0,0), (3,0), (6,0), (9,0) at $t = 1/3$. Then your intermediate points should be (1,0), (4,0), (7,0), and then (2,0), (5,0), and finally (3,0).

3.2 B-splines

B-splines are harder to debug than Bézier curves due to the more complex indexing. A simple first test is to set the knot vector to $\{0, 0, 0, 1, 1, 1\}$ (or possibly $\{0, 0, 0, 0, 1, 1, 1, 1\}$), in which case you should get a Bézier curve. After that, though, you should print the indices into the knot vector and see that they match the recurrence formula.

3.3 Surface

The added complication in debugging triangular Bézier surface patches is the additional index. The things to check are

- Make sure your barycentric coordinates are correct (they should sum to 1 if nothing else).

- Evaluation at the corner of the domain gives nice intermediate points, since they will all be initial control points.
- Use a patch with constant z-value and xy-coordinates at uniformly spaces integer locations, and evaluate at (a) the corner; (b) an edge midpoint; (c) at barycentric coordinates (1/3,1/3,1/3).

Chapter 4

Miscellaneous Curve Topics

4.1 Convex Hull Root Finding

1. There are many methods for finding roots of polynomials. One due to Sederberg works with univariate polynomials in the Berstein basis. The idea illustrated in the top row of Figure 4.1. We compute the convex hull of the control polygon and find its left most intersection with the x-axis. By variation diminishing, we know that the root can't lie to the left of this point on the x-axis, so we subdivide the curve at this point (Figure 4.1, bottom left), and repeat the process until we get a root within our desired tolerance (Figure 4.1, bottom right, where three steps were need to find the root to within 10^{-4}).

2. Unfortunately, as shown in Figure 4.2, the algorithm runs into problems when the curve is nearly horizontal near the x-axis. Alternative methods must be used to handle cases such as these.

3. Some additional notes:

   (a) The algorithm can be used to find all roots of a polynomial in increasing order. The idea is to find the left most root, then construct the polynomial with this root removed and repeat.

   (b) Note that one doesn't actually need the entire convex hull: just one quarter of the hull and often not even that. All that's needed is the part of the hull from the left most point to the x-axis.

   (c) It appears that there are two cases that are needed: one when the left most control point is above the x-axis and one when it is below the x-axis. But by negating the y-values of the control points, one case can be converted to the other.

   (d) The algorithm can be extended to compute the intersection of a ray with a tensor product Bézier surface.

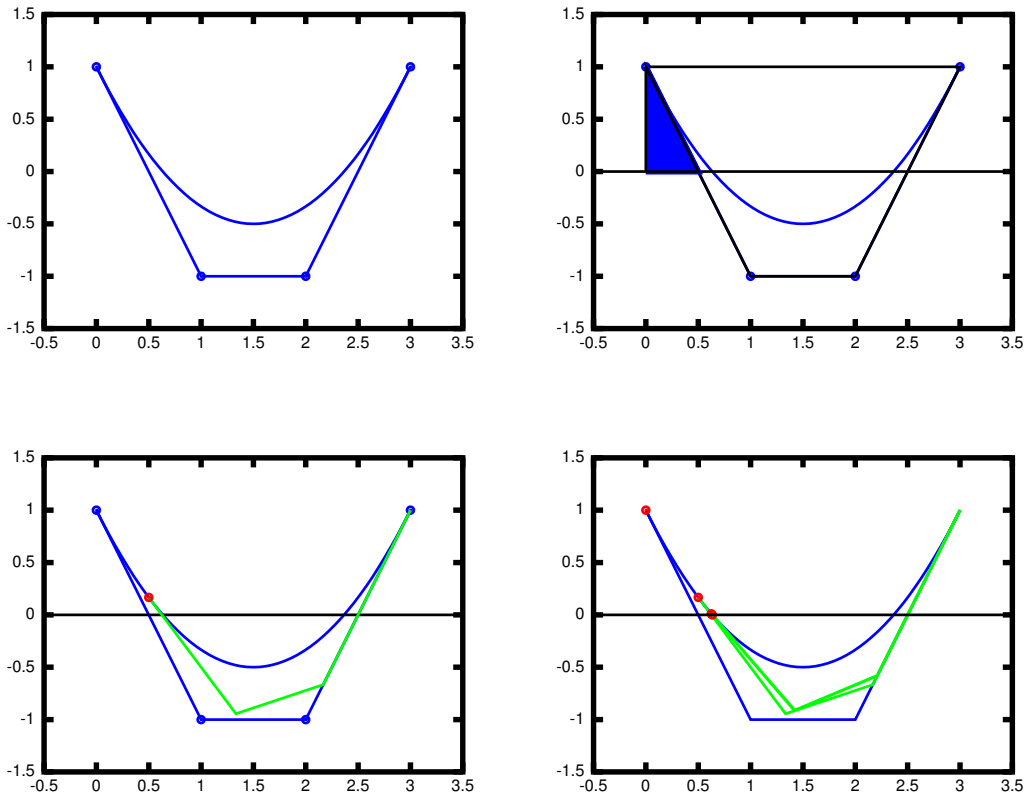


Figure 4.1: Using the convex hull to compute roots of a polynomial.

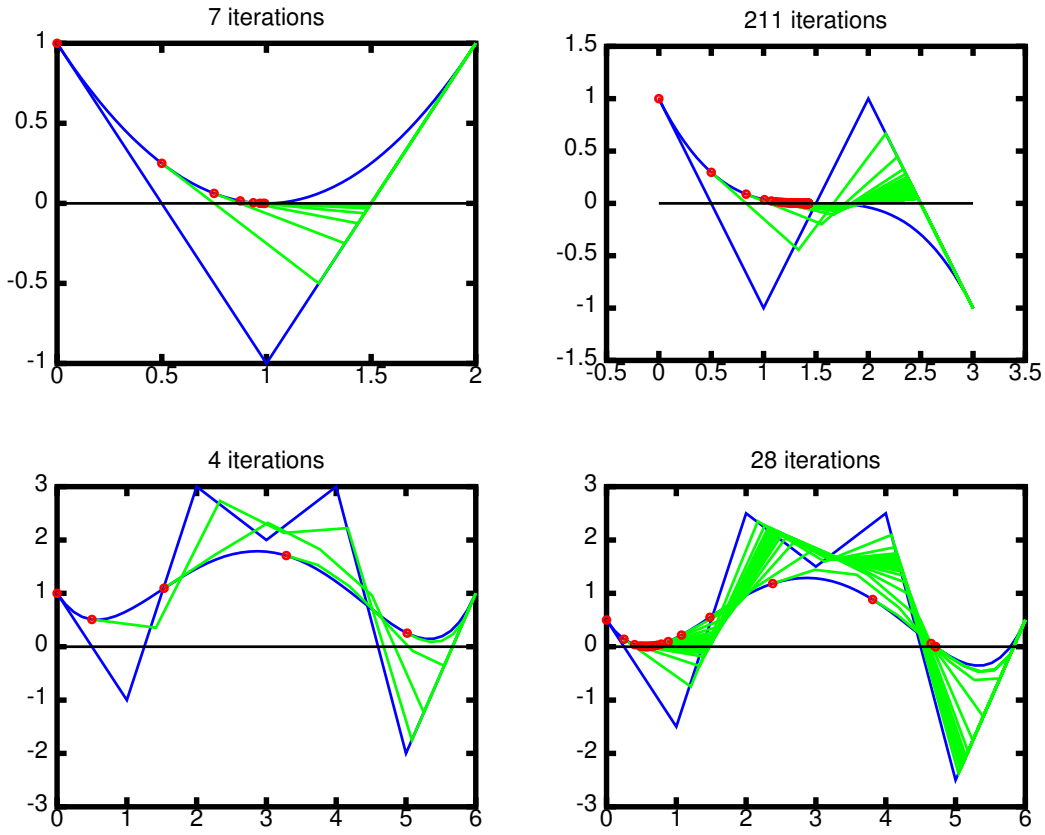


Figure 4.2: Examples of using the convex hull to compute roots of a polynomial to compute the root to with 10^{-4}.

References

- Thomas W. Sederberg and Scott R. Parry. A comparison of curve-curve intersection algorithms. Computer-Aided Design, 18:58–63, 1986.
- Melvin R. Spencer. Polynomial Real Root Finding in Bernstein Form. PhD thesis, Brigham Young University, 1994.

4.2 A Geometric Interpretation of Bézier Stability

Based on a talk by Helmut Pottmann,
4th Dagstuhl Workshop on Geometric Design

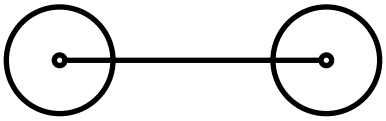
1. Bézier Curves:

   Numerically stable over $[0, 1]$ – what does this mean?

   Floating point errors, data gathering errors

2. Given: two points P and Q, where P and Q have some error ϵ

   Where will linear interpolation be relative to exact P and Q?

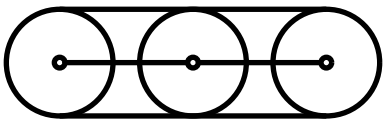


3. Mathematically:

$$(1-t)(P+\vec{p})+t(Q+\vec{q})=(1-t)P+tQ+(1-t)\vec{p}+t\vec{q}$$

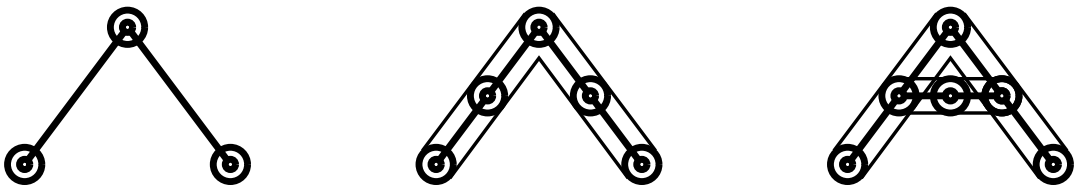
   Error is maximized when $\vec{p}=\vec{q}$:

$$(1-t)P+tQ+\vec{p}$$



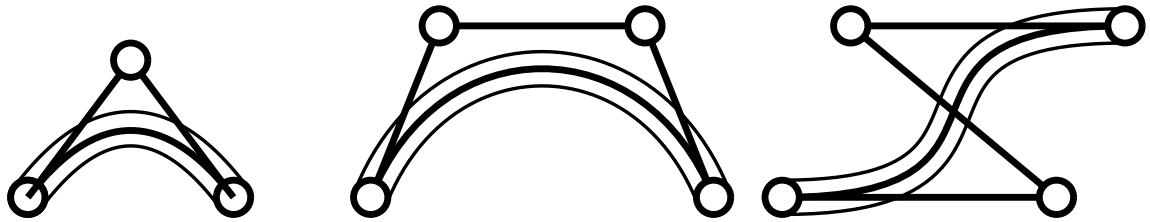
4. **Linear interpolation does not magnify error**

5. de Casteljau's algorithm is repeated linear interpolation:



6. **Repeated linear interpolation does not magnify error**

7. Over $[0, 1]$, Bézier curves have good numerical properties



8. Error in curve is no greater than error in control points

9. Also true of B-spline basis

10. Monomials

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 \cdot 1 + a_1 x + a_2 x^2 + \dots$$

11. Geometric interpretation:

    To form affine combination, note first basis function is 1

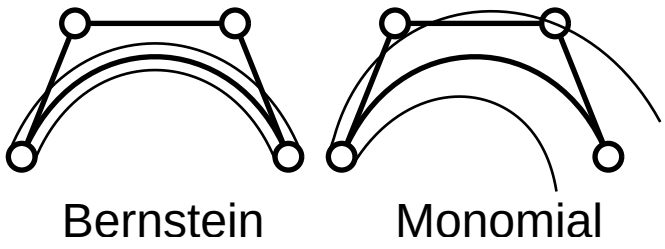
    Therefore, first coefficient is point, rest are vectors

$$P(t) = P_0 + \sum_{i=1}^{n} \vec{v}_i t^i$$

12. Error of ϵ in all coefficients:

    $P(0)$ has error of ϵ

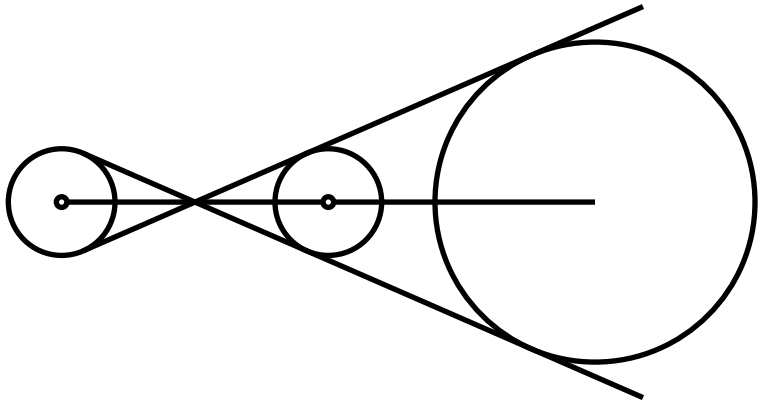
    $P(1)$ has error of $(n + 1)\epsilon$



13. Error is magnified in monomial basis

    Error worse for high degree

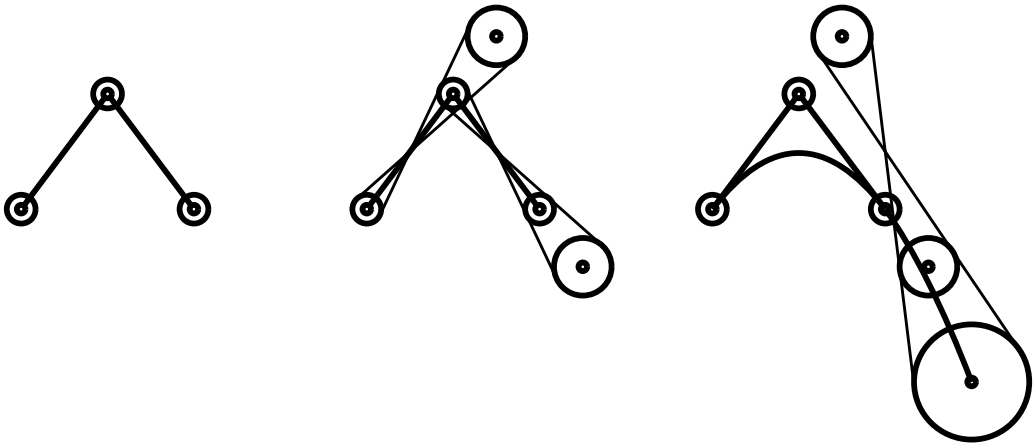
14. For extrapolation, connect lines from pairs of points in circles



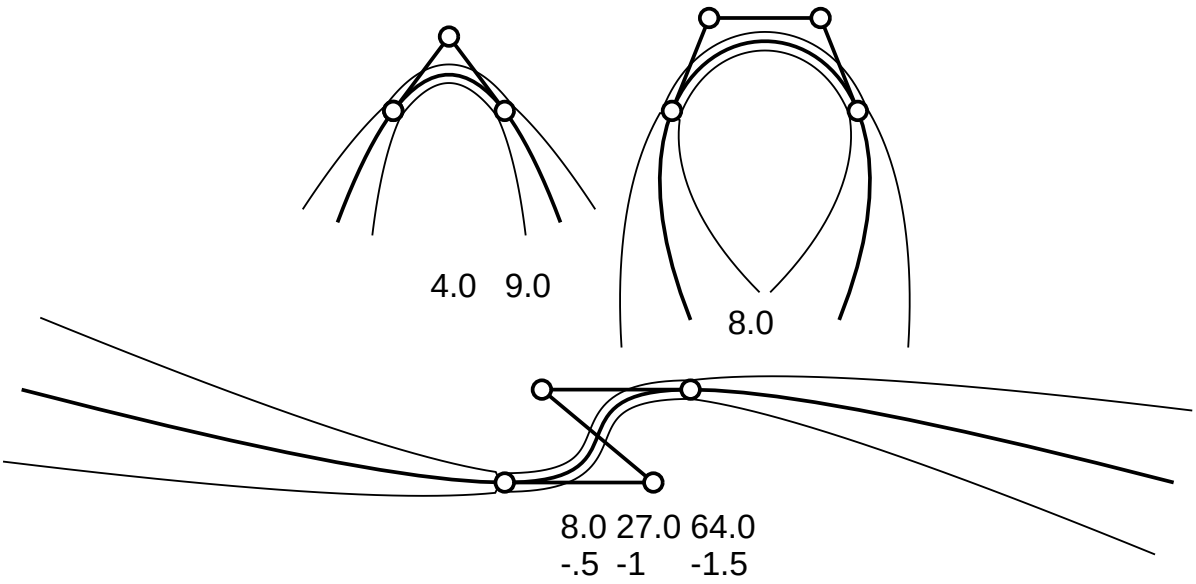
15. Error is linear with distance from midpoint

16. **Extrapolation magnifies error**

17. Repeated linear interpolation repeatedly magnifies the error



18. Bézier Curves Go Wild



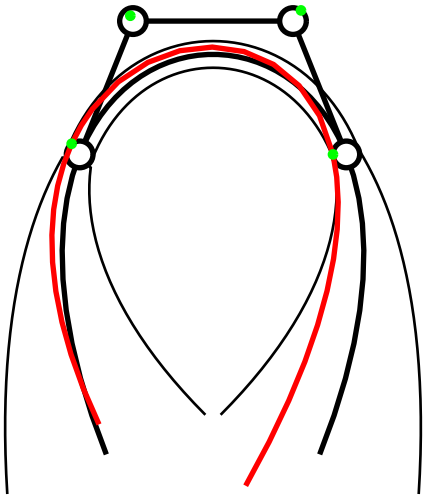
19. How Bad Is It Really?

    - Error increase as $O(d^n)$, d = distance, n = degree
      (distance is relative distance)
    - Same big-O error regardless of basis
      (but constant better in some bases than others)
    - Degree 3: at 10^5, error of 10^{-16} increases to 1
    - Degree 5: at 10^3, error of 10^{-16} increases to 1
    - High degree unstable
    - **This error is in addition to any other floating point errors**
    - Relative vs Absolute error

      $$10(1+\epsilon)x^n$$

      For $n = 15$, evaluate at 10.
      Error is $10^{16}\epsilon(= 1)$, but value is 10^{16}
    - Worst case relative error may be acceptable
    - Error may cancel
    - Beware of 0's at large values of t!

20. Any particular curve is well behaved



Green are CPs for red curve

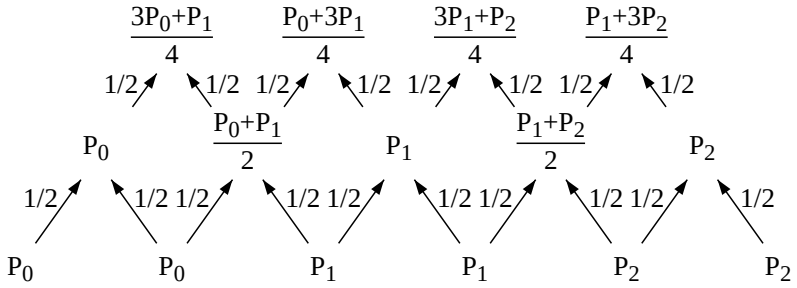
21. Start with 10^{-16} error, evaluate cubic at 10, error is 10^{-13}

    Probably good enough

4.3 Lane-Riesenfeld Algorithm

1. Lane-Riesenfeld Algorithm

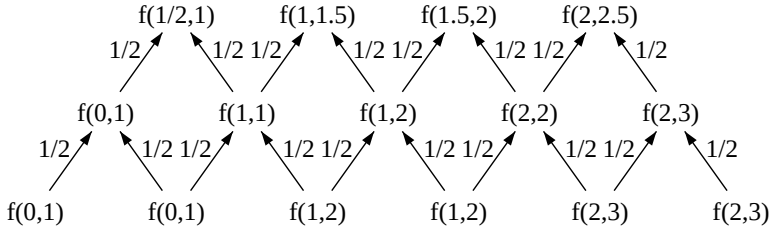
   - Quadratics: Replicate each control point, average twice.



     Original knots 0, 1, 2, 3

     New knots 1/2, 1, 1.5, 2, 2.5

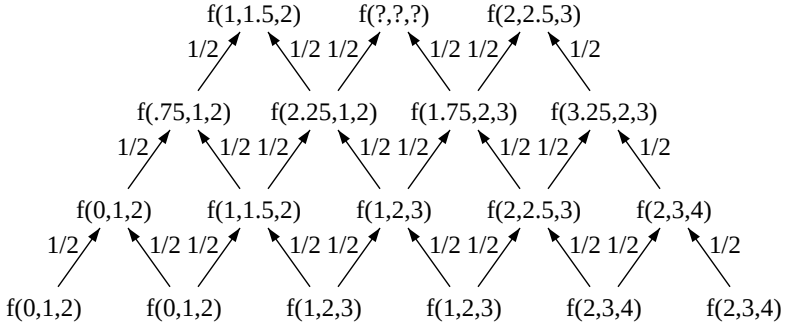
   - For degree $n > 2$, average n times.
   - How do we prove this?
     (Goldman, Warren, 1993, induction, de Boor recurrence)
   - Blossoming Lane-Riesenfeld - Quadratics Relabel quadratic triangle diagram with blossom labels:



Original knot vector: 0,1,2,3
New knot vector: 1/2, 1, 1.5, 2, 2.5

2. Cubics

When we try with cubics...

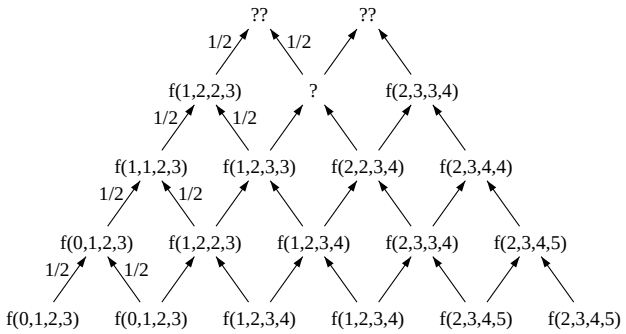


Rewrite...

$$\begin{aligned} f(?,?,?) &= \frac{1}{4}f(1,1.5,2) + \frac{2}{4}f(1,2,3) + \frac{1}{4}f(2,2.5,3) \\ &= \frac{1}{4}f(1,1.5,2) + \frac{3}{4}\Big(\frac{2}{3}f(1,2,3) + \frac{1}{3}f(2,2.5,3)\Big) \\ &= \frac{1}{4}f(1,1.5,2) + \frac{3}{4}f(1.5,2,3) = f(1.5,2,2.5) \end{aligned}$$

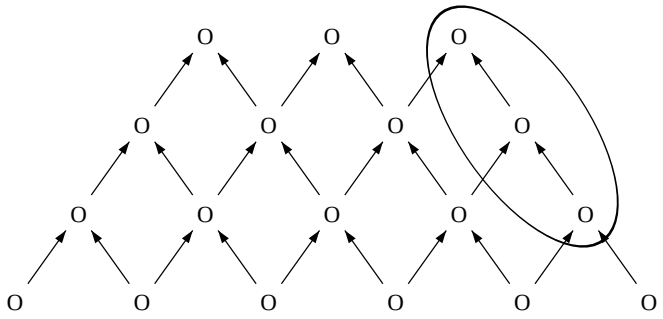
3. Quartics and Higher

Similar problem occurs with quartics



Solve in similar way

4. General Proof
   - Uses explicit formula for interior nodes
   - Uses induction
   - Lots of $\sum$s, lots of "n choose i"s
   - 6 slides full of equations
5. Computational Comparison to uniform sampling with de Boor evaluations
   (a) Does Lane-Risenfeld reduce evaluation costs?
   Although there is a "quadratic" in degree amount of work done to compute the first point in the refined control polygon, each additional point requires only a linear amount of work as indicated by the three points circled in the cubic diagram below:



   To obtain a smooth approximation to the spline, we need to refine at least three times for cubics. Each refinement roughly doubles the number of control points. For cubics, the cost of the top level is 3 affine combinations; the middle level is 1.5 affine combinations, and of the bottom level is 0.75 affine combinations for a total of 5.25 affine combinations. Compare this to the de Boor algorithm, which requires 6 affine combinations for each evaluation of the B-spline. Note also that the de Boor weights are more expensive to compute than the 1/2 weights of Lane-Riesenfeld.

   (b) As the degree increases, the cost benefits of the Lane-Riesenfeld algorithm improve, since the cost per sample point is $O(n \log n)$ while for de Boor it is $O(n^2)$, with the de Boor algorithm having an additional cost to compute the weights.

   (c) For low degree, the primary reason for choosing the Lane-Riesenfeld algorithm is code simplicity. The de Boor algorithm allows you to handle non-uniform B-splines, and gives more flexibility in the sampling density, etc. Further, there may be memory access patterns that favor one algorithm over the other, and other issues likely arise if making GPU implementations of the algorithms.

6. Knots in Geometric Progression

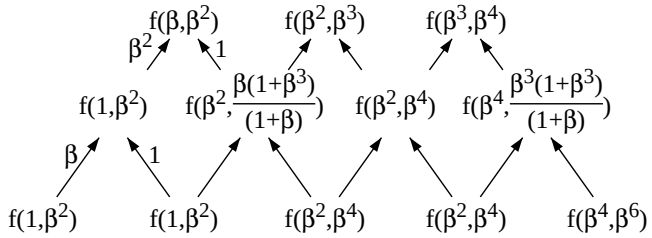
   Knots in Arithmetic Progression

   - Original knots = {0,1,2,...}
   - New knots = {0, 1/2, 1, ...}

   Knots in Geometric Progression

   - Original knots = $\{1, \beta^2, \beta^4, \ldots\}$
   - New knots = $\{1, \beta, \beta^2, \ldots\}$

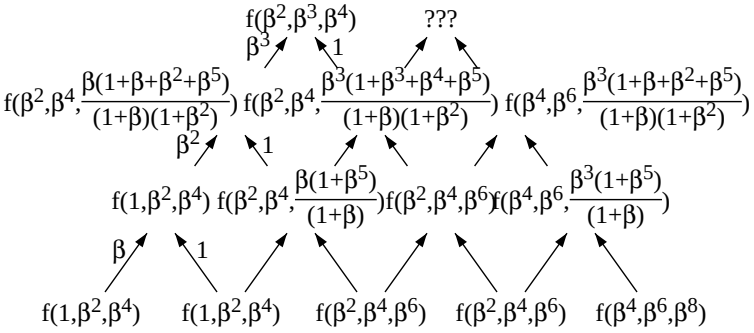
7. Quadratic Case



   Original knots = $\{1, \beta^2, \beta^4, \ldots\}$

   New knots = $\{1, \beta, \beta^2, \ldots\}$

8. Cubic Case



   Original knots = $\{1, \beta^2, \beta^4, \ldots\}$

   New knots = $\{1, \beta, \beta^2, \ldots\}$

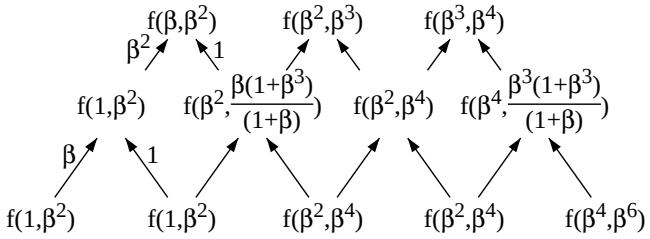
9. No Proof for Geometric Progression Lane-Riesenfeld

   - Everything is a lot messier in the geometric progression case

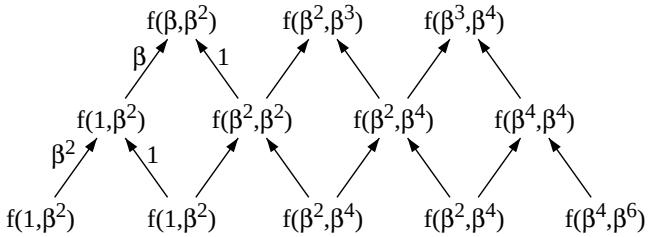
- Straight forward to do Ron's trick for cubic and quartic arithmetic progression (we know appropriate weights exist)
- The formula for each node is a recurrence relation
  Is there a closed form?

10. Can clean up quadratic case:

    Switch Edge Labels from this...



    To this.



    Doesn't help for higher degree

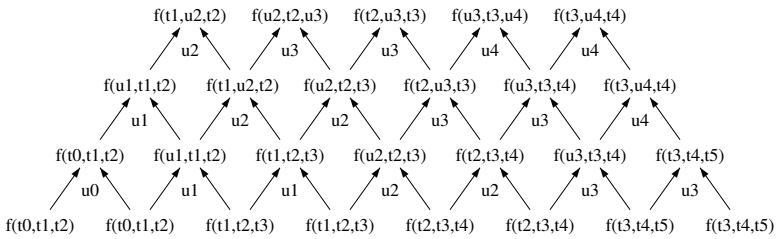
11. Schaefer-Goldman have addressed at least some of the above issues:

    (a) Given B-spline with arbitrary knot vector,

    (b) Duplicate control points,

    (c) At kth level ($k = 1..d$), insert knots $u_{k-1}, u_k, u_k, u_{k+1}, u_{k+1}, ...$

    As an example, we start with knot vector $\{t_0, t_1, t_2, t_3, t_4, t_5\}$ and we want to insert knots u_0, u_1, u_2, u_3, u_4 with $u_0 < t_0,\ t_0 < u_1 < t_1,\ t_2 < u_2 < t_2, ...$



Note that every other value (above the 0th layer) is copied from the previous layer (rather than being an affine combination).

Note also that since this cubic B-spline is defined over the interval $[t_2, t_3]$, the knots of the refined B-spline will be $\{t_1, u_2, t_2, u_3, t_3, u_4, t_4\}$.

References

- Schaefer S. and Goldman R., Non-uniform Subdivision for B-splines of Arbitrary Degree, Computer Aided Geometric Design, Vol. 26, No. 1 (2009), pages 75-81.

4.4 Degree Raising B-splines

1. Earlier saw how to degree raise Bézier curves. In general, we can show that given degree n polynonial F^n with blossom f^n, the blossom of the degree $n+1$ polynomial F^{n+1} with blossom f^{n+1} where $F^{n+1} = F^n$ is

$$f^{n+1}(u_1, \ldots, u_{n+1}) = \frac{1}{n+1} \sum_{i=1}^{n+1} f^n(u_1, \ldots, \hat{u}_i, \ldots, u_{n+1}), \tag{4.1}$$

   where the notation $\hat{u}_i$ means u_i is not an argument of the blossom.

   We can see that this is the appropriate blossom since (i) it is multiaffine; (2) it is symmetric; and (3) it agrees with F^{n+1} on the diagonal.

2. We can now apply this directly to B-splines. Given a B-spline F^n with knot vector $\{t_0, \ldots, t_M\}$ with all knots occurring with multiplicity 1. We wish to find the degree $n+1$ B-spline F^{n+1} where $F^{n+1} = F^n$.

   First realize that F^n is C^{n-1} and so F^{n+1} is C^{n-1}. This means that its knot vector should have all knots occurring with multiplicity 2. So the knot vector is $\{t_0, t_0, \ldots, t_M, t_M\}$. However, we will only have $n-1$ end knots (this requires an even/odd degree specification of the knot vector, since for n odd, all knots in the degree raised knot vector will have multiplicity 2, while for n even, the first and last knot will have multiplicity 1).

   We can now compute the B-spline control points of F^{n+1} by using Equation 4.1 and using consecutive knots from the degree raised knot vector as arguments.

3. Example: suppose we have a cubic B-spline with knot vector $\{0, 1, 2, 3, 4, 5\}$. The degree raised knot vector is $\{1, 1, 2, 2, 3, 3, 4, 4\}$. The control points are

   - $f^{n+1}(1, 1, 2, 2) = (f(1, 1, 2) + f(1, 2, 2))/2$
   - $f^{n+1}(1, 2, 2, 3) = (f(2, 2, 3) + 2f(1, 2, 3) + f(1, 2, 2))/4$
   - $f^{n+1}(2, 2, 3, 3) = (f(2, 2, 3) + f(2, 3, 3))/2$

- $f^{n+1}(2,3,3,4) = (f(3,3,4) + 2f(2,3,4) + f(2,3,3))/4$
- $f^{n+1}(3,3,4,4) = (f(3,3,4) + f(3,4,4))/2$

4. The main issue with degree raising B-splines is how to efficiently compute the control points of the degree raised B-spline. $O(nm)$ algorithms have been developed, where n is the degree and m is the number of control points.

4.4.1 Exercises

1. Prove that the above formulas give a generalization for the degree raising formulas for Bézier curves.

4.4.2 References

- W. Liu, "A simple, efficient degree raising algorithm for B-spline curves", Computer Aided Geometric Design 14 (1997) 693–698.

4.5 Degree Reduction

1. We have seen how to increase the degree of a Bézier curve (and of a B-spline curve). We can also ask: how can we reduce the degree of a Bézier curve? In particular, suppose we have a Bézier curve $P(t) = \sum_{i=0}^{n} P_i B_i^n(t)$ and we want to find the control points Q_i of a Bézier curve $Q(t) = \sum_{i=0}^{m} Q_i B_i^m(t)$ where $m < n$.

   It may turn out that P is a degree raised version of Q, in which case we can compute the Q_i through various methods (e.g., just reverse the steps of degree raising). However, in general the problem has no solution. For example, think of an arbitrary quadratic curve: there is no linear curve that matches it exactly.

   Instead, when we degree reduce we have to settle on a solution Q that is an approximation to P. For example, we could require that the solution minimize the L_2-norm between the curves over $[0,1]$, e.g., find the Q_i such that

$$\varepsilon = \int_0^1 ||B_n P - B_m Q||^2 dt$$

   is minimized, where B_k is a row matrix of the degree k Bernstein polynomials, P is a column matrix of the P_i and Q is a column matrix of one unknown Q_i. Differentiating with respect to the unknown Q_i gives us

$$\frac{\partial \varepsilon}{\partial Q_i} = \int_0^1 2B_i^m (B_n P - B_m Q).$$

   Differentiating with respect to all of the Q_i gives us the following linear system,

$$G_{m,n} P - G_{m,m} Q,$$

where the elements of $G_{m,n}$ are

$$g_{i,j} = \int_0^1 B_i^m(t) B_j^n(t) dt = \frac{\binom{m}{i}\binom{n}{j}}{(m+n+1)\binom{m+n}{i+i}}.$$

Setting this equal to 0 gives us

$$G_{m,n}P - G_{m,m}Q = 0,$$

and we can now solve for the unknown Q:

$$Q = G_{m,m}^{-1} G_{m,n} P.$$

$G_{m,m}$ is real, symmetric, and positive definite, so it is always invertible.

2. In general, while we want an approximation to P, we usually want this approximation to agree with P at the end points. I.e., we will additionally require $P^{(i)}(0) = Q^{(i)}(0)$ and $P^{(i)}(1) = Q^{(i)}(1)$ for at least $i = 0$ and possibly for $i = 1$ and higher. These end conditions fix the end control points of Q (essentially via the degree raising conditions). Separating Q's control points into the corner control points ($Q^c = [Q_0, ..., Q_i, Q_{m-i}, ..., Q_m]^t$) and "free" control points ($Q^f = [Q_{i+1}, ..., Q_{m-i-1}]^t$), and likewise decomposing B_m into B_m^c and B_m^f, the error term becomes

$$\varepsilon = \int_0^1 ||B_n P - B_m^c Q^c - B_m^f Q^f||^2 dt.$$

Differentiating with respect to the Q^f and equating to zero gives

$$G_{m,n}^p P - G_{m,m}^c Q^c - G_{m,m}^f Q^f = 0,$$

where

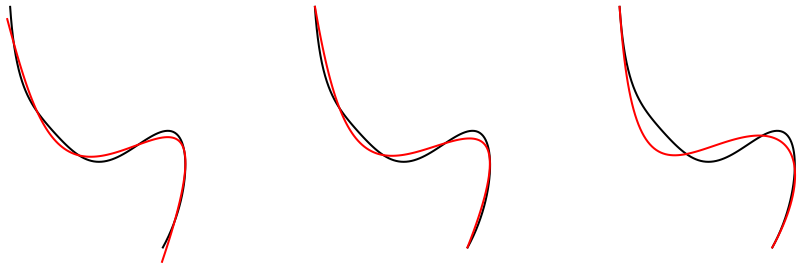
$$\begin{aligned} G_{m,n}^p &:= G_{m,n}(i+1, \ldots, m-i-1; 0, 1, \ldots, n), \\ G_{m,m}^c &:= G_{m,m}(i+1, \ldots, m-i-1; 0, 1, ..., i, m-i, ...m-1, m), \\ G_{m,m}^f &:= G_{m,m}(i+1, \ldots, m-i-1; i+1, \ldots, m-3-1), \end{aligned}$$

and $G_{m,n}(\ldots;\ldots)$ is the sub-matrix of $G_{m,n}$ formed by the indicated rows and columns.

Again, we can solve for the unknown control points as

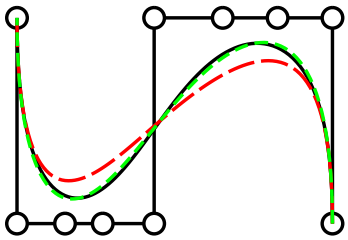
$$Q^f = (G_{m,m}^f)^{-1}(G_{m,n}^p P - G_{m,m}^c Q^c). \tag{4.2}$$

Below are some examples of degree reducing a degree 10 curve:



degree 4 C^0, degree 4 C^1, degree 4

3. This solution is essentially a "parametric continuity" solution. We could also ask for a geometric continuity solution; i.e., a solution where the position and tangents must be equal, but without requiring that the end first (or higher order) derivatives are equal. This gives us extra parameters to use to improve the approximation as seen in this example



where the black curve is the original curve, the red one (long dashes) meets the black curve with C^2 continuity at the end points, and the green curve (short dashes) meets the black curve with G^2 continuity at the end points.

However, using geometric continuity unfortunately can lead to a non-linear solution, although there are various ways to avoid the non-linearity.

4. Although the matrix $G_{m,m}$ is invertible, potentially the inversion can become numerically unstable. In practice, this occurs around $m = 24$ when using simple inversion techniques. However, using Gaussian elimination with pivoting results in stable solutions for values of m of at least 50.

4.5.1 Exercises

- Implement degree reduction with C^0 and C^1 continuity at the end points (equation 4.2), and test it on a few examples.

4.5.2 References

- Y. Ahn, B.G. Lee, Y. Park, and J. Yoo (2004). Constrained polynomial degree reduction in the L^2-norm equals best weighted Euclidean approximation of Bézier coefficients. Computer Aided Geometric Design 21: 181-191.

- L. Lu and G. Wang (2006). Optimal multi-degree reduction of Bézier curves with G^2-continuity. Computer Aided Geometric Design 23: 673-683.

- Abedallah Rababah and Stephen Mann (2011). G^2-Multi Degree Reduction of Bézier Curves. To appear in Journal of Applied Mathematics and Computation. http://dx.doi.org/10.1016/j

4.6 Interpolatory Curves

1. Suppose we want a curve to interpolate the control points. First, we've already seen basis functions for such a curve, at least as a single polynomial segment: The Lagrange basis functions. However, a second way to view the question is "is there an algorithm similar to the repeated linear combination algorithms for interpolatory curves?" The answer is "yes".

2. Suppose we have two points, P_0 and P_1 that we wish to interpolate at domain values t_0 and t_1. It's clear that

$$C(t) = \frac{t_1 - t}{t_1 - t_0} P_0 + \frac{t - t_0}{t_1 - t_0} P_1$$

   will do the trick.

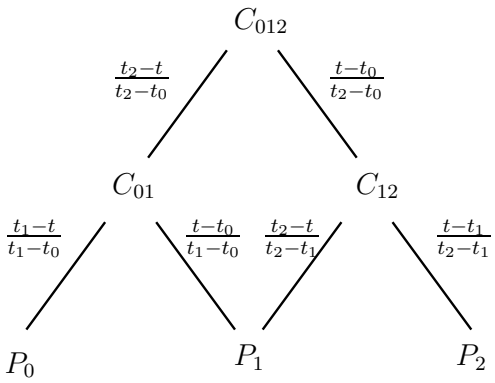
3. Suppose now that we have three points, P_0, P_1, and P_2, that we wish to interpolate at t_0, t_1, and t_2. We could form two linear curves to interpolate consecutive pairs of these points:

$$\begin{aligned} C_{01}(t) &= \frac{t_1 - t}{t_1 - t_0} P_0 + \frac{t - t_0}{t_1 - t_0} P_1 \\ C_{12}(t) &= \frac{t_2 - t}{t_2 - t_1} P_1 + \frac{t - t_1}{t_2 - t_1} P_2 \end{aligned}$$

   Now consider what happens if we linearly blend these two curves over $[t_0, t_2]$:

$$C_{012}(t) = \frac{t_2 - t}{t_2 - t_0} C_{01} + \frac{t - t_0}{t_2 - t_0} C_{12}.$$

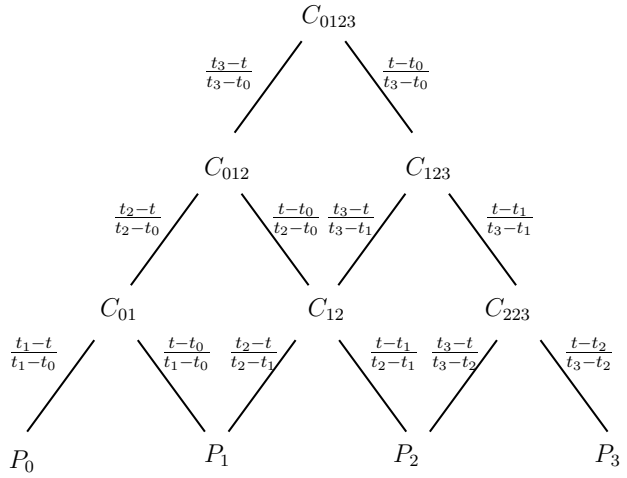
   Evaluating, we find $C_{012}(t)$ has the properties we want, i.e., that $C_{012}(t_i) = P_i$. And we can draw a pyramid diagram for this:



This is *Neville's algorithm.*

4. Note that we have done two things different than with de Casteljau's algorithm: we have evaluated over different intervals for each linear combination, and we have extrapolated. While the de Boor algorithm uses different linear combinations at different locations, it doesn't extrapolate.

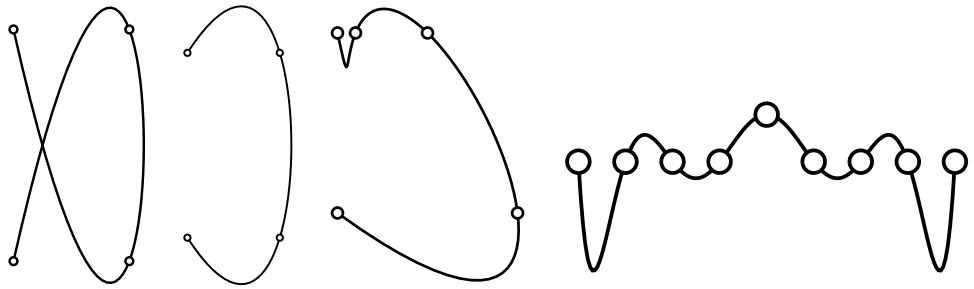
5. We can extend this algorithm to interpolate n points. For example, to interpolate four points, we obtain the following data flow diagram:



6. We note the following about these curves:

   - As already mentioned, the basis functions are the Lagrange basis functions. If we trace the data flow diagram backwards, they look different from (but similar to) the Lagrange basis functions. In fact, you can prove that they are the same. Further, this gives us that the degree n Lagrange basis polynomials sum to 1.
   - We can't assign blossom labels that make sense of the evaluation diagram (well, I can't — if you can, please show me!), nor is it easy to draw a de Casteljau type evaluation diagram.

- At high degrees, interpolatory polynomial curves exhibit poor behaviour, and are generally not used. Piecewise polynomial interpolatory curves (such as cubic Hermite splines) get used instead.



All of the above curves have knots at 0,1,2,...

4.7 Conic Sections

1. We can model a large variety of curves with polynomial Bézier curves and B-spline curves. However, we are restricted to parametric polynomial curves. In particular, if we consider degree two parametric polynomials, we only have one type of curve: a parabola.

   Traditionally, however, designers have used conic sections for curve design. While a parabola is one type of conic section, there are two other type: ellipses (including circles) and hyperbolas.

   While we can't represent conics as parametric polynomial functions, we can represent them as parametric rational polynomial functions (a rational polynomial is the ratio of two polynomials).

   I will only give a brief overview of rational curves. For more information, see Farin's book. There is also a multiprojective blossom of rational polynomials that I will not be discussing. See Ramshaw's Tech Report for more details.

2. A degree n rational Bézier curve is defined as

$$B(u) = \frac{\sum_i^n w_i P_i B_i^n(u)}{\sum_i^n w_i B_i^n(u)}.$$

   Here the w_is are real numbers while the P_is are points. The B_i^ns are the Bernstein polynomials. Note that this is an affine combination. Normally we think of evaluating this curve over the interval $[0, 1]$.

   One way of thinking of rational Bézier curves is as a polynomial curve in a higher dimensional space that we evaluate and then project down into a two dimensional space:

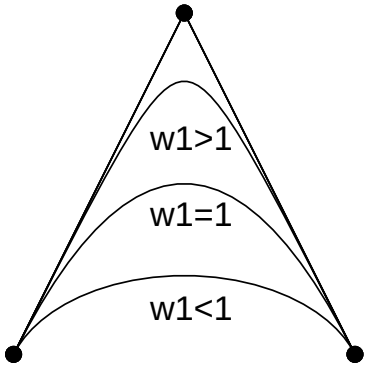
$$\bar{B}(u) = \Big(\sum_i^n w_i P_i B_i^n(u), \sum_i^n w_i B_i^n(u) \Big).$$

This is also the easiest way to evaluate a rational Bézier curve, although it is not the most numerically stable way. See Farin for a de Casteljau type of evaluation that is more stable.

Note that if some of the w_is are less than zero that we may get a divide by zero. Also, if all the w_i are 1, then our curve is a polynomial curve.

3. Let's consider the case when $n = 2$. These curves are the conic sections: parabolas, ellipses, and hyperbolas. In fact, if all the w_i are 1, then the curve is a parametric polynomial curve, which is a parabola.

   Further, it turns out that we can express our degree two rationals in a standard form where $w_0 = w_2 = 1$. We can now consider what happens as we vary w_1. For now, we'll assume that $w_1 > 0$. You can prove that if $w_1 < 1$ then the curve is a ellipse. If $w_1 = 1$ then the curve is a parabola. And if $w_1 > 1$ then the curve is a hyperbola.



4. There are two things to note in the above figure: first, we have only traced part of the conic. And second, we have traced the portion corresponding to the domain $[0, 1]$. If we trace over the domain $[-\infty, \infty]$, then we will trace the entire conic. But again, there are two things to note: first, our parameterization of the portion of the ellipse outside of $[0, 1]$ will be very poor, since an infinite portion of the domain will map to a finite portion of the curve. And second, for the parabola and hyperbola, we will have points at infinity.

5. It turns out there is another way to trace the portion of the conics outside of the interval $[0, 1]$ (actually, there are at least three other ways). Each rational quadratic with weights $w_0 = 1,\ w_1 \neq 0,\ w_2 = 1$ has a complement with weights $w_0' = 1,\ w_1' = -w_1,\ w_2' = 1$. If we trace the complement over the domain $[0, 1]$ then we get the remaining portion of the conic. Thus, if we trace both (each over $[0, 1]$) we get the entire conic section.

6. A third way to get the entire conic section (at least for ellipses) is to make it a piecewise rational curve, representing it as multiple rational curves. For the ellipse, we will need

at least three pieces (assuming we don't allow points at infinity). For the other two types of conics, we would need an infinite number of pieces (again, unless we allow points at infinity).

7. A fourth way to get the entire conic is to use the real projective one space as our domain. See the paper by DeRose in NURBS for Curve and Surface Design, Hans Hagen editor, SIAM, 1991.

8. Just as we made a rational form of Bézier curves, we can do the same for B-splines. If we allow a non-uniform knot vector, then we have a NURBs (Non-Uniform Rational B-spline) curve:

$$B(t) = \frac{\sum_{i=0}^{n} w_i P_i N_i^n(t)}{\sum_{i=0}^{n} w_i N_i^n(t)}.$$

4.7.1 Circles

1. For circles, note that we can construct a rational quadratic segment to represent a segment of the circle where the first and last control points are points on the circle. It turns out that the first and last segment of a rational Bézier are tangent to the curve at the start and end point of the curve (just as they are for polynomial Bézier curves), and because of symmetry of the circle, the three control points will form an isosceles triangle.

2. Setting the initial and final weights of the control points to 1, we still need to determine the weight for the interior control point. Farin and others have shown that this weight should be $\cos(\alpha)$, where α is the angle of the triangle of control points rooted at either the first and last control point.

4.7.2 Exercises

1. Suppose we set one of the weights in a rational Bézier curve $F(t)$ to zero. If we set one of the interior weights to zero, then the corresponding control point has no contribution in the curve. But if we set either the first or last weight to zero, we get a zero divide by zero singularity when $t = 0$ or $t = 1$ (assuming F is parametrized over the interval $[0, 1]$).

   We can remove this singularity by taking the limit as $t \to 0$. Derive the exact location of $F(0)$ when $w_0 = 0$ by removing this singularity via L'Hôpital's rule (you should submit your derivation). Verify your result by adding this special case to your rational Bézier code and testing it on the example in

   `http://www.student.math.uwaterloo.ca/~cs779/Data/Rational/test4`

4.7.3 Implementations

1. Implement an evaluator for rational Bézier curves. Your program should write PostScript output, and plot both the control polygon and the curve (you don't have to show the weights). Run your program on the test data {test0,test1,test2,test3} found in the course web page

   `http://www.student.math.uwaterloo.ca/~cs779/Data/Rational/`

   The format of each file is

   `d =` *degree*
   $x_0\ y_0\ w_0$
   $\vdots$
   $x_d\ y_d\ w_d$

   Note that there can be more than one rational Bézier curve specified per file. All the curves within one file should be drawn on a single page/diagram.

   Assume all curves are parametrized over $[0, 1]$. Note that to plot `test2` your program will have to be a little bit smarter: technically, there are two divide by zeros that cause the curve to exit at infinity in one direction and return from infinity in another direction.

Chapter 5

Geometric Continuity

5.1 Geometric Continuity

1. Let's return to curves momentarily. Suppose we have two curves F and G parameterized over $[0,1]$ and $[1,2]$ respectively, and further suppose that F and G meet C^1 at 1.

   Let $H(u) = G((u-1)/2+1)$. Then F and H meet C^0 at 1 (as $H(1) = G((1-1)/2+1) = G(1) = F(1)$, but fail to meet C^1 at 1:

   $$\begin{aligned} H'(u) &= \frac{dG((u-1)/2+1)}{du} \\ &= \frac{1}{2}G'((u-1)/2+1), \end{aligned}$$

   and thus $H'(1) = G'(1)/2 \neq F'(1)$.

   But geometrically our curves haven't changed (assuming we adjust Hs domain to be $[1,3]$). If we don't care about the parametrization of the curves, and only care about the geometry, then we would like to say these curves meet smoothly. This motivates the definition of geometric continuity.

2. Definition (informal): Two curves F and G meet with G^k continuity at t_0 if there exists a non-decreasing function l such that $F^{(i)}(t_0) = G^{(i)}(l(t_0))$ for all i from 0 to k.

   Technically we need more conditions on our functions, such as requirements that the derivatives of F and G don't vanish. Further, we should probably make a statement about the domains of both functions, and insist that $l(t_0) = t_0$.

   It should be clear that all curves meeting G^0 also meet C^0. It is reasonably easy to see that two curves meet with G^1 continuity if they have the same tangent line at the point of contact.

   To find a geometric meaning for G^2 continuity, we need to know about curvature. If we look in a local neighborhood of a point on a curve, we can find the best approximating

circle to the curve. The reciprocal of the radius of this best approximating circle is known as the curvature of the curve at the point on the curve. If we give a sign to the curvature (based one whether the best approximating circle is on the "right" side or "left" side of the curve), then we have what's known as signed curvature.

If two curves meet with G^2 continuity at a point, then they have the same tangent lines and signed curvatures at that point.

3. For surfaces, there is a formal definition of geometric continuity (which we may give later). However, as we will restrict ourselves to surfaces meeting G^1, we won't give a formal definition now.

   Informally, two surface patches are said to meet G^1 along a common boundary if they meet C^0 along that boundary and if they have the same tangent planes at each point along this boundary.

   Note that there are two benefits to using G^1 continuity rather than C^1 continuity. The first benefit is the one that motivated the definition: the notion of continuity is invariant under "nice" transformations of the domain.

   The second benefit is an increase in the kinds of surfaces we can construct. In particular, with C^1 continuous surfaces, we are restricted to constructing surfaces that have planar topological type. In particular, we can not construct a piecewise C^1 surface that is topologically a sphere without introducing singularities. In particular, we can not construct a consistent first derivative vector field ("you can't comb the hair on a billiard ball"). Note that we can do a little better than just surfaces of planar topological type: we can also construct cylinders and tori. The cylinder has an infinite strip as its domain while the torus has a square domain. Both domains can be used to tile the plane, and thus, we can identify domain edges and construct consistent first derivative fields across the boundary.

   With tangent plane continuity, we can consider an arbitrary polyhedron as our domain. We no longer need neighboring domain faces to be coplanar as we no longer need to construct a domain vector in both domains simultaneously.

4. However, we do pay a price for tangent plane continuity. Consider our condition: that the tangent planes of both patches agree along the boundary. Equivalently, both patches have the same surface normals along the boundary. The two normals will be equal if the cross product of the normal to one patch is perpendicular to a crossboundary tangent to the other patch. I.e., if $H(t)$ is the common boundary curve, $N_F(t)$ is the normal to F along the boundary, and $G_{\vec{v}}$ is the crossboundary derivative G in direction $\vec{v}$, then for all t we need

$$N_F(t) \cdot G_{\vec{v}}(t) = 0.$$

   For both triangular and tensor product patches, this becomes a polynomial equation as N_F can be written as the crossproduct of H and a crossboundary tangent of F; we

can ignore the normalizing factor since this equation must equal zero:

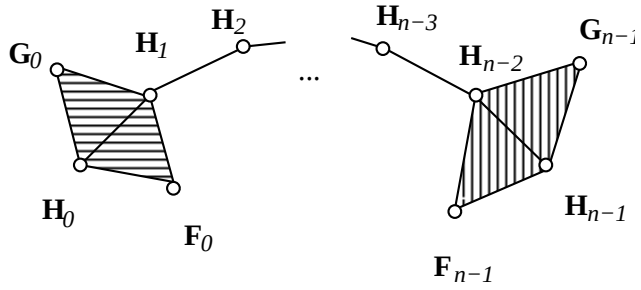
$$(F_{\vec{w}}(t) \times H'(t)) \cdot G_{\vec{v}}(t) = 0.$$

For triangular patches, this equation can be written as $3(n-1)$ polynomial equations in the control points of F and G.

If all of our control points are unknown, this yields a set of cubic equations to solve. More commonly, we will know the boundary control points, leaving a set of equations that are quadratic in the unknowns.

Geometric interpretations are even more difficult except at the ends of the boundary curves, where the condition on the end panels is that they are coplanar. With some work, geometric conditions can be determined for quadratic triangular patches, although even this construction is fairly complicated. No further geometric conditions are known.

5. However, Farin has shown that if we have two degree n patches meeting along a degree $n-1$ boundary then the conditions simplify and become a linear set of equations. In particular, in the drawing below let H be the control points for the degree $n-1$ boundary, and let F and G be the control points for the degree n Bézier patches.



Write G_0 and G_{n-1} as affine combinations of the other points in the coplanar panels:

$$\begin{aligned} G_0 &= \alpha_1 H_0 + \alpha_2 H_1 + \alpha F_0, \\ G_{n-1} &= \alpha_3 H_{n-2} + \alpha_4 H_{n-1} + \alpha F_{n-1}, \end{aligned}$$

where $\alpha_1 + \alpha_2 + \alpha = 1 = \alpha_3 + \alpha_4 + \alpha$. Then the two patches meet G^1 if the following hold for $i = 0, \ldots, n-1$:

$$G_i = \frac{n-1-i}{n-1}(\alpha_1 H_i + \alpha_2 H_{i+1} + \alpha F_i) + \frac{i}{n-1}(\alpha_3 H_{i-1} + \alpha_4 H_i + \alpha F_i).$$

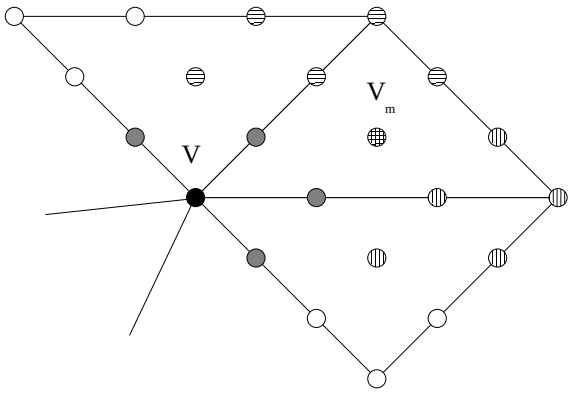
The point here is that we have moved from a non-linear situation to a linear situation. Further, similar conditions exist for rectangular patches.

Note that there is a geometric interpretation of the locations of G_0 and G_{n-1}: we need the following triangle ratios to hold:

$$\frac{\text{Area}(G_0 H_0 H_1)}{\text{Area}(F_0 H_0 H_1)} = \frac{\text{Area}(G_{n-1} H_{n-2} H_{n-1})}{\text{Area}(F_{n-1} H_{n-2} H_{n-1})}$$

6. The above can be used to construct two patches that meet with G^1 continuity. Suppose now that we want to fill a triangular curve network with triangular patches. Alternatively, suppose we are given a polyhedron with triangular faces and we wish to construct a G^1 polynomials surface that interpolates the vertices of the polyhedron. One way to proceed is to first build a curve network along the edges of the faces and then fill in the interiors.

   We immediately run into the vertex consistency problem. Consider a vertex of the polyhedron and the ring of curves and faces surrounding this vertex. Suppose we try to construct neighboring patches to meet G^1.



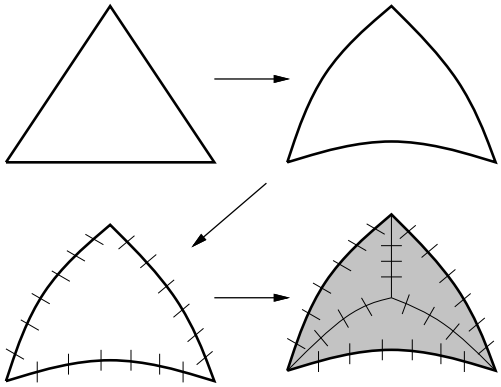
   Consider the ring of faces around the vertex V in the figure above. If we first construct the boundaries to meet G^1 at V and then try to construct three patches to meet G^1, then we might construct two positions for the vertex V_m. This, however, is not allowed for polynomials patches.

   What the above implies is that we need to construct all boundaries surrounding V simultaneously. However, the same argument can be applied at the other end of each of the boundary curves. Thus, to solve both the vertex consistency problem and the G^1 problem, we would have to solve them for all vertices in our mesh simultaneously. This is known as a global solution. Some of the problems with global solutions include

   - They're slow
   - If we move a single vertex, the entire surface changes.

   The above two problems would be bad enough. However, it can be shown that if the number of faces surrounding the vertex is even and more than four, then in general there will not be a solution. Thus, in general the above problem can *not* be solved.

7. A work around exists for this problem. What we'll do is rather than fit one patch per face, we'll fit three. These methods are known as split domain schemes. The construction is illustrated in the following diagram:



We begin with a triangle of data. As a first step, we'll construct boundary curves between the data points. Then, we'll construct a G^1 along each patch independently. Finally, we'll fill the interior with three patches, once adjacent to each of the boundary curves we have created. Note that we don't have the inconsistent mixed partial derivative problem at the corners of the triangle. We will have to ensure consistent mixed partial derivatives when we construct the remainder of the interior, but that's a solvable problem.

8. The first step in the problem is to construct boundary curves. However, we'll look a head a little bit to see what kinds of boundaries we can construct. In particular, we need to know what is the lowest degree patch we can use to get a G^1 join across the external boundaries.

   The above picture is a little bit deceiving: before we construct the G^1 joins along the triangle boundaries, we will be constructing the first derivatives along the interior boundaries at the corner vertices.

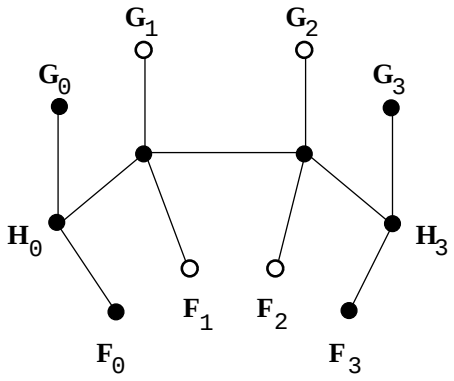
   In this new situation, it can be shown that cubic patches work in most but not all situations. In particular, if there are certain symmetries in our data, then a cubic construction fails. Thus, to handle all settings with one patch type, we have to use quartic patches.

   We'll now look at one particular crossboundary derivative construction.

9. The conditions given above can be used to a) test if two patches meet G^1, or b) given F we can construct G to meet G^1 along one boundary. But suppose we want to construct two patches to meet G^1. In a common situation we already have the boundary control points for both patches and we merely want to fill in the interiors to meet G^1. We could build one patch arbitrarily and then build the second using the above conditions. However, our construction will not be symmetrical.

   Chiyokura and Kimura describe a constructive approach to creating a G^1 join. Their scheme was intended for rectilinear patches, but also works for triangular patches. In fact, you can use their scheme to connect triangular patches to rectilinear patches. Similar schemes were created by Herron and Jensen.

Suppose we have control points for two patch F and G with common boundary H as shown below:



Here we can either think of all the control points as belonging to bicubic tensor product patches, or we can think of them as belonging to quartic triangular patches with cubic boundaries. We will assume the boundaries have already been constructed and thus, we are merely looking for G_1, G_2, F_1, and F_2.

The approach taken is to first construct a crossboundary vector field, and then build both F and G to agree with crossboundary vector field along some parametric direction. Thus, F and G will have the same boundary normals and therefore will meet G^1.

The cross-boundary tangent vector field is defined by linearly blending two vectors, $\hat{C}_0$ and $\hat{C}_1$, one in each of the tangent planes at the endpoints. Chiyokura and Kimura choose these vectors to be unit vectors perpendicular to the tangents at the boundary curve's endpoints, i.e., $< \hat{C}_0, H_1 - H_0 > = < \hat{C}_1, H_3 - H_2 > = 0$. For patch F, the cross-boundary field is given by

$$\vec{C}(t) = (1-t)\hat{C}_0 + t\hat{C}_1.$$

The two perpendiculars $\hat{C}_0$ and $\hat{C}_1$ are unique, up to sign. The signs are chosen so that k_0 and k_1 below are positive. $\vec{C}(t)$ together with $H'(t)$ completely specifies the tangent plane field along the boundary.

For F to agree with the tangent plane field given by $H'(t)$ and $\vec{C}(t)$, there must exist functions $k(t)$ and $h(t)$ such that

$$D_{\vec{r}(t)}F(0,t,1-t) = k(t)\cdot\vec{C}(t) + h(t)\cdot H'(t), \tag{5.1}$$

where $\vec{r}(t)$ is the radial direction in the domain of F (i.e., if the domain of F is $\triangle pqs$ where $F(q) = H_0$ and $F(s) = H_3$, then $\vec{r}(t) = ((1-t)q + ts) - p)$.

Let $\vec{F}_i = F_i - H_i$ and $\vec{H}_i = H_{i+1} - H_i$. The values of $k(t)$ and $h(t)$ are determined at the endpoints by evaluating Equation 5.1 at $t = 0$ and $t = 1$:

$$\vec{F}_0 = k_0\cdot\hat{C}_0 + h_0\cdot\vec{H}_0, \tag{5.2}$$

$$\vec{F}_3 = k_1\cdot\hat{C}_1 + h_1\cdot\vec{H}_2, \tag{5.3}$$

where $k_0 = k(0)$, $k_1 = k(1)$, $h_0 = h(0)$, and $h_1 = h(1)$. For h and k to interpolate these endpoint conditions, they both must be at least linear functions. If we restrict them to be no more than linear, then each is uniquely determined:

$$\begin{aligned} k(t) &= k_0 \cdot (1-t) + k_1 \cdot t, \\ h(t) &= h_0 \cdot (1-t) + h_1 \cdot t. \end{aligned}$$

Rewriting Equation 5.1 in the cubic Bernstein basis, the coefficients of $B_1^3(t)$ and $B_2^3(t)$ determine the desired interior control points:

$$F_1 = \frac{1}{3}\{(k_0 + k_1)\hat{C}_0 + k_0\hat{C}_1 + 2h_0\vec{H}_1 + h_1\vec{H}_0\} + H_1, \tag{5.4}$$

$$F_2 = \frac{1}{3}\{k_1\hat{C}_0 + (k_0 + k_1)\hat{C}_1 + h_0\vec{H}_2 + 2h_1\vec{H}_1\} + H_2. \tag{5.5}$$

There is still some freedom left in Equation 5.1. If $h(t)$ is a linear function, then the product $k(t) \cdot \vec{C}(t)$ must be a polynomial of no higher than third degree. In the above formulation, this product is only a quadratic polynomial. Either $k(t)$ or $\vec{C}(t)$ can be increased from a linear function to a quadratic function. Increasing $k(t)$ to a quadratic function gives a scalar degree of freedom, while increasing the degree of $\vec{C}(t)$ yields a vector degree of freedom. Jensen used the former generalization. He used the same linear blend of unit vectors for $\vec{C}(t)$, but used the following quadratic scale function:

$$k^*(t) = k_0 \cdot u_0(t) + C \cdot \frac{(k_0 + k_1)}{2} u_1(t) + k_1 \cdot u_2(t),$$

where

$$u_0(t) = 2t^2 - 3t + 1, \quad u_1(t) = 4t - 4t^2, \quad u_2(t) = 2t^2 - t,$$

and C is a scalar shape parameter. For $C = 1$, $k^*(t) = k(t)$.

A second degree of freedom in Equation 5.1 is in the choice of $\hat{C}_0$ and $\hat{C}_1$. These two vectors may be chosen in any fashion that uses information available to both patches, where the construction from both sides gives the same vectors with opposite sign. For example, in a later paper, Chiyokura defines $\hat{C}_0$ and $\hat{C}_1$ as

$$\hat{C}_0 = \frac{G_0 - F_0}{|G_0 - F_0|},$$

$$\hat{C}_1 = \frac{G_3 - F_3}{|G_3 - F_3|}.$$

Note that this definition of $\hat{C}_0$ and $\hat{C}_1$ is affine invariant and requires knowledge about both patches neighboring the boundary, whereas the earlier definition is not affine invariant but uses only information about the boundary curve.

Chapter 6

Bivariate Data Fitting and Modeling

6.1 Interpolatory Surfaces

We will briefly look at polynomial surfaces that interpolate grids (triangular or rectilinear) of data points. Such schemes are unsuitable for most purposes as (a) they require high degree surfaces; (b) they have the same shape problems that interpolatory polynomial curves have.

6.1.1 Triangular Lagrange Patches

1. Suppose we are given a triangular grid of 2D domain points D_{ijk} and a triangular grid of points P_{ijk}, where
$$D_{ijk} = \frac{i}{n} D_{n00} + \frac{j}{n} D_{0n0} + \frac{k}{n} D_{00n},$$
with $i + j + k = n$. Our goal is to construct a surface S such that $S(D_{ijk}) = P_{ijk}$.

2. If $n = 1$ then the question is easily solved: Let (u_0, u_1, u_2) be the barycentric relative to $\triangle D_{100} D_{010} D_{001}$. Then
$$S(u) = u_0 P_{100} + u_1 P_{010} + u_2 P_{001}$$
has the desired interpolatory properties.

3. Suppose now that $n = 2$. We can easily build three surfaces S_{100}, S_{010}, and S_{001} that each interpolate three of the data points, where in particular each S will interpolate the points $P_{i+1\ jk}$, $P_{ij+1\ k}$, and P_{ijk+1} at the corresponding domain point. Eg,
$$S_{100}(u) = u_0 P_{200} + u_1 P_{110} + u_2 P_{101},$$
where (u_0, u_1, u_2) are the barycentric coordinates of u relative to $\triangle D_{200} D_{110} D_{101}$

4. Consider the following surface:
$$S(u) = u_0 S_{100}(u) + u_1 S_{010}(u) + u_2 S_{001}(u),$$

where (u_0, u_1, u_2) are the barycentric coordinates of u relative to $\triangle D_{200}D_{020}D_{002}$. Note that the subsurfaces $S_{100}, S_{010}, S_{001}$ blend three points using barycentric coordinates relative to different domain triangles.

If we evaluate S at one of the corners of $\triangle D_{200}D_{020}D_{002}$ then we obtain the corresponding P value, since (for example) D_{200} has barycentric coordinates $(1, 0, 0)$ relative to $\triangle D_{200}D_{020}D_{002}$, and thus $S(D_{200}) = S_{100}(D_{200}) = P_{200}$.

If we evaluate S at one of the other three domain vertices (for example, D_{110}), we get

$$\begin{aligned} S(D_{110}) &= \frac{1}{2}S_{100}(D_{110}) + \frac{1}{2}S_{010}(D_{110}) + 0S_{001}(D_{110}) \\ &= P_{110}. \end{aligned}$$

5. This brings us to the idea for the arbitrary degree triangular Lagrange surface: Assume we have three lower degree Lagrange surfaces defined over overlapping domains. When we blend these surfaces, if we evaluate at a corner of the enlarged domain such as D_{n00}, then only one barycentric coordinate is non-zero, and we obtain the corresponding range point, while if we evaluate at a common domain point (such as D_{ijk}, where $i, j, k \geq 0$), then all three subsurfaces have same value P_{ijk}.

   More formally, assume we have three degree n triangular Lagrange surfaces, $S_{n00}, S_{0n0}, S_{00n}$ where

$$\begin{aligned} S_{n00}(D_{ijk}) &= P_{ijk} \qquad i \neq 0 \\ S_{0n0}(D_{ijk}) &= P_{ijk} \qquad j \neq 0 \\ S_{00n}(D_{ijk}) &= P_{ijk} \qquad k \neq 0. \end{aligned}$$

   Build the surface S as

$$S(u) = u_0 S_{n00}(u) + u_1 S_{0n0}(u) + u_2 S_{00n}(u),$$

   where (u_0, u_1, u_2) are the barycentric coordinates of u relative to $\triangle D_{n00}D_{0n0}D_{00n}$.

   Then $S(D_{ijk}) = P_{ijk}$.

6. Note that there is a pyramidal evaluation algorithm for these surfaces.

6.1.2 Rectilinear Lagrange Patches

1. We can construct an interpolatory rectilinear patch by using univariate methods. In particular, suppose we have two knot vectors, $s_0 < s_1 < \ldots < s_m$ and $t_0 < t_1 < \ldots < t_n$, and a set of points $P_{i,j}$ for $0 \leq i \leq m$ and $0 \leq j \leq n$, and we want a surface $S(s, t)$ such that

$$S(s_i, t_j) = P_{i,j}.$$

Then the following surface clearly has these properties:

$$S(s,t) = \sum_i \sum_j L_i^m(s) L_j^n(t) P_{i,j},$$

where $L_i^m(s)$ are the Lagrange basis functions formed from the s-knots, and $L_j^n(t)$ are the Lagrange basis functions formed from the t-knots. This is also known as a *tensor product Lagrange surface*.

2. Like tensor product Bézier patches, we can regroup the above expression and evaluate the tensor product Lagrange patch via repeated linear interpolation:

$$S(s,t) = \sum_i L_i^m(s) \Big(L_j^n(t) P_{i,j} \Big).$$

3. Note that there is a pyramidal evaluation algorithm for these surfaces, similar to repeated bilinear interpolation for tensor product Bézier surfaces.

6.2 Functional Triangular Interpolation Schemes

In this section, we will begin to look at methods for fitting surfaces to a set of triangulated points. Our main interest is on parametric data, but we will start here by looking at two functional schemes as they are the basis for some of the parametric schemes.

Recall that in the functional case, for C^1 continuity you merely need adjacent panels of adjacent patches to be co-planar. The two schemes in this section rely heavily on this fact.

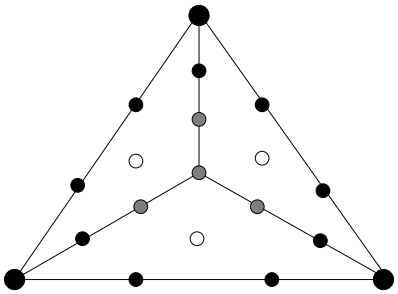
6.2.1 Clough-Tocher

1. The functional variation of our problem is given a set of z values over the plane whose corresponding (x, y) values have been triangulated, find a piecewise polynomial, C^1 surface that interpolates this data. Usually, we also assume that we have normals at the data points and that we're trying to interpolate both position and normal.

2. While it is easy to achieve C^1 continuity between two functional Bézier patches, it is more difficult to achieve the desired continuity in a patch network. This difficulty is a result of the continuity conditions across the boundaries imposing a cycle of constraints around each vertex.

3. Clough-Tocher bypassed this problem by splitting each triangle of the domain into three triangles. This effectively splits the cycle of contraints.

   Clough-Tocher then fit a cubic Bézier patch to each of these subtriangles:



Their construction works as follows (remember, we only need to find the z value for each point):

(a) The large black points are the data points.

(b) The small black points are determined by the normals at the data points; these points much lie in the tangent plane at the nearest corner vertex (data point), and since we only need the z value, we can find their location by intersecting a line with a plane.

(c) The white points are set to achieve a C^1 join with the patch in the neighboring macro-triangle; there is a linear degree of freedom in this construction.

(d) The grey points are set so as to achieve C^1 continuity across the interior boundaries.

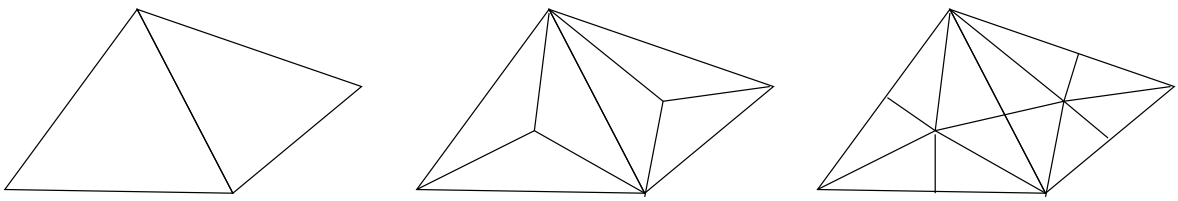
4. The only freedom in this construction is in the setting of the white points. Clough-Tocher used a setting that achieves quadratic precision. Later, Farin used a setting to minimize the C^2 discontinuity, which gives the resulting patch cubic precision. Later still, Foley and Opitz gave a cross boundary construction similar to Farin's that also gives cubic precision.

5. Franke gives a survey of this and other functional interpolation schemes.

6.2.2 Powell-Sabin

1. Another functional data fitting scheme performs a 6:1 split of the domain. The advantage of this scheme is that it fits quadratic patches to the data (position and normals) and is C^1 at the joins.

2. The first step is to subdivide the domain into six triangles. This is done as a two step process, illustrated on two data triangles by the following diagram:

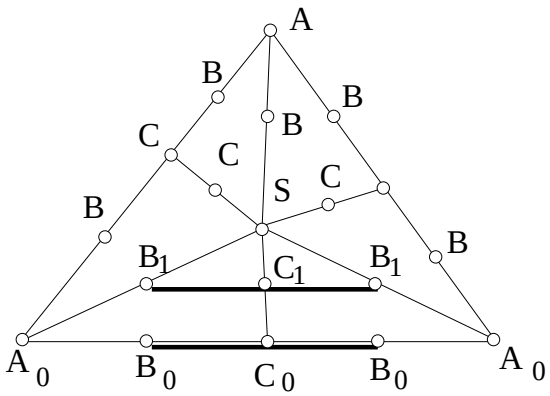


In the first step, we perform a 3:1 split on each domain triangle, splitting at some point on the interior of each triangle. The split point might be the centroid, but isn't required to be.

As a second step, we connect the split points of adjacent triangles, and use this to split each triangle from the previous step into two triangles. Note that this divides the edge between neighboring data triangles at a point *other than the midpoint*.

For edges that don't have neighboring triangles, we split them along at an arbitrary point, usually the midpoint.

3. Next, we need to set the z-values of the control points. We will use the following diagram for the six patches constructed for one data triangle:



The construction of z-values proceeds in the following steps:

(a) Set the A z-values to those of the data points.

(b) Set the B z-values so that the B points lie in the tangent plane at the corresponding data point.

(c) Set the C_i points to lie on the segment between the corresponding pairs of B_is. Note that this is not at the midpoint of segment. Rather, it is in the ratio that the C_i domain point (i.e., just the xy values) breaks the domain B_i segment.

(d) Set S to lie in the plane spanned by the Bs and Cs.

4. C^0 continuity across all boundaries is clear. C^1 continuity across internal boundaries is also clear: The As and neighboring Bs are coplanar; S and the Bs and Cs are coplanar, and $B_0C_0B_0C_1$ are coplanar.

5. What is less clear is why there is C^1 continuity across the external boundary: nothing in the construction appears to even address the issue. The trick that gives us C^1 continuity comes from the way we joined the split points when going from a 3:1 split to a 6:1 split. Proof of continuity across the macroboundaries is left as an exercise.

6.2.3 Exercises

1. Prove that the Powell-Sabin scheme constructs patches that meet with C^1 continuity across the macroboundaries.

2. Suppose we want to construct a function $f(x, y)$ with the following properties:

   - f is zero on the boundaries and exterior of a regular n-sided polygon centered at the origin in the x-y plane.
   - f is non-zero everywhere inside the polygon.
   - $f(0, 0) = 1$.
   - f is C^1 everywhere.

   Further suppose we wish to construct such a function using quadratic (degree 2) triangular Bézier patches. As a first step we need to triangulate the domain polygon. We will then place one Bézier patch over each triangle.

   (a) (5 pts) Prove that if we use the obvious triangulation of the domain (connecting the center to the corners) then we can not construct quadratic Bézier patches over these triangles that meet C^1. Note: you only need to prove this for a particular polygon (I suggest the square). The generalization to an arbitrary n-gon is immediate.

   (b) (15 pts) Derive a construction for such a function f using quadratic (degree 2) triangular Bézier patches. You may orient the polygon in the plane in any fashion that is convenient to you. You may use as many triangular patches as you need (although the problem can be solved with $3n$ patches). Note any shape parameters your construction has.

   (c) (5 pts) Use your program from the first problem to plot solutions for $n = 3, 4, 5, 6$.

   Note: This problem uses functional Bézier patches. Once you know the domain of such a patch, you know the x-y values of all the contorl points. All you need to find is the z-values of the control points.

6.3 Parameteric Triangular Interpolation Schemes

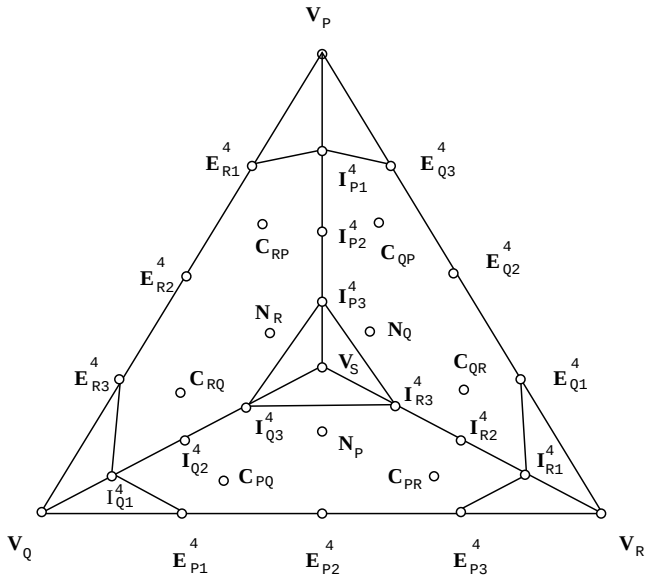
We will now look at parameteric data interpolation. The C^1 conditions for parameteric patches are more difficult than those for functional patches. But even those are inadequate, since we need to use G^1 continuity between patches in the parametric case. The schemes in this section rely heavily on the continuity construction of Farin and Chiyokura-Kima described earlier in these notes.

6.3.1 Shirman-Séquin

1. The description below is based on Shirman-Séquin's construction, with notes as to where degrees of freedom occur.

   To each triangle of data, we will fit three quartic patches. These patches will have cubic boundaries. Along the exterior boundaries, we will use the Chiyokura-Kimura G^1 construction. And along the interior boundaries, we will use Farin's constraints to achieve a G^1 join.

   We will label the control points as illustrated in the following diagram:



   The boundaries of all patches are cubic. Because we will want sometimes to refer to the cubic control points and at other times the quartic control points, we will super script them with the degree.

2. The construction proceeds as follows:

   (a) Corner points (V_Q etc) are required to be the corners of the triangle.

   (b) Along each exterior boundary, there are two control points that need to be set, the E^3s. Each control point has two scalar degrees of freedom. Empirical tests indicate that it is advantageous to restrict each boundary to a plane, reducing the number of degrees of freedom to one per control point (plus one for the plane).

   Note that we must construct the same boundary curve for both the patch and its neighbor on the other side of the triangle's edge.

   (c) Next, the I^3_{*1} control points are set. While each has two scalar degrees of freedom, for reasons of symmetry it is best to restrict them to the line through V_* and the midpoint of the nearest Es.

(d) Now we can use the Chiyokura-Kimura construction to build the C control points. Note that we have five scalar degrees of freedom for each pair of Cs. However, four of these degrees of freedom must be used in an identical fashion by the construction for the patch on the other side of the triangle's edge.

(e) The remaining control points are set to satisfy Farin's G^1 conditions along all the interior boundaries. First, we set V_S to be the centroid of the I^3_{*2}s. Note that this is the only symmetric choice for this control point. Also note that this choice of V_S and the above choice of I^3_{*1}s meets the first requirement of Farin's conditions (that of equal ratio panels at the ends).

We now use Farin's conditions to get six equations with six unknowns. For $ijk \in \{PQR, QRP, RPQ\}$, the following relationships are imposed:

$$E^4_{j3} = \alpha_{i2} I^3_{i1} + \alpha_{i1} V_i + \alpha_i E^4_{k1}, \tag{6.1}$$

where $\alpha_{i1} + \alpha_{i2} + \alpha_i = 1$. Similarly, at the other end of the internal boundaries, the following relationships hold:

$$I^4_{k3} = \alpha_{i3} I^3_{i2} + \alpha_{i4} V_S + \alpha_i I^4_{j3}, \tag{6.2}$$

where $\alpha_{i3} + \alpha_{i4} + \alpha_i = 1$. For reasons of symmetry, we set $\alpha_i = -1$ for all i.

This system of equations involving the interior control points that has the following solution:

$$I^3_{i2} = -\frac{\alpha_{i3}}{2\alpha_{i2}} V_i - \left(\frac{\alpha_{i1}}{\alpha_{i2}} + \frac{\alpha_{i4}}{2\alpha_{i2}}\right) I^3_{i1} + \frac{3}{2\alpha_{i2}} \left(C_{ji} + C_{ki}\right),$$

$$\begin{aligned} N_i \;=\; & -\frac{\alpha_{i3}}{3} I^3_{i1} + \frac{\alpha_{i3}}{3}(I^3_{j1} + I^3_{k1}) + \left(\frac{\alpha_{i2}}{18} - \frac{\alpha_{i1} + 2\alpha_{i4}}{6}\right) I^3_{i2} + \\ & \left(\frac{\alpha_{i2}}{18} + \frac{\alpha_{i1} + 2\alpha_{i4}}{6}\right) \left(I^3_{k2} + I^3_{j2}\right). \end{aligned}$$

Setting V_S to be the centroid of the I^3_{i2}s fixes the following αs:

$$\alpha_{i3} = -\frac{3}{4},\ \alpha_{i4} = \frac{11}{4}.$$

α_{i1} and α_{i2} are now related by $\alpha_{i1} + \alpha_{i2} = 2$. This leaves a scalar shape parameter to influence the shape of the patch interiors. Note: Setting I^3_{i1} sets α_{i1} and α_{i2}; alternatively, setting α_{i1} and α_{i2} sets I^3_{i1}. The point is, these three things are not independent.

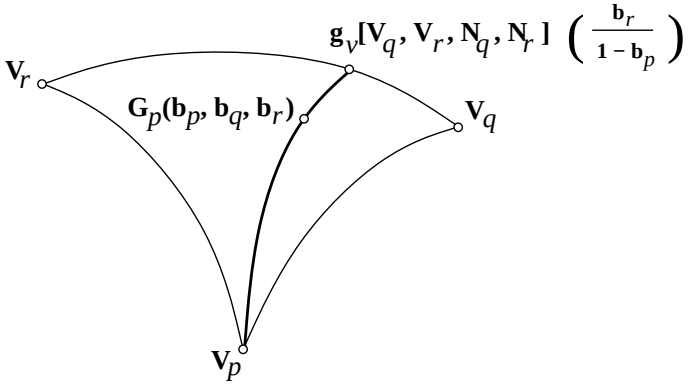
Note that even after making simplifications (using cubic boundaries, using Chiyokura-Kimura's G^1 conditions, using Farin's G^1 conditions), we still have a large number of degrees of freedom remaining. *These degrees of freedom have a large impact on the shape of the surface.* In the papers describing this construction, these degrees of freedom are set arbitrarily, resulting in surfaces with arbitrarily poor shape.

3. There are several similar schemes due to Jensen, Piper, and Peters. Hansford also has a rational polynomial scheme that operates in a similar manner.

6.3.2 Nielson

1. We now look at Nielson's method for filling triangular networks. This method does not create polynomial patches. In fact, it does not even give an explicit representation for the surface. Rather, it gives a construction for the patch.

   Nielson's scheme is a side-vertex method. The method proceeds by first constructing three boundary curves, one corresponding to each edge of the input triangle. Next, three patches are created, one for each boundary/opposite-vertex pair. The interior of each patch is constructed by passing curves from points along the boundary (or "side") to the opposite vertex. Hence the name "side-vertex", as shown in the figure below. The three patches are then blended together to form the final patch.



2. All curves are constructed from two points and associated normals. We assume the existence of a curve construction operator g_v that takes two vertices with normals and constructs a curve:

$$g_v[V_0, V_1, \hat{N}_0, \hat{N}_1](t),$$

   such that $g_v(0) = V_0$, $g_v(1) = V_1$, $< g_v'(0),\ \hat{N}_0 > = 0$, and $< g_v'(1),\ \hat{N}_1 > = 0$. We also assume the existence of a normal field constructor g_n that constructs a continuous normal field along the curve g_v, where g_n is required to interpolate $\hat{N}_0$ and $\hat{N}_1$ at the endpoints.

3. The construction proceeds by building three patches, G_i, $i \in \{p, q, r\}$ defined as:

$$\begin{aligned} G_i(b_p, b_q, b_r) &= g_v\left[V_i,\ g_v[V_j, V_k, \hat{N}_j, \hat{N}_k]\left(\frac{b_k}{1-b_i}\right),\right. \\ &\qquad \left.\hat{N}_i,\ g_n[V_j, V_k, \hat{N}_j, \hat{N}_k]\left(\frac{b_k}{1-b_i}\right)\right](1-b_i), \end{aligned}$$

   where b_p, b_q, b_r are the barycentric coordinates of the domain point. Nielson notes the following two properties of G_i:

   (a) G_i interpolates all three of the boundaries.

(b) G_i interpolates the tangent plane field of the boundary opposite vertex V_i.

4. The final surface is defined to be

$$G[V_p, V_q, V_r, \hat{N}_p, \hat{N}_q, \hat{N}_r] = \beta_p G_p + \beta_q G_q + \beta_r G_r,$$

where

$$\beta_i = \frac{b_j b_k}{b_p b_q + b_q b_r + b_r b_p}. \tag{6.3}$$

Nielson shows that if three surfaces having properties 1 and 2 are blended with these β_i, then the resulting surface will interpolate all the boundary curves and tangent fields. The theorem is true for a large class of g_v and g_n. The operator g_v that Nielson presents constructs tangent vectors from the two normals and interpolates the two points and these vectors with a cubic polynomial curve.

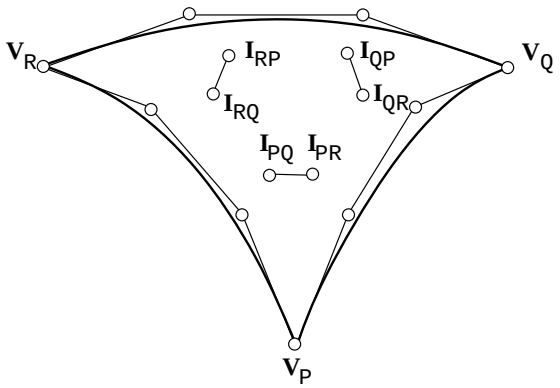
5. The construction of curves is discussed later in the notes. Here we will look at a normal constructor. If we first construct the curve g_v and want a normal field to be perpendicular to the surface, we need the normal field to be perpendicular to this curve.

As an initial guess, we can linearly interpolate the normals at the vertices. However, such a "normal" will not be perpendicular to the boundary curve. So starting with this initial normal guess, N_g, we compute the cross product of $g_v' \times N_g$, giving a vector perpendicular to both N_g and g_v'. We now compute the normal as

$$(g_v' \times N_g) \times g_v'.$$

6.3.3 Triangular Gregory Patches

1. Triangular Gregory patches are a variant of Gregory squares. Essentially, we construct a quartic patch with cubic boundaries, where the twist points of the patch are a blend of two twist points, one for each boundary:



2. The points I_{ij} and I_{ik} are blended to form the point I_i as follows:

$$I_i = \frac{b_k(1-b_j)I_{ij} + b_j(1-b_k)I_{ik}}{b_k(1-b_j) + b_j(1-b_k)}$$

where (b_i, b_j, b_k) are the barycentric coordinates of the domain point relative to the domain triangle. Note that if we are on the edge between i and j, then b_k is zero, so $I_i = I_{ik}$.

We will use Chikura-Kimura to construct our crossboundary points. To evaluate the patch, we compute I_p, I_q, and I_r and evaluate the patch as a polynomial Bézier patch.

It can be shown that this patch interpolates the crossboundary derivative behavior on all three boundaries.

3. The patch, however, is a rational polynomial patch. Further, it has a removable singularity at the corner points, since we get a 0/0 in the computation of one of the I_is.

6.3.4 Herron

1. Herron introduced a triangular surface fitting scheme in the following form:

$$F = \mathcal{B} + b_p b_q b_r \mathcal{X}$$

where $\mathcal{B}$ interpolates the boundaries and $\mathcal{X}$ is presented as a function that adjusts the cross boundary derivatives. Although F is a point-valued function with geometric meaning, neither $\mathcal{B}$ nor $\mathcal{X}$ represent affine geometric entities (points, vectors, etc).

2. Alternatively, we can rewrite Herron's scheme as

$$F = \sum_{i=p,q,r} \beta_i F_i,$$

where the β_i are Nielson's weight functions. Each F_i is a quartic Bézier patch with cubic boundaries. Two of the interior control points are set using Chiyokura-Kimura's method to achieve G^1 continuity across one boundary. The third control point is a free parameter.

3. When we blend the three patches with Nielson's weights, we know that the resulting patch will interpolate the derivative behavior along all three boundaries.

6.3.5 Gregory and Charrot

1. Gregory and Charrot devised a "side-side" surface scheme that linearly blends three patches, each of which interpolates the crossboundary behavior on two of the three boundaries.

The three patches are then blended using Nielson's weights.

6.3.6 Hagen-Pottmann

1. Similar to Nielson's scheme, except it creates a G^2 surface.
2. Blending functions:
$$\beta_i = \frac{b_j^2 b_k^2}{b_j^2 b_k^2 + b_k^2 b_i^2 + b_i^2 b_j^2}$$
3. Curve construction operator interpolates position, first, and second derivatives at two points.
4. Requires creating a "tangent plane field" and "curvature field" along each boundary.
5. Numerical problems.

6.3.7 References

Design of solids with free-form surfaces H. Chiyokura and F. Kimura, SIGGRAPH 1983.

Finte Element Stiffness Matrices for Analysis of Plates in Bending, R. Clough and J. Tocher, in Proceedings of Conference on Matrix Methods in Structural Analysis, 1965.

A Modified Clough-Tocher Interpolant, G. Farin, CAGD, Vol 2, 1985.

Hybrid Cubic Bézier Triangle Patches, T. Foley and K. Opitz, in Mathematical Methods for Computer Aided Geometric Design II, T. Lyche and L. Schumaker (eds), Academic Press, 1992.

Scattered Data Interpolants: Tests of Some Methods, R. Franke, Math. Compu., 38(1982).

Curvature Continuous Triangular Interpolants, Hans Hagen and Helmut Pottmann, in Mathematical Methods in Computer Aided Geometric Design, T. Lyche and L. Schumaker (eds), Academic Press, 1989.

A C^1 Triangular Interpolation Patch for Computer-Aided Geometric Design, J. Gregory and P. Charrot, Computer Graphics and Image Processing, Vol 13, 1980.

Smooth closed surfaces with discrete triangular interpolants, Gary Herron, Computer Aided Geometric Design, Vol 2, No 4, December 1985.

Assembling triangular and rectangular patches and multivariate splines, Thomas Jensen, in Geometric Modeling: Algorithms and New Trends, G. Farin (ed), SIAM, 1987.

Interpolating Patches Between Cubic Boundaries, L. Longhi, T.R. UCB/CSD 87/313, University of California, Berkeley, Berkeley, CA 94720, October, 1986.

A transfinite, visually continuous, triangular interpolant, Greg Nielson, in Geometric Modeling: Algorithms and New Trends, G. Farin (ed), SIAM, 1987.

Smooth mesh interpolation with cubic patches, Jorg Peters, CAD, Vol 22, No 2, 1991.

Visually Smooth Interpolation with Triangular Bézier Patches, Bruce Piper, in Geometric Modeling: Algorithms and New Trends, G. Farin (ed), SIAM, 1987.

Local surface interpolation with Bézier patches, Leon Shirman and Carlo Séquin, CAGD, Vol 4, No 4, December 1987.

Local surface interpolation with Bézier patches: errata and improvements, Leon Shirman and Carlo Séquin, CAGD, Vol 8, No 3, August 1991.

6.4 Boundary Curves

Missing from the above techniques are methods for constructing boundary curves (which in Nielson's case is also needed for the interior of the patch). There is a fair amount of freedom in the construction of these curves, and the resulting surface shape is very sensitive to these parameters. In this section, I will discuss some simple methods for constructing boundaries, and discuss one more sophisticated method.

6.4.1 Simple Curve Construction

- Note that the boundary curves of the various schemes have to satisfy the conditions required of Nielson's g_v curve constructor:

  - $g_v(0) = V_0$,
  - $g_v(1) = V_1$,
  - $< g_v'(0),\ \hat{N}_0 > = 0$,
  - $< g_v'(1)\, \hat{N}_1 > = 0$,

  where V_0 and V_1 are the vertices we want to interpolate and where $\hat{N}_0$ and $\hat{N}_1$ are the normals we want to interpolate. This data can easily be interpolated by a cubic Bézier curve with control points b_0, b_1, b_2, b_3:

  - $b_0 = V_0$
  - $< b_1 - b_0,\ \hat{N}_0 > = 0$,
  - $< b_3 - b_2,\ \hat{N}_0 > = 0$,
  - $b_3 = V_1$.

  The only difficulty is that b_1 and b_2 have two degrees of freedom each.

- Shirman and Séquin used the following method (which they do *not* recommend) for setting b_1 and b_2. Project V_1 into the tangent plane $(V_0,\ \hat{N}_0)$ giving point pv_1. Set b_1 on the line containing V_0 and pv_1 such that $3|b_1 - b_0| = |V_1 - V_0|$. Do the symmetric construction to set b_2.

  Alternatively, we could do the following:

  1. Set $\vec{v} = \hat{N}_0 \times (V_1 - V_0)$
  2. Set $\vec{w} = \hat{N}_0 \times \vec{v}$
  3. Set $b_1 = b_0 + t\hat{w}$ where $t = |V_1 - V_0|/3$
  4. Construct b_2 in a symmetric fashion.

(Note: The above two methods construct the same point.)

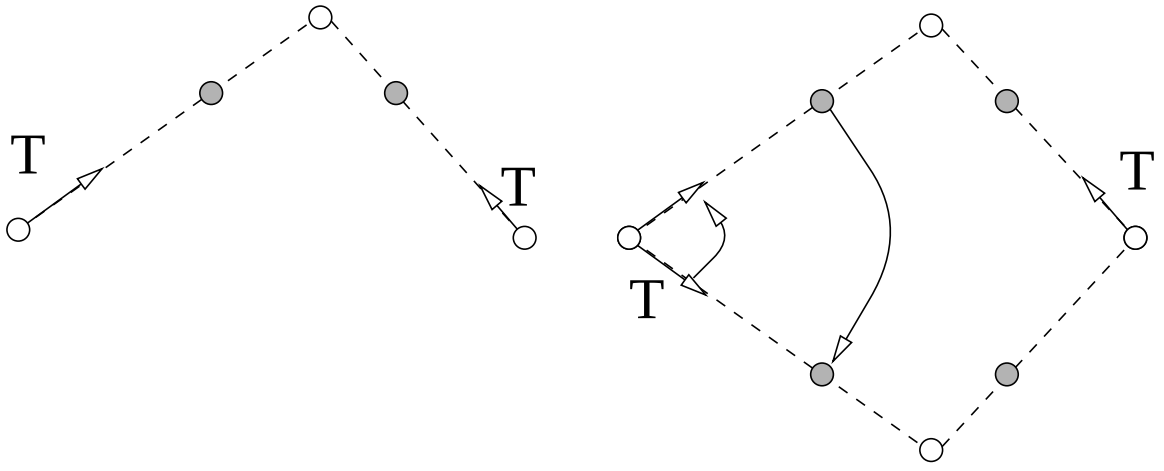
You can imagine variations on these techniques. However, they all construct non-planar boundary curves. The factor of 3 gives us linear precision (for the boundary curve).

- Empirical evidence indicates that planar boundary curves yield surfaces with better shape. To construct a planar boundary curve, we first pick a plane in which to place our boundary curve. We then intersect this plane with the two tangent planes to get tangent lines. Finally, we position the interior control points along this tangent line.

  One approach choice of plane is the following:

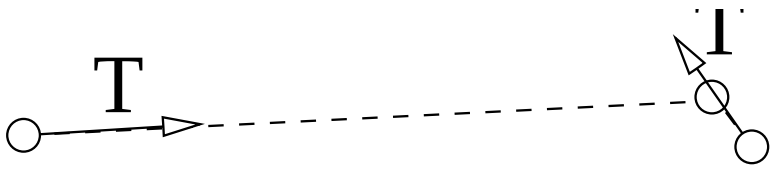
  - Let $\hat{N} = (\hat{N}_0 + \hat{N}_1)/|\hat{N}_0 + \hat{N}_1|$.
  - Choose the plane containing V_0, V_1, and $\hat{N}$ as the plane in which to place the boundary curve.

- Nielson (together with Farin and Hamann) provided different g_v and g_n functions. For g_v, they used generalized degree raised conics (rational quadratics). Since a conic can't model an inflection point, they generalized the degree raised conics by flipping the data when it requires an inflection point, and then flipping one of the degree raised control points to lie in the tangent plane.



In the figure above, the left shows a conic (without weights) with white control points and the degree raised control points (in gray). The Ts are the given tangent data. The right shows tangent data that cannot be fit with a conic. So we flip one tangent, construct the conic, and then flip one of the degree raised control points.

The problem with this scheme is that it is easy to give it data where it will construct a curve with very high curvature at one end:



This will result as what appears to be a crease in the surface.

- Later still, Mann incorporated a cross-boundary scheme of Foley to achieve significant improvements in shape.

6.4.2 de Boor-Höllig-Sabin

1. If we have two points and two vectors in the plane, we can construct a parametric cubic curve C that interpolates the points and has first derivatives at the two points that interpolate the vectors.

   If the data has been sampled off some other curve F, then it can be shown that the error $|F - C|$ is $O(h^4)$ where h is the distance between the two points.

2. Suppose now that we have two points, two unit vectors, and two signed curvature values. Then in general, we can interpolate this data with a parametric cubic curve. Further, the error in the approximation is $O(h^6)$

3. In particular, let $b_0, \ldots, b_3$ be our cubic control points. If the data to interpolate are positions f_0, f_1, tangents d_0, d_1, and curvatures κ_0, κ_1, then we know $b_0 = f_0$ and $b_1 = f_1$. We also know

   $$\begin{aligned} b_1 &= b_0 + \delta_0 d_0 \\ b_2 &= b_3 - \delta_1 d_1 \end{aligned}$$

   for some scalars $\delta_0 > 0$ and $\delta_1 > 0$ to meet our G^1 conditions.

   Our G^2 conditions are

   $$\begin{aligned} \kappa_0 &= 2d_0 \times (b_2 - b_1)/(3\delta_0^2) \\ \kappa_1 &= 2d_1 \times (b_1 - b_2)/(3\delta_1^2) \end{aligned}$$

   We can solve these two equations for δ_0 and δ_1:

   $$\begin{aligned} (d_0 \times d_1)\delta_0 &= (a \times d_1) - \frac{3}{2}\kappa_1\delta_1^2 \\ (d_0 \times d_1)\delta_1 &= (d_0 \times a) - \frac{3}{2}\kappa_0\delta_0^2, \end{aligned}$$

   where $a = f_1 - f_0$. Note that δ_0 is linear in the first equation, and we can trivially solve for it in terms of δ_1 and substitute into the second equation. This gives us a fourth degree polynomial in δ_1 whose real roots can be substituted into the equation for δ_0. We are interested in solutions where both δs are positive.

6.4.3 References:

High accuracy geometric Hermite interpolation, Carl de Boor, Klaus Höllig, and Malcom Sabin, in CAGD, Vol 4, No 4, December 1987.

6.4.4 Crossboundary Schemes

1. In addition to boundary curves, we should also pay attention to the crossboundary construction.
2. Clough-Tocher: Linear variation, quadratic precision
3. Farin: Minimize C^2 discontinuity
4. Foley-Opitz: Cubic precision
5. Davidchuk, Mann: Put Foley-Opitz in parametric setting

6.4.5 References:

A Parametric Triangular Patch Based on Generalized Conics, Bernd Hamann, Gerald Farin, and Gregory Nielson, in NURBS for Curve and Surface Design, G. Farin (ed), SIAM, 1991.

An Improved Side-Vertex Triangle Mesh Interpolant, Stephen Mann, Graphics Interface '98, 1998.

6.4.6 Implementations

1. Implement one of the following surface construction schemes:
   - Shirman-Séquin
   - Nielson
   - Triangular Gregory Patches
   - Catmull-Rom subdivision
   - Loop/Peters subdivision

   Note while Shirman-Séquin is probably the hardest to implement, you can use the tesselator you wrote for the previous problem to render the surface. For the next two schemes, you'll need to integrate tesselation code into the scheme itself. For the subdivision scheme, you will need to make a data structure to access the relevant adjacency information.

   To compute surface normals, you'll have to write numerical approximation code for all but Shirman-Séquin's code. For subdivision schemes, just use the weighted sum of the faces normals around a vertex (this method is unacceptable for the other three schemes).

You should fit your surfaces to the corners of the patches given in Problem 1 of this assignment. Note that subdivision schemes will give surfaces that do NOT interpolate the data points.

6.5 B-patches

1. We'd like a B-spline like construction for triangular patches. That is, we'd like to be able to construct a triangular patch network with automatic C^{n-1} continuity.

   B-patches are a first step towards this. By themselves, they say nothing about continuity. However, their knot arrangement is similar to that of B-spline curves. Recall that a cubic Bézier curve is defined over an interval $[c, d]$ by control points

   $$f(c, c, c),\ f(c, c, d),\ f(c, d, d),\ f(d, d, d),$$

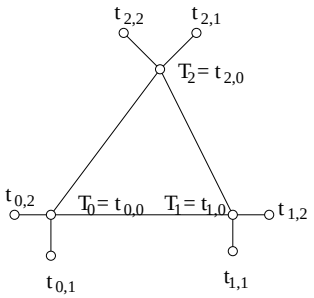
   while the corresponding B-spline reprentation is defined over a knot vector $a \le b \le c < d \le e \le f$ (for some values a, b, e, f) by control points

   $$f(a, b, c),\ f(b, c, d),\ f(c, d, e),\ f(d, e, f).$$

2. For a triangular surface patch, we will start with a domain triangle $\triangle T_0T_1T_2$ and associate a set of knots

   $$\{t_{0,0}, \ldots, t_{0,n-1}, t_{1,0}, \ldots, t_{1,n-1}, t_{2,0}, \ldots, t_{2,n-1}\},$$

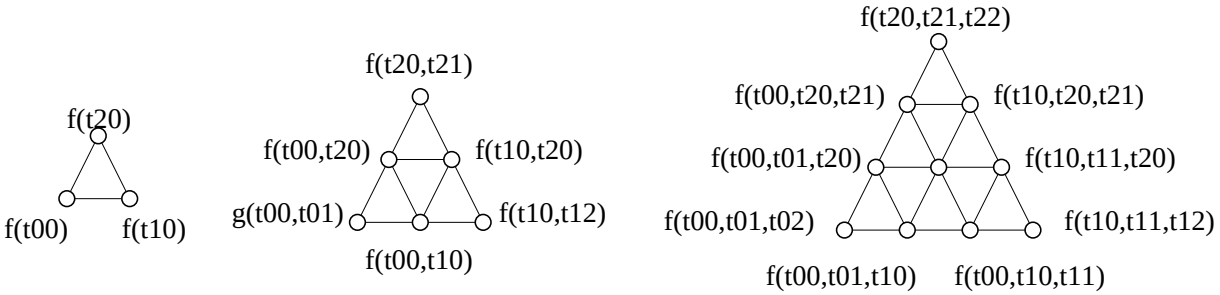
   where $T_0 = t_{0,0},\ T_1 = t_{1,0},\ T_2 = t_{2,0}$. For the cubic case, we have something like

   We now define our control points $P_{\vec{i}}$ (with $\vec{i} = (i_0, i_1, i_2)$ and $|\vec{i}| = n$) as

   $$P_{\vec{i}} = f(t_{0,0}, \ldots, t_{0,i_0}, t_{1,0}, \ldots, t_{1,i_1}, t_{2,0}, \ldots, t_{2,i_2}).$$

   Schematic drawings of degree 1, 2, and 3 control nets are:



with the center control point of the cubic patch having the blossom label $f(t_{0,0}, t_{1,0}, t_{2,0})$.

To evaluate a B-patch, note that in each of the upward pointing triangles the blossom labels only differ in one argument. Thus, we express our domain point in barycentric coordinates with respect to this triangle (of differing arguments) and use those as weights to the control points.

Thus, if we define $\lambda_{\vec{i},d}(u)$ to be the barycentric coordinates with respect to $\triangle t_{0,i_0} t_{1,i_1} t_{2,i_2}$, then we have

$$\begin{aligned} f(t00, t01, u) &= \lambda_{(3,0,0),0}(u) f(t00, t01, t02) + \lambda_{(3,0,0),1}(u) f(t00, t01, t10) + \\ &\quad \lambda_{(3,0,0),2}(u) f(t00, t01, t20). \end{aligned}$$

3. More generally, we can make a de Casteljau type algorithm:

$$\begin{aligned} c^0_{(i_0,i_1,i_2)}(u) &= c_{(i_0,i_1,i_2)} = f(t_{0,0}, \ldots, t_{0,i_0}, t_{1,0}, \ldots, t_{1,i_1}, t_{2,0}, \ldots, t_{2,i_2}) \\ c^\ell_{(i_0,i_1,i_2)}(u) &= \lambda_{(i_0,i_1,i_2),0}(u) c^\ell_{(i_0+1,i_1,i_2)}(u) + \lambda_{(i_0,i_1,i_2),1}(u) c^\ell_{(i_0,i_1+1,i_2)}(u) + \\ &\quad \lambda_{(i_0,i_1,i_2),2}(u) c^\ell_{(i_0,i_1,i_2+1)}(u) \end{aligned}$$

Further, we can construct basis functions:

$$\begin{aligned} B_{(0,0,0)}(u) &= 1 \\ B_{(i_1,i_1,i_2)} &= 0 \quad \text{if } i_j < 0 \\ B_{(i_1,i_1,i_2)} &= \lambda_{(i_0-1,i_1,i_2),0}(u) B_{(i_0-1,i_1,i_2)}(u) + \lambda_{(i_0,i_1-1,i_2),1}(u) B_{(i_0,i_1-1,i_2)}(u) + \\ &\quad \lambda_{(i_0,i_1,i_2-1),2}(u) B_{(i_0,i_1,i_2-1)}(u) \end{aligned}$$

4. Merging the ideas of B-patches with Simplex splines, one can make a type of triangular B-splines. These splines are not ideal in that they do not have a single polynomial patch per face, but they do give a C^{n-1} surface for a triangulated domain. See the references for more details.

6.5.1 References

H.-P. Seidel, Symmetric recursive algorithms for surfaces: B-patches and the de Boor algorithm for polynomials over triangles, Constructive Approximation, 7 (1991).

W. Dahmen, C.A. Micchelli, and H.-P. Seidel, "Blossoming Begets B-splines Built Better by B-Patches," Mathematics of Computations, Vol 59, No. 199, July 1992.

6.6 Evaluating Surface Quality

6.6.1 Line Drawings

Before graphics became cheap enough, quality of surfaces often shown with line drawings.

- The simplest way is to tessellate the surface and draw the outlines of the triangles.
- Hidden surface removal improved things a bit.
- A related line drawing method was to trace isoparametric lines.

  Ie, for parametric surfaces, fix one domain parameter and trace the image of the curve obtained by varying the other parameter value.
- Drawing triangulations would often achieve a form of isoparametric lines, since normally we sample in an isoparametric manner.
- Both triangulations and isoparametric lines fail to display severe shape defects. With some training (by using other techniques), you could learn to spot certain types of defects, mainly be noticing when the isoparametric lines don't have uniform distribution.

6.6.2 Shaded Images

A big step up from line drawings are shaded images. From one view point, shaded images are THE way we want to evaluate surfaces used for geometric design: our primary concern is how the final product looks. But a few problems remain: while we can often say "gee, that surface looks bad", it can be hard to decide why it looks bad. Also, we usually have to rotate the surfaces to try to spot defects, and we're never quite sure we've checked things enough. Finally, we need material properties for the surfaces (diffuse and specular reflection coefficients). These material properties have a strong impact on how we perceive the surface and any shape defects.

6.6.3 Curvature

Curvature plots are a common method for evaluating surface quality. Usually we get a scalar curvature value that we then map to a colour. Discontinuities in colour or rapid changes in colour indicate shape defects.

1. To understand surface curvature, we begin by looking at curve curvature for planar curves. The curvature of a curve is defined by looking at the best approximating circle to a point on a curve. If $F(t)$ is our curve, then the *curvature* of $F(x)$ is

$$\kappa(x) = \frac{f'(x) \times f''(x)}{||f'(x)||^3}.$$

   The curvature κ is the reciprocal of the radius of the best approximating circle to the curve at x, ($1/\kappa$ is known as the *radius of curvature*).

2. We will also care about on which side of the curve the best approximating circle lies, and we arbitrarily choose one side to have positive sign and the other side to have negative sign.

3. Note that at an inflection point, the curvature is 0 and the radius of curvature is infinite.

Surface curvature is harder to define than curvature for a curve. The issue is that at a point P on the surface S, the surface curves in different amounts in each direction $\vec{v}$. We could try a curve C passing through P and asking for the curvature of the C at P. However, even if we insist that the curve be a planar curve (for our definition of curve curvature is for planar curves), we still have an infinite number of planes that slice the surface at P and contain $\vec{v}$.

The last difficulty is usually addressed by considering *Normal Section Curvature*, where we slice the surface with planes that contain the normal $\hat{n}$ of S at P, and ask for the signed curvature of these curves at P.

If we consider the normal section curvature in all directions around a point P, we find that it is a quadratic function of the direction (roughly speaking) with a maximum and minimum value (although these may sometimes be infinite). The maximum and minimum signed normal curvatures are known as the *principle curvatures* κ_1 and κ_2. The directions in which we have these maximum and minimum values are known as the *principle directions*.

The term $K = \kappa_1\kappa_2$ is called *Gaussian Curvature*. Note that Gaussian curvature is independent of our choice of sign for signed curvature. We are most concerned about whether the Gaussian curvature is positive, negative or zero (although at times we will certainly care about the magnitude of Gaussian curvature). A point on a surface with positive Gaussian curvature bends in the same direction (relative to the surface normal) in any direction we move in the tangent plane. A sphere is an example of a surface where every point has positive Gaussian curvature (in fact, the Gaussian curvature is constant over the entire sphere).

A point on a surface that has negative Gaussian curvature is a hyperbolic point. And a point on the surface that has zero Gaussian curvature is flat in at least one principle direction. Thus, while a plane has zero Gaussian curvature, so does a cylinder and a cone.

The torus is an example of a surface that has positive Gaussian curvature (on the "outside"), negative Gaussian curvature (on the "inside") and zero Gaussian curvature (on the "top" and "bottom").

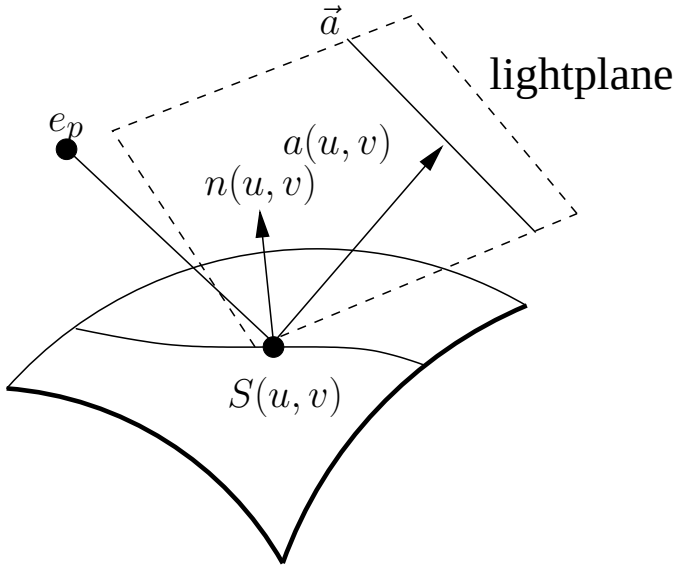


Figure 6.1: Reflection lines. Figure from Holger Theisel

(There is one additional type of surface worth mentioning, which is a *monkey saddle*.)

The term $(\kappa_1 + \kappa_2)/2$ is known as *Mean Curvature*. Unlike Gaussian curvature, mean curvature depends on our choice of sign for planar curvature. Both Gaussian and mean curvature have their uses in assessing surfaces.

A final issue is how to map the scalar curvature value to a colour for surface shading. One method is to map the scalar value to the Hue of the HLS color space, using L=0.5 and S=0.9 (or similar values), and then convert to RGB if needed. We still need to specify the mapping of curvature to Hue. One mapping is to truncate the curvature range from $-\infty, \infty$ to $-k, +k$, and then map $-k$ to blue, 0 to green, and $+k$ to red.

6.6.4 Lines on a Surface

While isoparametric lines are one of the simplest forms of lines on a surface to compute, they aren't very useful for evaluating surface shape. *Reflection Lines* are lines on a surface that are more useful. Conceptually, a reflection line is the curve on a surface obtained by reflecting a line through the surface. Typically, rather than reflecting in the ray tracing sense, we merely draw a curve on the surface at all points where reflection of the view direction in the surface intersects the reflection line.

Mathematically, let the eye be at e_p. Let $a(u, v)$ be the reflection of the ray from e_p to the surface point $S(u, v)$ through the surface normal $n(u, v)$. Let $\vec{a}$ be the direction of the line of light, and let Π be the plane containing the line of light and the point on the the surface $S(u, v)$. Let $\hat{p}(u, v)$ be perpendicular to this plane. The condition for the point $S(u, v)$ to be on the reflection line is

$$a(u, v) \cdot \hat{p}(u, v) = 0.$$

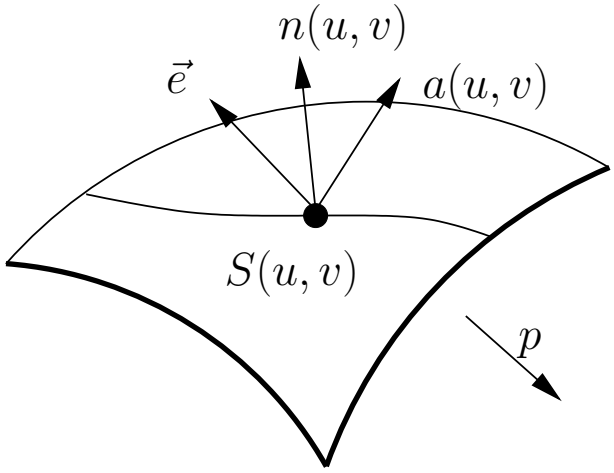


Figure 6.2: Reflection lines with eye, line at infinity.

Reflection lines simplify a lot if we place the eye at infinity (so we just specify a viewing direction $\vec{e}$) and the reflection line at infinity (which we now specify with the perpendicular $\hat{p}$). We again just check that $a(u,v)\cdot\hat{p}=0$ as our condition. We can get a family of reflection lines by varying the line at infinity.

Isophotes are defined from an eyepoint e_p and a constant c as all points on a surface $S(u,v)$ where

$$\frac{n(u,v)\cdot(e_p - S(u,v))}{|e_p - S(u,v)|} = c.$$

Again, we can simplify things by putting the eye position at infinity, leaving an eye direction: $n(u,v)\cdot\vec{e}=c$.

With both reflection lines and isophotes, the continuity of the lines on the surface is one less than the continuity of the surface. So a C^0 surface will yield reflection lines and isophotes that are C^{-1}, a C^1 surface will yield reflection lines and isophotes that are C^0 and so on.

Typically, for reflection lines and isophotes, we render a set of them. For example, for isophotes, we might render them for $c = 0, 0.25, 0.5, 0.75, 1.0$, giving five sets of curvess on the surface. For a triangulated surface, we can create this line set as follows: for each triangle vertex, we compute $v_i = \vec{e}\cdot\hat{n}_i$, giving three scalar values v_i. For each isophote values c_j, if $c_j < v_i$ for all i or if $c_j > v_i$ for all i, then the isophote is assumed not to cross the triangle. Otherwise, we have two sets of vertices at which c_j lies between the corresponding v_i values. We linearly interpolate between the two v_i values to find the crossing point for the isophote value, and draw a line segment between these two locations.

Alternatively, we could use a shader to draw the isophote on the triangle.

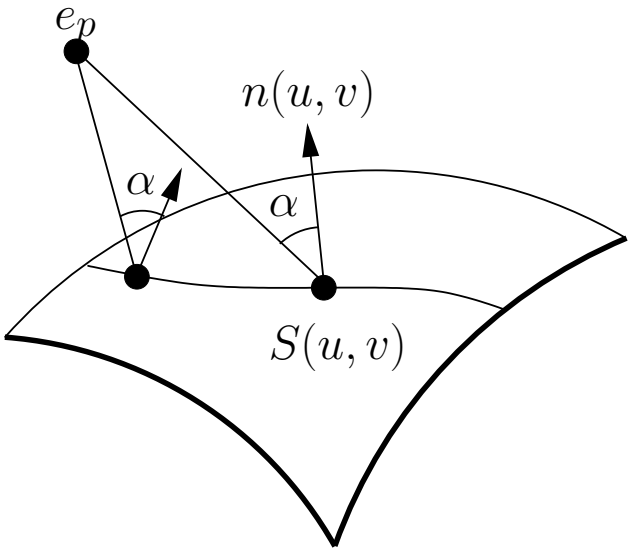


Figure 6.3: Isophotes.

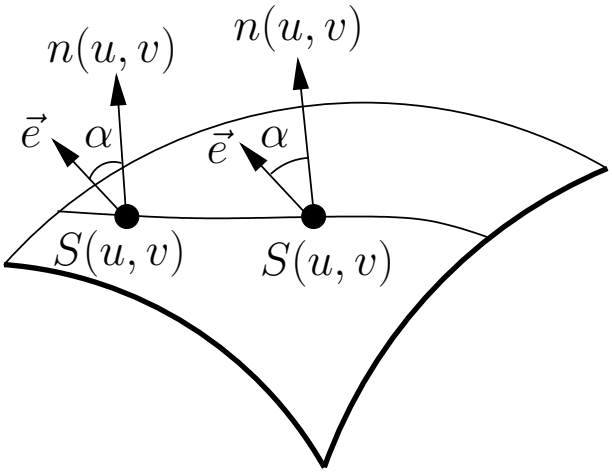


Figure 6.4: Isophotes with eye at infinity.

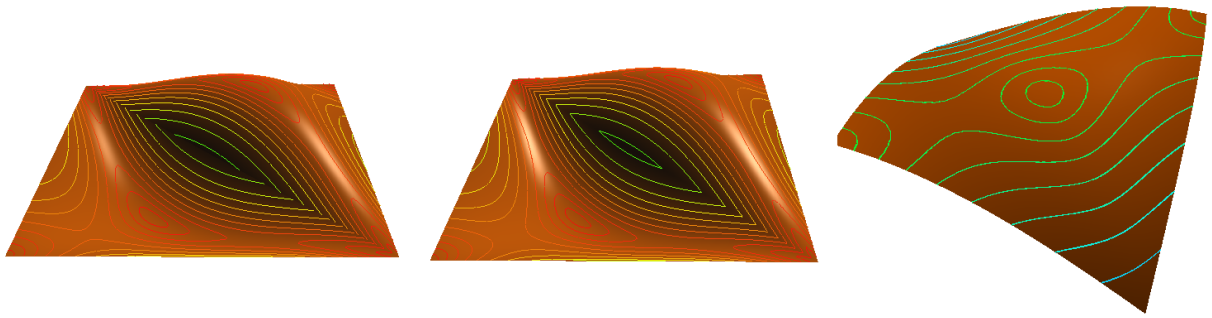


Figure 6.5: Isophotes on C^0, C^1, and C^2 surfaces. Figure from Mark Belcarz

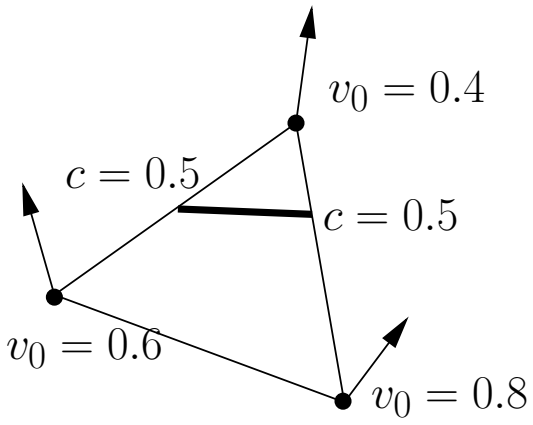


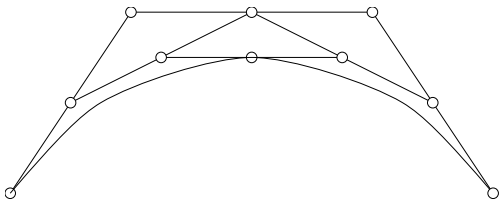
Figure 6.6: Isophotes on a triangle.

6.7 Subdivision Surfaces

Subdivision provides an alternative method for constructing smooth surfaces. The advantages of subdivision schemes are that they handle automatically the nasty problems that the schemes we've seen so far have, namely we don't have to work to get continuity (not C^1 continuity, anyway), and we can start with data meshes of arbitrary connectivity. However, the meshes should represent a manifold - the "arbitrary" part refers to the number of edges on a face and the number of faces meeting at a vertex (although this leads to extraordinary vertices, which are their own problem).

6.7.1 Subdivision Curves

1. If we repeatedly subdivide a Bézier curve, then in the limit, the control polygons converge to the curve.



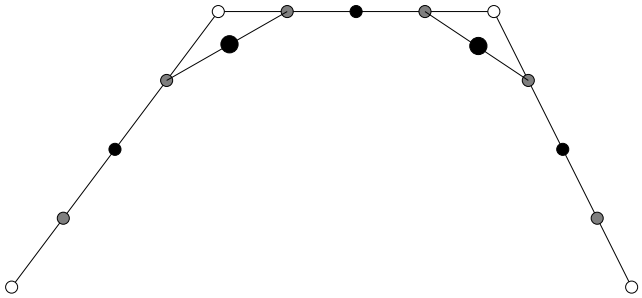
2. Chaiken's algorithm Chaiken's algorithm was originally described as a corner cutting algorithm: Given initial points $\{C_0^0, \ldots, C_n^0\}$. Compute

$$\begin{aligned} C_{2i}^1 &= \frac{3}{4}C_i^0 + \frac{1}{4}C_{i+1}^0 \\ C_{2i+1}^1 &= \frac{1}{4}C_i^0 + \frac{3}{4}C_{i+1}^0 \end{aligned}$$

and repeat this process. If we let m_i be the midpoint of $C_i^0 C_{i+1}^0$, then the point

$$p_i = \frac{1}{4}(m_{i-1} + 2C_i + m_i)$$

is a point on the curve. In the figure below, the white points are the C^0, the gray points are the C^1, the small black points are the m_i and the large black points are points on the curve.



Note that it is simpler to compute the points on the curve as

$$p_i = (C^1_{2i} + C^1_{2i+1})/2.$$

Further note that the curve is actually a C^1 piecewise quadratic Bézier curve. The points $m_{i-1},\ C^0_i, m_i$ form the control points of the ith segment (for $i = 1$ to n). All the algorithm is really doing is a midpoint subdivision on these curves.

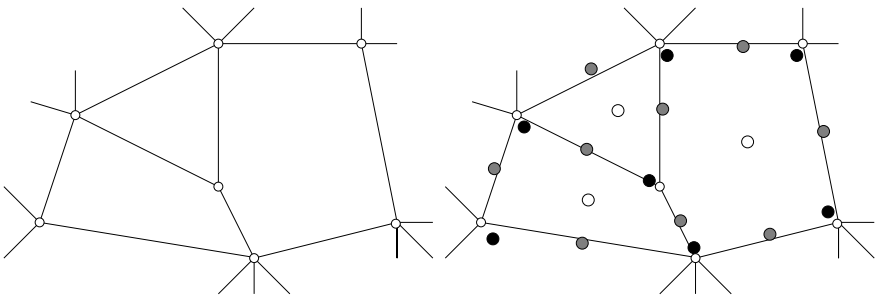
6.7.2 Subdivision Surfaces

1. Subdivision surfaces are an extension of the curve subdivision idea. Essentially, we start with a "polyhedron" (its faces are not required to be planar). At each step of the subdivision, we add one new point for each face, edge, and vertex. In the limit, these points will converge to a surface. The details of where the points are located and whether we just add points for each edge, or if we add multiple points per face, etc., vary from scheme to scheme. We will look at a couple of schemes.

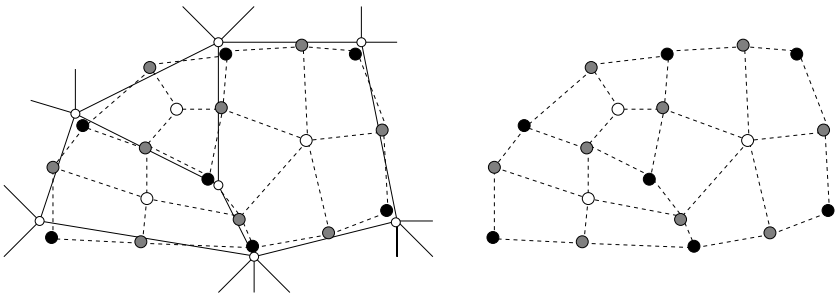
Catmull-Clark Subdivision

1. For each face, edge, and vertex of the polyhedron we will build a new vertex (call them f_i, e_i, and v_i respectively). We will create a set of new faces, one for each face/vertex pair (where the vertex bounds the face) by adding in edges from face to edge vertices and from vertex to edge vertices. The following sequence of pictures should clarify this process.

   On the left we have the initial mesh. Next, we add a new vertex for each face (shown in white), a new vertex for each edge (shown in gray) and a new vertex for each vertex (show in black):



   Now we connect together the appropriate vertices. On the left, the old and new meshes are superimposed, while on the right we have just the new mesh:



2. Missing from the diagrams is how to set the location of the new points. Catmull-Clark use the following scheme:

   - Set each face vertex f_i as the average of the vertices of the face;
   - Set each edge vertex e_i as the average of the vertices on the edge and the (new) face vertices for the two faces adjacent to the edge;
   - Set each vertex vertex v_i as

   $$v_i^n = v_i^o + \frac{1}{n}\sum_{j=1}^{n}(m_j^o - v_i^o) + \frac{1}{n}\sum_{j=1}^{n}(f_j^n - v_i^o),$$

   where m^o are the midpoints of the old edges, v_i^o is the vertex in the previous step, and n is the number of edges adjacent to v_i.

   Alt formula:

   $$F/n + 2M/n + P(n-3)/n,$$

   where F is the average of newly created face points, M is average of old edge midpoints, and P is old position.

   I'm not sure where first formula came from; the second formula is from the Catmull-Clark paper.

3. Note that while the initial mesh may have faces with an arbitrary number of sides, after one refinement all faces will have four sides.

4. Except near vertices that initially had other than four neighbors and faces that had other than four sides, this surface converges to bicubic spline surfaces.

Doo-Sabin Subdivision

1. In the same journal issue in which Catmull-Clark subdivision appeared, the Doo-Sabin subdivision scheme was also published. While Catmull-Clark is related to bicubic tensor product surfaces, Doo-Sabin is related to biquadratic tensor product surfaces.

2. The idea is that starting with a mesh, for each face, we create a new vertex for each edge of that face

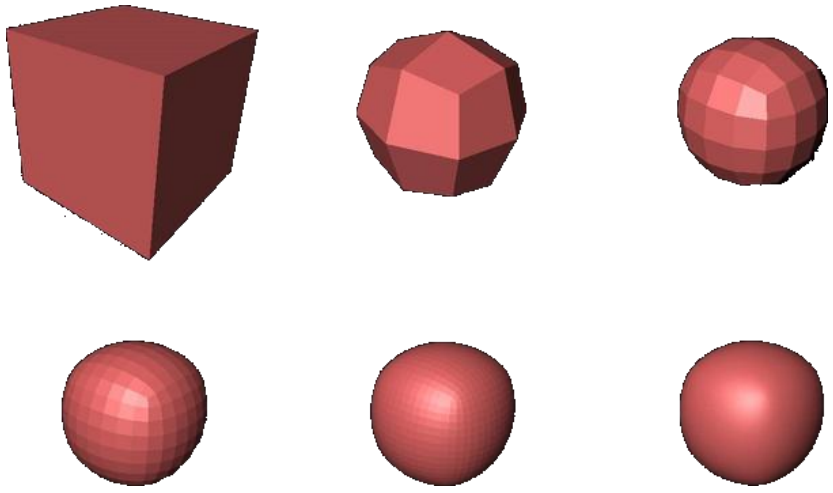
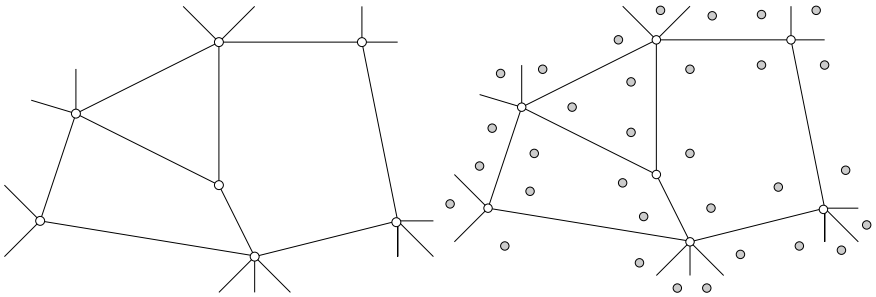
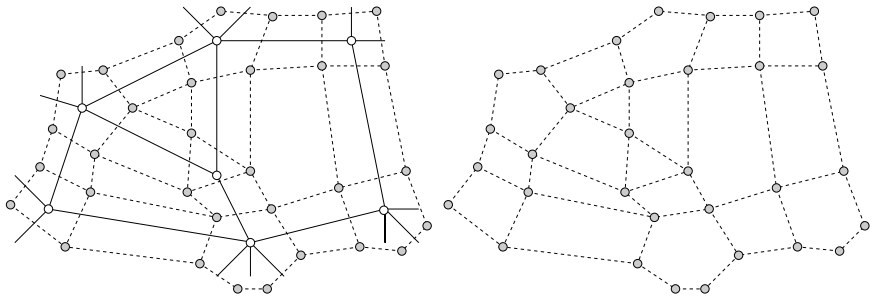


Figure 6.7: Catmull-Clark Subdivision



3. Then we create three types of new faces:
   (a) For each vertex of the original mesh, we create a face from the ring of new vertices surrounding it;
   (b) For each face of the original mesh, we create a face from the ring of new vertices inside it;
   (c) For each edge of the oringal mesh, we create a new face from the four vertices created for the two vertices of the edge.

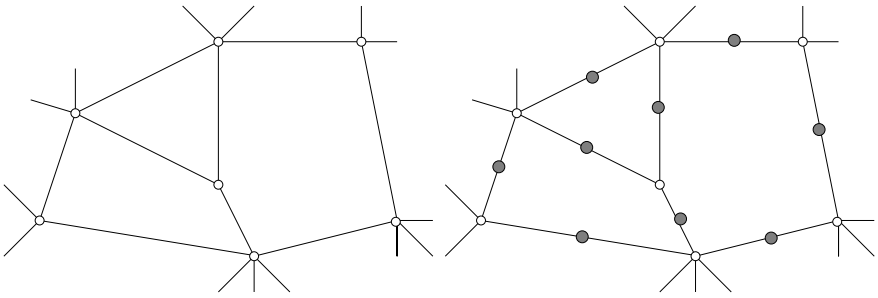


4. Details of the location of the new vertices can be found in Farin's book.

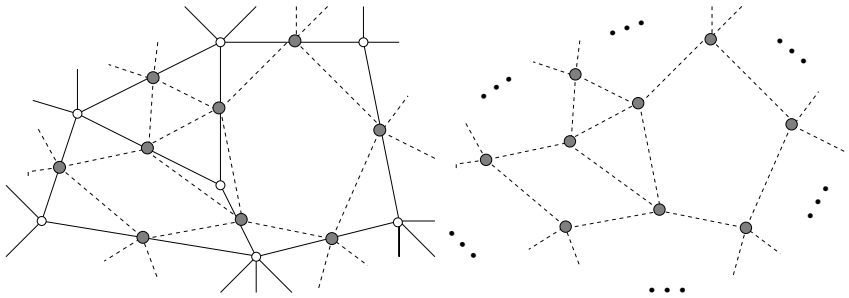
The Simplest Subdivision Scheme

1. A large variety of subdivision schemes exist. For example, Jorg Peters has a scheme where the only new points at each level of subdivision are the average of the two vertices on each edge. New faces are constructed (one for each old face, one for each old vertex), and the process is repeated.

   Again, starting from the same initial mesh, we first subdivide each edge:



   Next, we connect the edge-points to create new faces:



2. While simple, the surfaces created by this scheme are of extremely poor quality.

6.7.3 Details

1. There is much more written on subdivision schemes. See for example papers by Loop and Peters for some variations. In particular, there are interpolatory subdivision schemes.

2. Another issue of interest is meshes with boundaries. Such meshes usually require special subdivision rules along the boundaries.

3. Subdivision surfaces solve the hard problems faced by triangular polynomial patches (and to a lesser extent, rectangular polynomial patches): they can start with an arbitrary mesh and construct a surface without the hard continuity problems faced by the polynomial schemes.

However, they still fail the vertex consistency problem (at the extraordinary vertices, where most subdivision schemes yield surfaces are only C^1).

Usually, you can spot the extraordinary points in the final mesh (if nothing else, there are texture mapping issues near them). Most models have locations where you don't mind a special point (such as the tip of the nose), and if you can place your extraordinary points at these locations then usually they're not a problem.

4. Subdivision schemes are not as easy to implement as they might look. The biggest implementation problem is one of data structures: how do you store the data and how do you update the mesh after subdivision. The easy data structures for handling this don't have all the nice properties you'd like, and that harder data structures are, well, harder.

5. To get a smooth looking surface for rendering, you need to tessellate to a fairly finely, and you need to generate normals. The first issue prohibits the use of heavy-weight data structures like winged-edge, and really promotes the notion of "patch-like" evaluation: taking a subset of the mesh, subdividing it to a fine level, rendering it, then proceeding to the next "patch-like" region. Algorithms exist for such rendering, but they are non-trivial (although much of the complexity is in creating an algorithm that has a reasonable hardware implementation).

   The normal question is another sticky question. Since the subdivision surface is a limiting process, technically you need to subdivide an infinite number of times before you obtain points on the surface. In practice, subdividing a small (3-6) number of times produces an approximation to the surface that is sufficient for rendering. But what about normals to the surface? Most implementations just approximate the normal by averaging the neighbors of a vertex (at the highest level of refinement). While sufficient for many applications, it adds a further cost and data structure issue, and is somewhat unsatisfying.

6. Another issue facing subdivision schemes is that of parameterization. The subdivision surface itself has no natural parameterization, which is expected since they can be used on genus 0 surfaces which can not be fully parameterized.

   However, parameterization is important in rendering, since we commonly want to texture map our surfaces. Thus, to texture map the subdivision surface, we need to find a parameterization of the surface.

   A lot of work has been done on finding parameterizations of subdivision surfaces. The methods usually involve flattening the surface into a triangulation in the plane. Two issues that arise are

   (a) To flatten the mesh, you have to cut it, which results in seams on the surface; smart placement of these seams is important.

   (b) The flattening process is not unique. The idea is to find a flattening such that the resulting texture map looks good.

7. The methods above suggest ways in which to compute the vertices at each step of the subdivision. The obvious way is not the most efficient. See the Warren-Weimer book for details on how to efficiently compute the new vertices.

8. Nothing presented here supports the claim that the repeated subdivision converges to a surface with smoothness of any type. Such proofs are complicated (although not horribly so); see the Warren-Weimer book for the proofs.

9. Finally, note that many subdivision schemes face many of the shape problems that are encountered by polynomial patch schemes. Typically, optimization methods are required to yield reasonable looking surfaces.

   In particular, while subdivision surfaces usually do reasonably well on "human smoothed" data, they produce bumpy surfaces for more random data. By "human smoothed", I mean that people tend to create and smooth data in a way that results in smooth subdivision surfaces. When sampling points on a cylinder, for example, we will sample concentric rings, with samples also lying on lines parallel to the cylinder's axis. When subdividing a mesh of such data, a nice, smooth surface results.

   If instead we picked random point on the surface of the cylinder, triangulated them in a reasonable manner (ie, so that the resulting mesh was cylinder-like), then the resulting subdivision surface will be bumpy.

   While people have successfully created subdivision surfaces from laser-scan data, note that usually this data is reduced by a smoothing process that should be considered part of the surface construction technique.

6.7.4 References

An algorithm for high speed curve generation, G. Chaiken, Computer Graphics and Image Processing, Vol 3, 1974

Efficient, Fair Interpolation using Catmull-Clark Surfaces, Mark Halstead, Michael Kass, Tony DeRose, SIGGRAPH 93.

Charles Teorell Loop, Masters thesis, Smooth Subdivision Surfaces Based on Triangles, Dept of Mathematics, University of Utah, August, 1987.

The Design of Curves and Surfaces by Subdivision Algorithms, Alfred Cavaretta and Charles Micchelli, in Mathematical Methods for Computer Aided Geometric Design, T. Lyche and L. Schumaker (eds), Academic Press, 1989.

The Simplest Subdivision Scheme for smoothing polyhedra, J. Peters and U. Reif, ACM TOG, 16(4); 1997; 420–431.

Subdivision Methods for Geometric Design, Joe Warren and Henrik Weimer, Morgan Kauffmann, 2002

Artifacts in Recursive Subdivision Surfaces, Malcolm Sabin and Loíc Barthe, in Curve and Surface Fitting, Saint-Malo 2002, A Cohen, J-L Merrien, L Schumaker (eds), Nashboro Press; 2003; pg 353-362.

Chapter 7

Functional Interpolation With Polynomials

In this section, we consider the problem of interpolating data above the plane with a single polynomial. Because of a variety of problems in using polynomials to interpolate such data, typically people usually use other methods such as thin-plate splines instead.

The goal of this section if to understand why these problems occur with polynomials, and to realize that polynomials can do a much better job at interpolating data than previously realized.

7.1 Univariate

Problem: given $n+1$ pairs (x_i, y_i)
Find: degree n polynomial f such that $f(x_i) = y_i$
Solution: $f = \sum_{j=0}^{n} c_j x^j = XC$, where $C = [c_0, ..., c_n]^t$ and $X = [1, x, ..., x^n]$. We want c_j such that $f(x_i) = y_i$. In matrix form,

$$VC = Y$$

where $Y = [y_0, ..., y_n]^t$ and $V_{i,j} = x_i^j$. Then the solution is

$$C = V^{-1}Y$$

V is known as the *Vandermonde* matrix. While the inverse exists if the x_i are unique, computation of V^{-1} is numerically unstable (although it is possible to get the interpolant without inverting the matrix). And even if there is a solution, the answer may be poor. However, the solution is unique.

7.2 Multivariate

Problem: given n sets of data X_i, z_i with $X_i \in \mathcal{R}^d$.

Find: a polynomial f such that $f(X_i) = z_i$. We are interested in the case of $d = 2$ (surfaces), so $f(x_i, y_i) = z_i$.

The multivariate problem is far more interesting than the univariate problem. In particular, in the univariate case, when we increase the degree of the polynomials by one, we get one more basis function. In the bivariate case, when we increase the degree from degree $k-1$ to degree k, we get $k+1$ more basis functions. Thus, we can't interpolate our data with a complete degree k polynomial space except in the rare case when the number of data points is the size of a polynomial space of degree k.

So to interpolate n data points, we have to use a polynomial space where $\#\Pi_k \geq n$, where Π_k is a polynomial space of degree k and $\#\Pi_k$ is the size of Π_i. Thus, we need to choose a subset of Π_k as a basis for interpolating our data.

Once we've chosen a basis B, we proceed as in the univariate case, constructing a Vandermonde matrix for the data and inverting it. But unlike the univariate case, V may be singular (i.e., not invertible). As a simple example, if we have six points, then we should be able to interpolate the data with a quadratic, since $\#\Pi_2 = 6$. However, if the x_i, y_i values lie on the unit circle centered at the origin, and if $p(x, y) = x^2 + y^2 - 1$ were one of our basis functions, then $p(x_i, y_i) = 0$ and our Vandermonde matrix would be singular. Since $x^2 + y^2 - 1$ lies in the space of degree 2 polynomials, then V must be singular for this data set when trying to interpolate with Π_2.

Related to this, if we have nm x_i, y_j values that lie on a $n \times m$ grid, then a degree nm polynomial is required to interpolate the data, even though a much lower degree polynomial would be indicated by the count of the data.

Things are even worse than they seem. Consider any fixed basis. For example, as a basis for n data points, choose the smallest k such that $\#\Pi_k \geq n$. Choose for our basis $\#\Pi_{k-1}$ and for the remaining $\ell = n - \#\Pi_{k-1}$ basis functions, choose $x^k, x^{k-1}y, x^{k-2}y^2, ..., x^{k-\ell+1}y^{\ell-1}$. Call this basis *XBias*, since we are biasing things towards the x variable. Fix any $n-1$ of the data points and move the nth data point around the plane. What you find is that there is a curve C where if x_n, y_n lies on C then the Vandermonde matrix is singular. Worse, if x_n, y_n is near C then the error in the interpolant is huge.

As an example, consider Figure 7.1. Suppose we want to approximate the function $y = f(x) = x^3/6$ with a quadratic polynomial. One way to approximate it is to sample $f(x)$ at no more than six points and construct a quadratic function that interpolates those points. If we have six points, this is straightforward: we use the basis $\{1, x, y, x^2, xy, y^2\}$ and proceed as in the univariate case (construct a Vandermonde matrix, etc.).

But what if we have fewer than six points? The other three parts of Figure 7.1 shows that the choice of basis functions has a huge impact on the approximation. In all three cases we interpolate data at

x	y	x^3/6
0.9294600	0.3689200	0.1338261
0.9697300	0.2441800	0.1519852
-0.2211900	-0.9752300	-0.0018036
-0.9653400	-0.2610000	-0.1499304

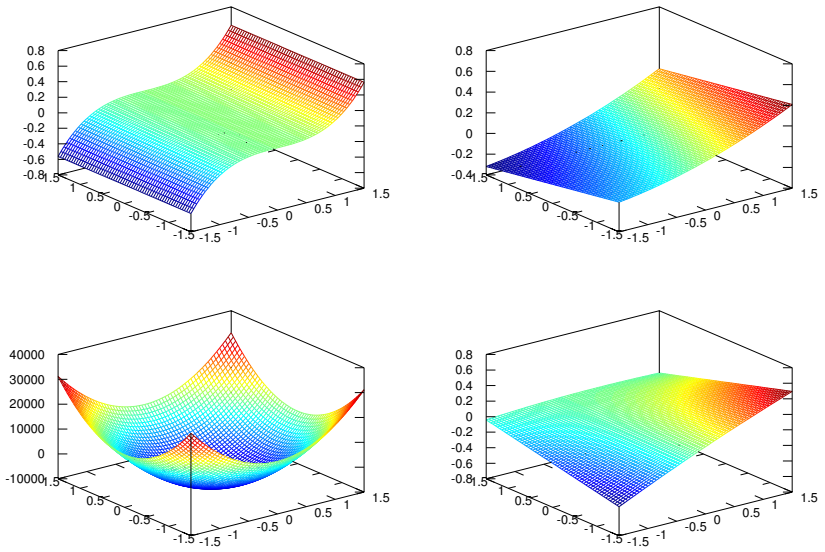


Figure 7.1: Upper left: $y = x^3/6$. Upper right: interpolate data at four points using $\{1, x, y, x^2\}$. Lower left: interpolate data at five points using $\{1, x, y, x^2, y^2\}$. Lower right: interpolate data at five points using $\{1, x, y, x^2, xy\}$.

and in the bottom row we also interpolate

```
-0.4233400    0.9059700  -0.0126449
```

I.e, in the upper right we interpolate the data at four point; in both bottom figures, we interpolate the data at five points. On the bottom row, we see that choosing y^2 for the fifth basis function leads to an approximation that is 10,000 times worse than if we choose xy for the fifth basis function.

There is a construction that tries to make a smart choice of basis functions and avoid these problems. In particular, the above problems occur because we fixed the polynomial basis without considering the data we want to interpolate. The Least is a construction that constructs an interpolation basis by looking at the data we want to interpolate. The Least iterates through the data points, selecting the next data point to interpolate and the next polynomial to add to the basis by choosing the pair that minimizes the error in the interpolant. This scheme is vastly superior to working with a fixed basis because it avoids choosing a data point that is near the Lagrange polynomial for the previously chosen data points, and it chooses the basis function for this data point to maximize the reduction in error of the interpolant.

As a bonus (a big bonus), with some modifications, the Least will generate either a Newton basis or a Lagrange basis. For our purposes, this means that the Vandermonde will either be triangular or diagonal, both of which are numerically stable to invert.

The Least isn't perfect, however, and in some situations it will be forced into a Bad Choice of basis functions. To understand the Least, we have to introduce varieties and ideals.

7.3 Varieties and Ideals

Definition: A subset $V \subset \mathcal{R}^d$ is called a *variety* if there exists a collection of polynomials $P \subset \Pi(\mathcal{R}^d)$ such that $p(v) = 0$ for all $v \in V$ and for all $p \in P$.

Closely related to varieties is the notion of an ideal.

Definition: An *ideal* is a collection of polynomials I such that

1. $0 \in I$

2. I is closed under addition; i.e., if $p, q \in I$ then $p + q \in I$

3. I is closed under multiplication by Π, i.e, if $q \in I$ then for any polynomial p, $pq \in I$.

In short, an ideal I is a collection of polynomials that are 0 at all points in a collection of points V known as a variety.

Our interest in ideals and varieties is that for data interpolation using a fixed basis, along a particular variety, the Vandermonde matrix will have no inverse.

7.4 Newton and Lagrange Bases

A Newton basis for a point set Θ is $\{\nu_i \in \Pi : i = 1, ..., n\}$ such that

$$\nu_i(\theta_j) = 0 \quad \text{if } j < i, \quad \nu_i(\theta_i) \neq 0.$$

A Newton basis yields a lower triangular Vandermonde matrix.

A Lagrange basis $\{\ell_\theta : \theta \in \Theta\}$ is such that

$$\ell_\theta(\vartheta) = \begin{cases} 0 & \text{if } \vartheta \neq \theta \\ 1 & \text{if } \vartheta = \theta \end{cases}$$

For a Lagrange basis, the Vandermonde is the identity matrix, which means

$$f = \sum_{\theta \in \Theta} f(\theta) \ell_\theta.$$

In our variation of the Least, we will construction a Newton basis, which can in turn be converted to a Lagrange basis if desired.

7.5 The Least

The presentation here is not on the Least as originally presented by de Boor and Ron, who presented things as some somewhat mysterious matrix operations. Instead, this is a more symbolic presentation, showing the mathematics behind the Least and polynomial interpolation in general. The approach will be to construct a Newton bases to interpolate the data.

Let $L(\Theta) = \{\ell_\theta \in \Pi(\mathcal{R}^d) : \theta \in \Theta\}$ be any collection of Lagrange polynomials on Θ. The *projection* of a function f on to L is

$$P_L(f) = \sum_{\theta \in \Theta} f(\theta)\ell_\theta,$$

which should look similar to interpolation using a Lagrange basis. The error in this projection is

$$f - P_L(f) = f - \sum_{\theta \in \Theta} f(\theta)\ell_\theta.$$

Define $g_{L,\alpha}$ as the error of interpolating the monomial t^α with $L(\Theta)$:

$$g_{L,\alpha} = t^\alpha - \sum_{\theta \in \Theta} \theta^\alpha \ell_\theta(t),$$

where t is d-variate, and $\alpha \in \mathcal{Z}^d$.

Our interest in $g_{L,\alpha}$ is that the set $\{g_{L,\alpha} : \alpha \in \mathcal{Z}^d_+\}$ covers the ideal $I(\Theta)$. Since we will be looking for Newton polynomials, the set of $g_{L,\alpha}$'s for the θ_i that we've already processed is a good place to look for our next basis function.

The following theorem will provide the formulas needed for some of the update steps of our algorithm.

Theorem: Let $L_{i-1} = \{\ell_{1,i-1}, ..., \ell_{i-1,i-1}\}$ be a Lagrange basis for Θ_{i-1}. Given any $\ell_{i,i}(t) \in I(\Theta_{i-1})$ such that $\ell_{i,i}(\theta_i) = 1$, one can construct a Lagrange basis $L_i = \{\ell_{1,i}, ..., \ell i, i\}$ for Θ_i with

$$\ell_{j,i}(t) = \ell_{j,i-1}(t) - \ell_{j,i-1}(\theta_i)\ell_{i,i}(t), \quad \text{for all } j < i. \tag{7.1}$$

We will make an array of the $g_{L,\alpha}$'s. Initially this array will be

$$g_{L_0} = [1,\ t_x, t_y,\ t_x^2, t_x t_y, t_y^2,\ ...].$$

One can construct $g_{L_i},\ i > 0$ by

$$g_{L_i} = g_{L_{i-1}} - g_{L_{i-1}}(\theta_i)\ell_{i,i}(t). \tag{7.2}$$

Note in particular that g_{L_i} covers the ideal $I(\Theta_i)$. Assuming that $g_{L_{i-1}}$ covers the idea $I(\Theta_{i-1})$, it is straightforward to check that for $g_i \in I(\Theta_i)$ we have $g_i(\theta_j) = 0$ for $j < i$:

$$g_i(\theta_j) = g_{i-1}(\theta_j) - g_{i-1}(\theta_j)\ell_{i,i}(\theta_j) = 0,$$

for some $g_{i-1} \in I(\Theta_{i-1})$, since $g_{i-1}(\theta_j) = 0$ and $\ell_{i,i}(\theta_j) = 0$. When $i = j$, we also have $g_i(\theta_i) = 0$, since $\ell_{i,i}(\theta_i) = 1$.

Repeatedly applying this theorem to construct a set of $\ell_{i,i}$'s, it's easy to see that the set $\{\ell_{i,i} : i = 1, ..., \#\Theta\}$ is a Newton basis for Θ, giving the following algorithm for constructing a Newton basis to interpolate Θ.

```
Initialize g_{L_0} = [t^α : α ∈ Z^d_+]
for i=1 to #Θ
    pick θ_i ∈ Θ\{θ_1, ..., θ_{i-1}}
    select ℓ_{i,i} = Σ c_α g_α such that ℓ_{i,i}(θ_i) = 1.
    g_{L_i} = g_{L_{i-1}} − g_{L_{i-1}}(θ_i) ℓ_{i,i}(t)
end
Return Newton Basis {ℓ_{i,i} : i = 1, ..., #Θ}
```

As an additional step, we can compute a Lagrange basis by using equation (7.1) inside the loop.

Missing from this algorithm are details about the steps "pick θ_i" and "select $\ell_{i,i}$". In the next section, we will illustrate the algorithm with a simple choice of θ_i and $\ell_{i,i}$ and in the section after that, we will give an example showing that this is (at times) a very poor way to pick $\ell_{i,i}$.

7.5.1 Example

As an example of the algorithm, we will choose the data points in the order in which they are given, and we will choose the first non-zero element in g_{L_i} as our next basis function (which we have to scale to obtain the requirement that $\ell_{i,i}(\theta_i) = 1$). This is essentially what XBias is doing.

Consider the point set $\Theta = \{(1,0), (0,1), (-1,0), (0,-1)\}$. We start with

$$g_{L_0}(t,s) = [1,\ t_x,\ t_y,\ t_x^2,\ t_x t_y,\ t_y^2,\ \dots\ ].$$

With $\theta_1 = (1,0)$, select $\ell_{1,1}(t) = 1$, then

$$g_{L_1}(t,s) = g_{L_0}(t,s) - \ell_{1,1}(t) g_{L_0}((1,0),s)$$

$$g_{L_0}((1,0),s) = [\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ \dots\ ]$$

$$\text{and} \quad g_{L_1}(t,s) = [\ 0,\ t_x - 1,\ t_y,\ t_x^2 - 1,\ t_x t_y,\ t_y^2,\ \dots\ ]$$

With $\theta_2 = (0,1)$, select $\ell_{2,2}(t) = (t_x - 1)/(-1) = 1 - t_x$, then

$$g_{L_2}(t,s) = g_{L_1}(t,s) - \ell_{2,2}(t) g_{L_1}((0,1),s)$$

$$\begin{aligned} g_{L_1}((0,1),s) &= [\ 0,\ -1,\ 1,\ -1,\ 0,\ 1,\ \dots\ ] \\ \text{and}\quad g_{L_2}(t,s) &= [\ 0,\ 0,\ t_x+t_y-1,\ t_x^2-t_x,\ t_xt_y,\ t_y^2+t_x-1,\ \dots\ ] \end{aligned}$$

With $\theta_3 = (-1,0)$, select $\ell_{3,3} = (t_x+t_y-1)/(-2) = \frac{-1}{2}((t_x+t_y-1)$, then

$$\begin{aligned} g_{L_2}((-1,0),s) &= [\ 0,\ 0,\ -2,\ 2,\ 0,\ -2,\ \dots\ ] \\ \text{and}\quad g_{L_3}(t,s) &= [\ 0,\ 0,\ 0,\ t_x^2+t_y-1,\ t_xt_y,\ t_y^2-t_y,\ \dots\ ] \end{aligned}$$

Finally, select $\ell_{4,4}(t) = \frac{-1}{2}(t_x^2+t_y-1)$, which results in a Newton Basis:

$$\left\{1, 1-t_x, \frac{-1}{2}(t_x+t_y-1), \frac{-1}{2}(t_x^2+t_y-1)\right\}.$$

If we continue the process once more, we have

$$\begin{aligned} g_{L_3}((0,-1),s) &= [\ 0,\ 0,\ 0,\ -2,\ 0,\ 2,\ \dots\ ] \\ \text{and}\quad g_{L_4}(t,s) &= [\ 0,\ 0,\ 0,\ 0,\ t_xt_y,\ t_x^2+t_y^2-1,\ \dots\ ] \end{aligned}$$

This gives us the set

$$\{t_xt_y, t_x^2+t_y^2-1\}$$

which generates $I(\Theta)$.

Performing the back substitution from (7.1), we get the Lagrange Basis of

$$\left\{\frac{1}{2}(t_x^2+t_x), \frac{-1}{2}(t_x^2-t_y-1), \frac{1}{2}(t_x^2-t_x), \frac{-1}{2}(t_x^2+t_y-1)\right\}.$$

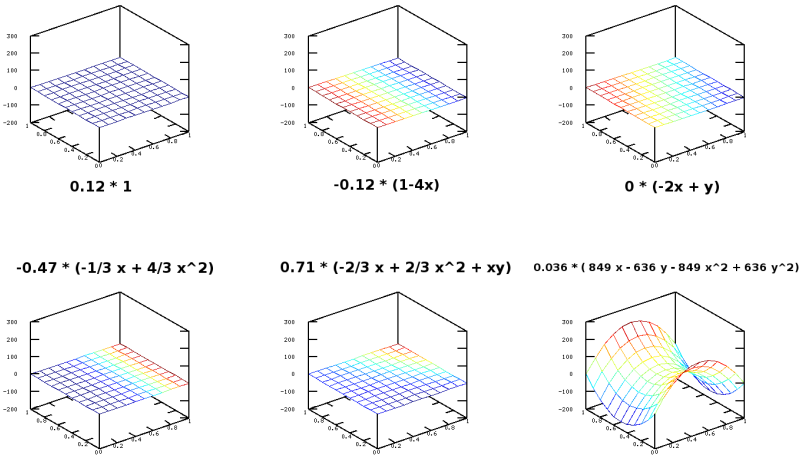
The choice of $\ell_{j,j}(t)$ in this example was made for illustration of the algorithm and is a very poor method for choosing the basis function as shown in the next section.

7.5.2 A Second Example

This second example shows that taking the approach used in the previous example, one basis function dominates the interpolant.

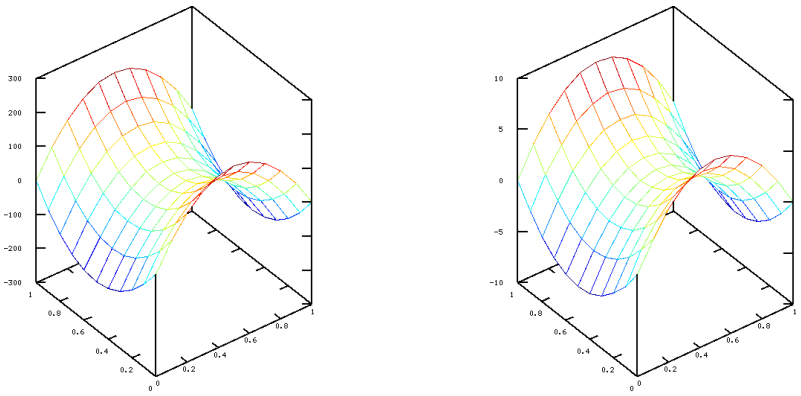
As our data, choose the points (0,0) (0,1) (1,0) (1,1) (0.25,0.5) (0.76,0.58), where the last point was chosen to be near variety through other five points. For this example, we set the z-value to interpolate as $z = \sin(x)\sin(y)$.

Choosing basis functions as described in the previous section we get the following basis functions, together with the weights that are associated with the interpolant:



In the figure, the number after the * symbol is the basis function, while the number in front of the * symbol is coefficient required to interpolate that data; the images are of the unscaled basis functions.

The result is that the interpolant looks like single basis function:



Left: $\ell_{6,6}$, Right: Interpolant.

7.5.3 The Least

The Least, which was concerned about correctness and computational efficiency, spends a lot of effort on the "pick θ_i" step; for the approach here, this step is less important because more information is retained during the computation, allowing us to address the problems the Least was avoiding in a different manner.

More important is the "select $\ell_{i,i}$" step. While XBias does not build a Newton basis, the basis it does build is equivalent to the above algorithm, where in the "select $\ell_{i,i}$" step, it choose the next non-zero g_α as the next basis function. But note the requirement that $\ell_{i,i}(\theta_i) = 1$. If the next $g_\alpha(\theta_i)$ is very small, then c_α will be huge, and this one polynomial will dominate the interpolant (away from the θ's). This is what we're seeing when XBias blows up.

The Least, on the other hand, chooses a weighted average of the non-zero g_α's of lowest degree. As long as one of the g_α's used in the average is not small at θ_i, then the c_α's will be reasonable. This "as long as" requirement can fail, however, in particular when interpolating $\#\Pi_k$ data points (in which case there is a single g_α) or if all the g_α's intersect at a single point. This latter problem is somewhat rare, although it is easy to construct cases where it happens when interpolating $\#\Pi_k - 1$ data points (i.e., you just need two g_α's to intersect). In these cases, it would be better to increase the degree of the polynomials being used.

Chapter 8

Higher Dimensions

8.1 Higher Dimensional Bézier Simplices

- Generalized Bernstein Polynomials k variate Bernstein polynomials of degree d:

$$B^d_{\vec{i}}(b_1, \ldots, b_k) = \binom{d}{\vec{i}} b_1^{i_0} b_1^{i_1} \ldots b_k^{i_k}$$

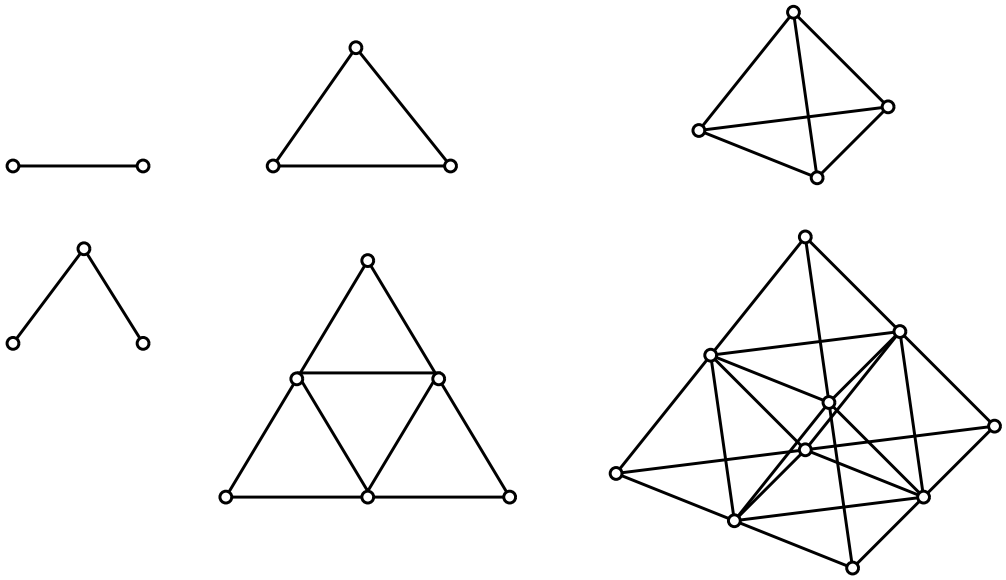
where

$$\binom{d}{\vec{i}} = \frac{d!}{i_0! i_1! \ldots i_k!}$$

$$\vec{i} = (i_0, \ldots, i_k),\ i_j \geq 0, \quad |\vec{i}| = \sum_j i_j = d$$

- Generalizing the dimension.

  We can use the generalized Bernstein polynomials to construct "surfaces" of higher dimension. Instead of choosing a domain simplex of dimension 2, we choose one of dimension d. Simplices of dimensions 1, 2, and 3 are drawable, as are their control nets. What appears below is an example of these three cases:



In the top row are three domains. In the bottom row are quadratic control nets. Evaluation, regardless of dimension, is done with a de Casteljau algorithm similar to the one for surfaces.

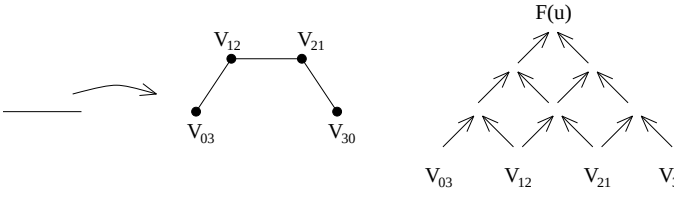
- Bézier Simplices Any polynomial function can be written as

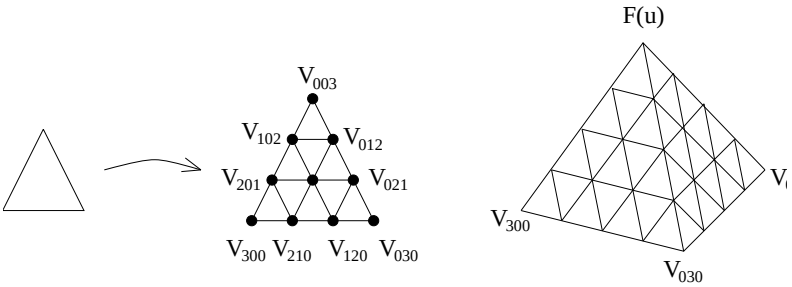
$$Q(u) = \sum_{\vec{i}} V_{\vec{i}} B^d_{\vec{i}}(b_0, \ldots, b_k)$$

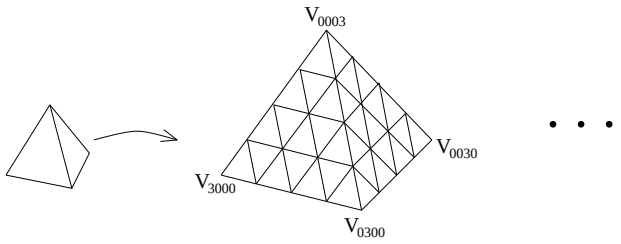
where

  - $V_{\vec{i}}$ are control points,
  - $b_0, \ldots, b_k$ are barycentric coordinates of u relative to domain simplex

- Example Simplices



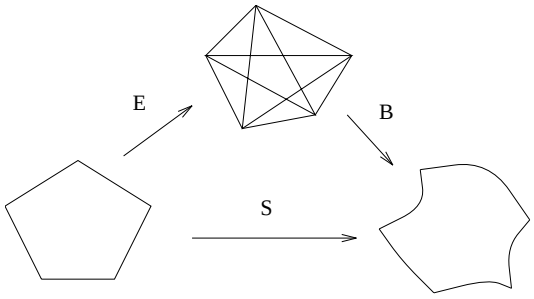




8.2 S-patches

1. So far we have looked at three sided patches and four sided patches. Suppose we want a patch with an arbitrary number of sides. How do we construct such a patch? S-patches (or Simplex-patches) provide one such construction.

2. The basic idea behind S-patches is to map our polygonal domain into a high dimensional simplex, and then map back to our 3 space with a high dimensional Bézier map:



   Thus, our S-patch S is given by the composition of E and B:

$$S = B \circ E$$

   The question is: how do we construct E?

3. Assume our domain polygon is given by $p_1, \ldots, p_n$. We first construct the functions $\alpha_i(p)$ as

$$\alpha_i(p) = \frac{\triangle p p_i p_{i+1}}{P},$$

   where P is chosen so that $\alpha_i(p_{i+2}) = 1$. Note that $\alpha_i(e) = 0$ for all e on the segment $\overline{p_i p_{i+1}}$.

   Next, let

$$\Pi(p) = \alpha_1(p)\alpha_2(p)\ldots\alpha_n(p)$$

   and let

$$\pi_i(p) = \frac{\Pi(p)}{\alpha_{i-1}(p)\alpha_i(p)}.$$

Finally, let

$$\ell_i(p) = \frac{\pi_i(p)}{\pi_1(p) + \ldots + \pi_n(p)}.$$

Note the following:

- $\sum \ell_i(p) = 1$ by construction
- $\alpha_i(p) \geq 0$ for p in our polygon, and therefore $\ell_i(p) \geq 0$ for p in our polygon.

We now use the ℓ to form our map from the regular n-gon to our $n-1$ simplex:

$$L(p) = \ell_1(p)v_1 + \ldots + \ell_n(p)v_n$$

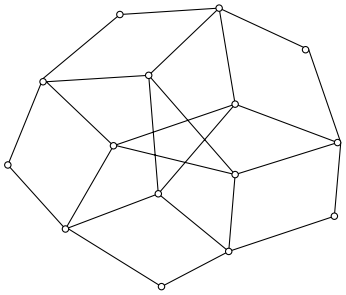
Note that

- $L(p_i) = v_i$ since $\alpha_{i-1}(v_i) = \alpha_i(v_i) = 0$
- For e_i is on the edge p_ip_{i+1}, we have $L(e_i) = e_i^*$, for some e_i^* on the edge v_iv_{i+1}.

4. Notes on S-patches

   - S-patches are rational polynomial functions for degree $(n-2)m$, where n is the number of sides of the domain and m is the degree of B.
   - The boundaries of S-patches are polynomial curves.
   - Three sided S-patches *are* triangular Bézier patches.
   - Four sided S-patches generalize $n \times n$ tensor product Bézier patches. Basically, four sided S-patches have two interior control points for every one interior control point of a tensor product patch. If we make each pair of point be the same point, then the S-patch becomes a tensor product Bézier patch.
   - There is a de Casteljau evaluation algorithm.

5. Example control net: degree 2, five sided



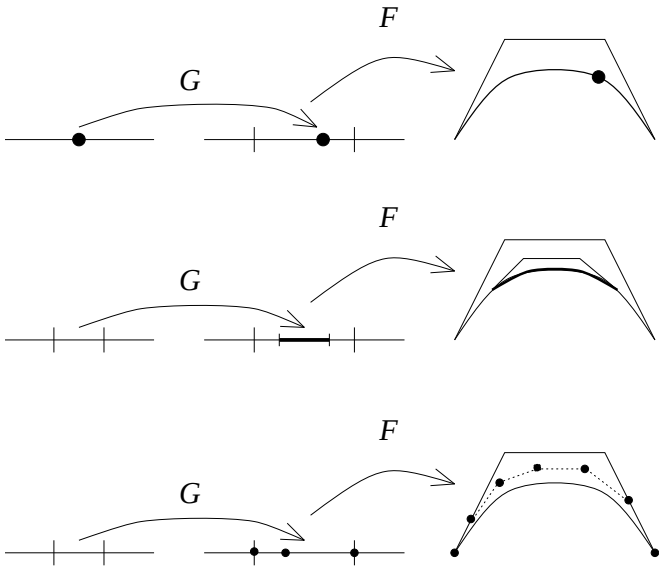
8.2.1 References

S-patches: A Class of Representations for Multi-Sided Surface Patches, Tony DeRose and Charles Loop, Technical Report 88-05-02, Department of Computer Science, FR-35 University of Washington, Seattle, WA, 98195, May 1988.

8.3 Polynomial Composition

1. Problem Statement
   - Given: Two polynomials F and G in Bézier form.
   - Find: $F \circ G$ in Bézier form.
   - Note: Domain of F better be range of G.
   - Variations for tensor products, rationals, and B-splines
2. Applications
   - Evaluation: Let G be the constant polynomial
   - Subdivision: Let G be a linear polynomial parameterized over the interval of interest
   - Polynomial Reparametrization
   - Freeform deformation
   - Triangular $\Leftrightarrow$ tensor product forms
   - Representation of trimmed tensor product as an S-patch

   Note: some of these may require rational functions
3. Evaluation / Subdivision / Reparametrization



4. Product of Bernsteins
   - Define $\vec{\imath} + \vec{\jmath} = (i_0 + j_0, \ldots, i_k + j_k)$.

- Let $I = (\vec{i}_1, \ldots, \vec{i}_m)$ and $|I| = \sum_j \vec{i}_j$.
- Let $B_I(u) = B_{\vec{i}_1}(u) \cdot \ldots B_{\vec{i}_m}(u)$
- Then

$$B_I(u) = C(I) B_{|I|}(u)$$

where

$$C(I) = \frac{\binom{|\vec{i}_1|}{\vec{i}_1} \ldots \binom{|\vec{i}_m|}{\vec{i}_m}}{\binom{||I||}{|I|}}$$

5. Polynomial Composition

- Given: $F : Y \to Z$ and $G : X \to Y$ where

$$F(u) = \sum_{\vec{i},\ |\vec{i}|=m} F_{\vec{i}} B^m_{\vec{i}}(u)$$

(relative to a domain simplex in Y) and

$$G(u) = \sum_{\vec{i},\ |\vec{i}|=\ell} G_{\vec{i}} B^\ell_{\vec{i}}(u)$$

(relative to a domain simplex in X)

- Find: $H = F \circ G$ relative to G's domain simplex.
- Note: Should be more careful with definition of $\vec{i}$

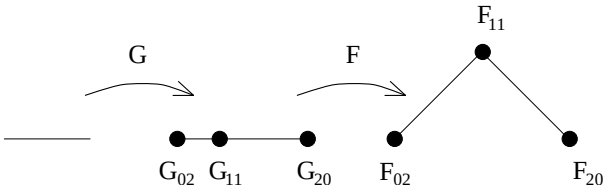
6. Derivation

$$\begin{aligned}
H(t) &= F(G(t)) \\
&= f(G(t), \ldots, G(t)) \\
&= f(\sum_{\vec{i}_1} G_{\vec{i}_1} B_{\vec{i}_1}(t), \ldots, \sum_{\vec{i}_m} G_{\vec{i}_m} B_{\vec{i}_m}(t)) \\
&= \sum_{\vec{i}_1, \ldots, \vec{i}_m} f(G_{\vec{i}_1}, \ldots, G_{\vec{i}_m}) B_{\vec{i}_1}(t) \cdot \ldots \cdot B_{\vec{i}_m}(t) \\
&= \sum_I f(G_I) B_I(t), \qquad f(G_I) = f(G_{\vec{i}_1}, \ldots, G_{\vec{i}_m}) \\
&= \sum_I f(G_I) C(I) B_{|I|}(t)
\end{aligned}$$

We can extract the control points of H:

$$H_{\vec{i}} = \sum_{|I|=\vec{i}} C(I) f(G_I)$$

7. Example



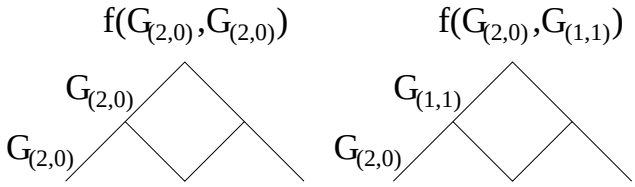
$$
\begin{aligned}
H_{(0,4)} &= f(G_{(0,2)}, G_{(0,2)}) \\
H_{(1,3)} &= f(G_{(0,2)}, G_{(1,1)}) \\
H_{(2,2)} &= \Big(f(G_{(2,0)}, G_{(0,2)}) + f(G_{(1,1)}, G_{(1,1)}) + \\
&\qquad f(G_{(0,2)}, G_{(2,0)})\Big)/3 \\
H_{(3,1)} &= f(G_{(2,0)}, G_{(1,1)}) \\
H_{(4,0)} &= f(G_{(2,0)}, G_{(2,0)})
\end{aligned}
$$

8. Implementation

   - Naive implementation very inefficient since, for example, we evaluate at all permutations of each I:

     $$f(G_{(2,0)}, G_{(0,2)}) = f(G_{(0,2)}, G_{(2,0)})$$

   - Improve by only evaluating at I where $\vec{\imath}_j <= \vec{\imath}_{j+1}$
     (weight by number of permutations)
   - This gives roughly $m!$ speedup (m is degree of F).
   - Can further improve if reuse partial evaluations of blossom.



9. Code

```
RecComp(F,G,H,n,m⃗,s⃗,c,μ)
if n = F.degree then
    H.cp_s⃗ += F.cp_0^n
```

```
else
    for all i⃗ ≥ m⃗ in increasing order
            // Eval computes F^[n+1] from F^[n]
        EvalBlossomArg(F,n+1,G.cp_i⃗)
        if i⃗ = m⃗ then μ' = μ + 1else μ' = 1
        RecComp(F,G,H,n+1,i⃗,s⃗ + i⃗, c * (|i⃗| choose i⃗)/μ', μ')
    endfor
endif
```

10. Generalized Composition

    - Compose a blossom f with a set of polynomials:

      $$f \circ G^I(u) = f(G^1(u), \ldots, G^m(u))$$

    - If we let

      $$G_I = (G^1_{\vec{i}_1}, \ldots, G^m_{\vec{i}_m}),$$

      then control point formula is

      $$H_{\vec{i}} = \sum_{\vec{i}} f(G_I) C(I)$$

    - Note: need to be careful about how we define I.

11. Degree Raising Let $L^r(t)$ be the degree r representation of the linear function.

    Then we degree raise F by looking at

    $$f\Big(L^2(t), L^1(t), L^1(t), \ldots, L^1(t)\Big)$$

12. Implementation

    - The code is basically the same, but for loop changes.
    - Thus, we can reuse partial blossom evaluations, but can't use symmetry as effectively.
    - In the case of degree raising, we can derive standard formula from the composition formula.

8.3.1 References:

Functional Composition Algorithms via Blossoming, by Tony DeRose, Ronald Goldman, Hans Hagen, and Stephen Mann, in TOG, Vol 12, No 2, April 1993

`http://www.cgl.uwaterloo.ca/~smann/Papers/tog93.ps`

An Optimal Algorithm for Expanding the Composition of Polynomials, by Wayne Liu and Stephen Mann, in TOG, April 1997.

8.4 A-patches

8.4.1 Implicit Surfaces

1. Given a function $f(x, y, z)$ over $\mathcal{R}^3$, an implicit surface is the set of all x, y, z such that $f(x, y, z) = 0$.

   Usually, we have more information. The zero set will normally divide space into two halves:

   $$\begin{aligned} f(x,y,z) &< 0 \quad \text{inside} \\ f(x,y,z) &= 0 \quad \text{on} \\ f(x,y,z) &> 0 \quad \text{outside} \end{aligned}$$

   If f is a polynomial function of x, y, z, then f is said to be algebraic.

   Similarly we can define an implicit curve $f(x, y)$ over $\mathcal{R}^{\in}$ as $f(x, y) = 0$ with a similar inside/outside test.

2. Example:

   $$f(x, y, z) = x^2 + y^2 + z^2 - 1$$

   The set of x, y, z such that $f(x, y, z) = 0$ is the sphere. If f is less than zero, then the point is inside the sphere. If f is greater than zero, then the point is outside the sphere.

3. Advantages of implicit (algebraic) surfaces:

   (a) Inside-outside test

   (b) Ray tracing becomes root finding

   (c) Offsets of algebraic surfaces are algebraic surfaces
   This is useful for CNC machining.

4. Disadvantages of implicit surfaces

   (a) Tesselation is hard

   (b) Multiple sheets, multiple zeros, degeneracies

   - hard to control, use in modeling system
   - rendering is even more difficulty

8.4.2 A-patches

1. Ideas:

   (a) Control algebraic in small region

(b) Make a patch inside a triangle/tetrahedron with desired properties inside the triangle/tetrahedron

2. Bernstein-Bézier form:

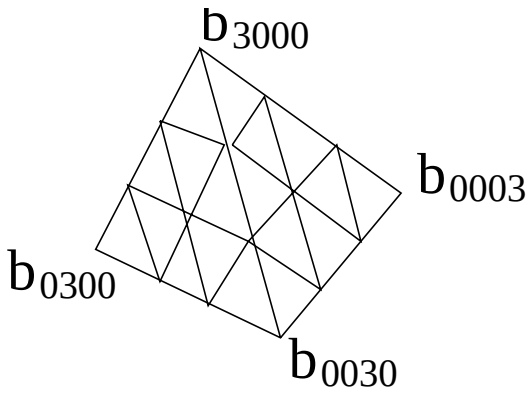
   Given a tetrahedron $V = [p_0p_1p_2p_3]$, let $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ be the barycentric coordinates of a point p. Normally we will want $p \in V$.

   Then define our surface to be

$$F(p) = \sum_{|\vec{i}|=n} b_{\vec{i}} B^n_{\vec{i}}(p).$$

   Our surface patch will be all $p \in V$ such that $F(p) = 0$.

   Pictorially, we can think of control "points" (scalar values) as being distributed uniformly through V:



   Similarly, over a triangle $T = [p_0p_1p_2]$, we can express a point P in its barycentric coordinates relative to T and use the bivariate Bernstein polynomials to express our curve in algebraic form over the triangle.

3. A few simple, obvious facts

   (a) If all $b_{\vec{i}} > 0$, then $F(p) > 0$ for all $p \in V$. Similarly for $b_{\vec{i}} < 0$.

   So to have a non-void surface patch, a necessary (but not sufficient) condition is that some $b_{\vec{i}}$ are positive and some are negative.

   (b) If $b_{ne_i} = 0$, then $F(p_i) = 0$ and the surface passes through the corner of the tetrahedron.

   (c) Along a face of V, only the control points have non-zero weight. Thus, C^0 continuity between patches in adjacent tetrahedron is easy: Just have the same boundary control points.

4. A few useful, not-so-obvious facts

(a)

$$b_{(n-1)e_i+e_j} = b_{ne_i} + \frac{1}{n} < (p_j - p_i), \nabla f(p_i) >$$

where

$$\nabla f(p) = \left[\frac{\partial f(p)}{\partial x}, \frac{\partial f(p)}{\partial y}, \frac{\partial f(p)}{\partial z} \right].$$

This will be used to construct a patch that interpolates both position and normal at a corner of the tetrahedron.

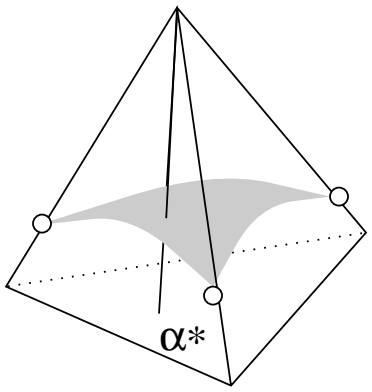
(b) Let f and g be two polynomials defined on tetrahedra $[p_0p_1p_2p_3]$ and $[p'_0p_1p_2p_3]$ respectively (i.e., they share the face $p_1p_2p_3$).
Then f and g are C^1 at the common face iff they are C^0 there and

$$g_{1i_1i_2i_3} = \beta_0 f_{1i_1i_2i_3} + \beta_1 f_{0i_1i_2i_3+0100} + \beta_2 f_{0i_1i_2i_3+0010} + \beta_3 f_{0i_1i_2i_3+0001},$$

where β are the barycentric coordinates of p'_0 relative to $[p_0p_1p_2p_3]$.
This follows from blossoming.

5. Three-sided patch

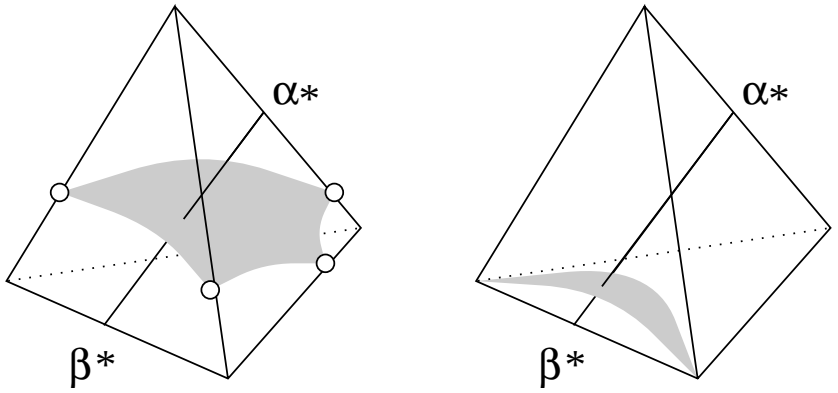
If any open line segment $(p_j, \alpha*)$ with $\alpha*$ on the face opposite p_j intersects S_F at most once (including multiplicities) then we call S_F a three-sided j-patch.



Note: with this definition, the surface may leave and re-enter the tetrahedron (and thus may have disjoint components within the tetrahedron).

6. Four-sided patch

If any open line segment $(\alpha*, \beta*)$ with $\alpha*$ on segment p_ip_j and $\beta*$ on segment p_kp_l intersects S_F at most once (counting multiplicites) then we call S_F a four-sided ij-kl-patch.



Again, the surface may leave and re-enter the tetrahedron, and we may have degeneracies. In fact, we will be using the degeneracy shown on the right.

7. A variety of lemmas are needed about non-singularities, but the following theorems are the results we need to construct our patch network:

   Theorem: If there exists k (with $0 \leq k < n$) such that

   $$\begin{aligned} b_{\vec{i}} &\geq 0 \quad i_j = 0, \ldots, k-1, \\ b_{\vec{i}} &\leq 0 \quad i_j = k+1, \ldots, n, \end{aligned}$$

   and $\sum_{i_j=0} b_{\vec{i}} > 0$ and $\sum_{i_j=m} b_{\vec{i}} < 0$ for at least one m (with $k < m \leq n$) then S_F is a three-sided patch.

   What this says is that the coefficients near one vertex are negative and those near the opposite face are positive. Plus, there is one layer in between where the coefficients can be either positive or negative.

   Note also that we can "flip the sign" and have the positive coefficients near a vertex and the negative ones near the opposite face.

   A similar theorem can be stated to get four-sided patches (essentially, layer the tetrahedral array from one edge to the opposite edge, and have one "half" be positive and the other negative, with a layer in between that can be either).

   Both theorems provide sufficent but not necessary conditions to get three- and four-sided patches.

8. If in the above theorem, we have $0 < k < n$ (i.e., we don't allow k to be equal to 0), then it's straightforward to prove that the A-patch is singled sheeted in the triangle/tetrahedron, and that there is exactly one zero on line segments from the corner of one sign to the edge/face of the opposite sign (or in the case of four-sided patches, along line segments between points on the edges of opposite sign).

   We will begin by considering an A-patch triangle. We can perform 2-1 subdivision from the point of one sign to any point on the edge of opposite sign. Since 2-1 subdivision

can be performed by doing curve subdivision one each row of the patch, we see that the control points along the split edge will be a sequence of positive values followed by a sequence of negative values. By the variation diminishing property, this implies that there is exactly one zero along this split line, which is the desired result.

For three-sided patches, the same proof applies, only we need to use 3-1 subdivision of the tetrahedron. This 3-1 subdivision is readily seen to be a generalization of 2-1 subdivision of triangles, and can be performed by doing "triangle" subdivision on each layer of the tetrahedron. Our result immediately follows.

The proof for four-sided patches is similar and left as an exercise for the reader.

As a side note, to find the point on the curve/surface, we need to doing a numerical search along the line segment between the vertex and a point on the opposite side. After doing the 2-1 split (or 3-1 split), we can extract the edge control points and perform this search in a univariate setting, rather than bi- or tri-variate.

9. Note that if we have a three-patch or a four-patch that is a single piece, then it is reasonable to tesselate them into triangles. For a three-patch, we just place a triangular grid on the $\alpha*$ side, and root find along each $(p_j, \alpha*)$ segment to find the point on the surface.

   For a four-sided patch, we sample each edge and create a tensor product "triangulation" (with degeneracies at each end).

   If the surface passes through the face of the tetrahedron, then tesselation is more difficult, but in this A-patch construction, the problem is readily resolved as shown in the next section.

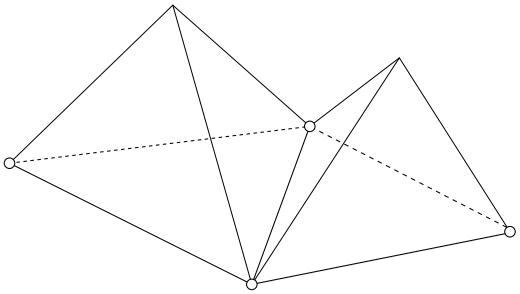
8.4.3 Simplicial Hulls

1. Eventually, we will build a C^1 piecewise algebraic surface with A-patches that interpolate the vertices of a triangular polyhedron.

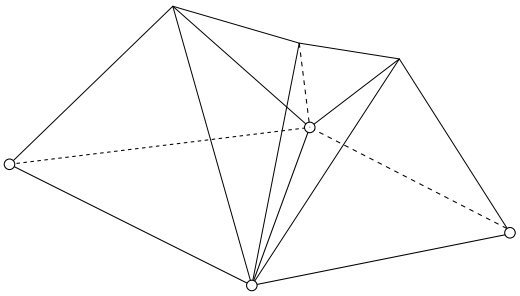
   Since each A-patch is contained within a tetrahedron, as a first step, given a triangular polyhedron, we need to build a "tetrahedralization" of the space around the polyhedron into which we will build our C^1 surface.

   Based on the normals, we can determine (roughly) where the surface will go.

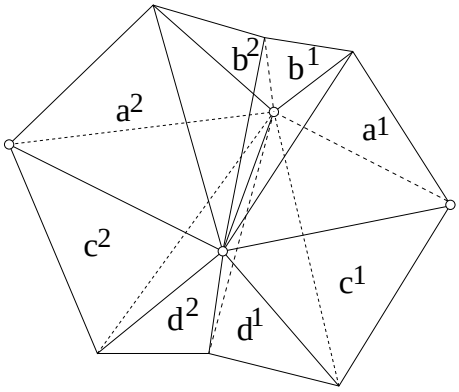
2. Above and/or below each face, we will put a tetrahedron.



Then use two more polyhedron to fill in the gaps.

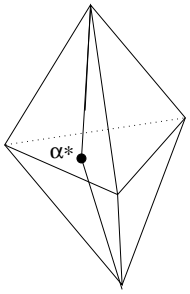


In some configurations of the normals, we will get twice as many tetrahedrons:



3. See the paper for details; in particular, where to place the extra vertices of the tetrahedron. And degenerate cases for their construction. Note in particular the various cases based on the normals at an edge. An edge may be convex (positive or negative) or non-convex, or possible zero-convex. Faces may also be convex, etc.

4. As an extra note, the surface may be on both sides of a data triangle. However, the surface will lie within a pair of tetrahedron that share the same $\alpha*$ face. And if we look at the two line segments from two vertices to any $\alpha*$ point, this pair of line segments will intersect the surface in exactly one point:



Again, tesselation will be easy.

8.4.4 C^1 Construction Ideas

1. We will not present the complete construction here as it is fairly complicated (volume vs surface).

   The idea is to construct a piecewise algebraic cubic, working face pair at a time.

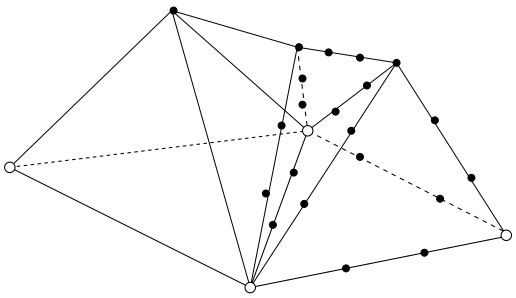
   Note that for each pair of faces in the polyhedron, there will be between 4 to 8 tetrahedron in the simplicial hull.

2. Based on the normals at the four vertices in a face pair, there will be three major cases, with additional subcases and special cases. These cases correspond to combinations of convex and non-convex faces meeting one another, with the subcases handling positive and negative convexity. Co-planar faces are a special case.

   The general idea of the construction is as follows:

   (a) Set/restrict face vertice of adjacent polyhedron to be the same. This gives C^0 continuity.

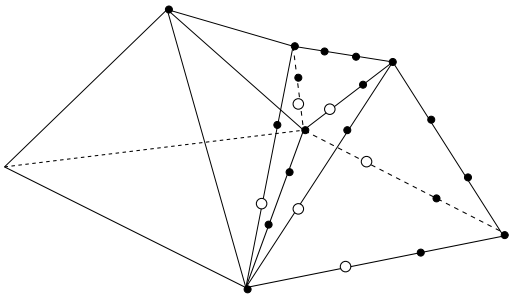
   (b) Set to 0 the corner vertices corresponding to the vertices of polyhedron.

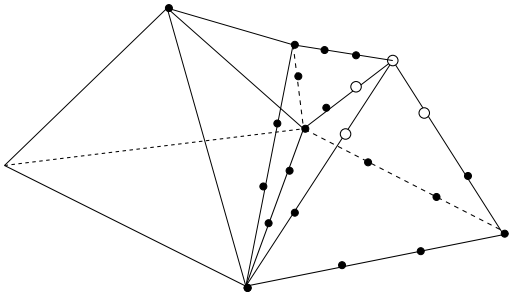
   This will interpolate our data points.

   (c) Use the normal conditions to set (most of) the boundary control points. Since our corner points are 0, each point will be set via a formula like

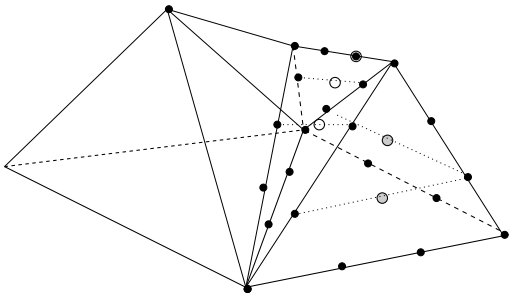
   $$b =< (p_i - p_j), n_j > /3$$



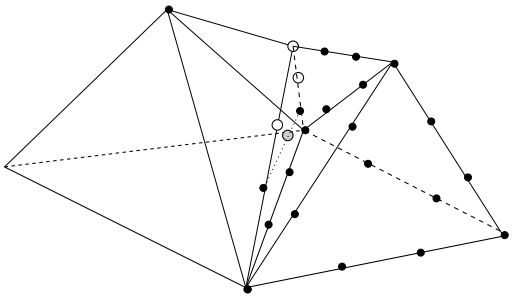
(d) The tops of the face pyramids are free parameters.



(e) The C^1 conditions impose a relationship between CP's in A and B, so treat two of the relevant CP's as set with two to be determined later.



(f) The C^1 conditions impose a relationship between CP's in B and B', which sets more CP's, with one to be determined later.



The constructions continues, with further steps. Basically, CP's are set to get continuity between each pair of tetrahedron. And as noted earlier, there are several different constructions based on the normals to the face pairs.

Some where, the vertex consistency problem must get solved, but I couldn't see it, and it isn't explicitly mentioned. However, it appears that in general they have enough degrees of freedom to solve the v.c. problem, but for co-planar faces they lose a degree of freedom and have to subdivide a tetrahedron to solve it (a.k.a. a Clough-Tocher split).

3. Note that there are a lot of degrees of freedom in this construction.

8.4.5 References:

Modeling with Cubic A-Patches, Chandrajit Bajaj, Jindon Chen, and Guoling, ACM Transactions on Graphics, Vol 14, No 2, April 1995.

8.5 Universal Splines

1. Basic idea: map spline to high dimensional space where there will be no degeneracies. Then, define blossom as intersection of osculating flats.

2. Osculating Flats. Define $\mathrm{Osc}_k F(u)$ as the space spanned by $F'(u),\ F''(u), \ldots, F^{(k)}(u)$, shifted by $F(u)$. Ie, if T is the space spanned by the first k derivatives, then $\mathrm{Osc}_k F(u) = F(u) + T$. This is known as the the kth osculating flat of F at u.

   Note the following:

   - $\mathrm{Osc}_0 F(u) = F(u)$
   - $\mathrm{Osc}_1 F(u)$ is the tangent line of F at $F(u)$.
   - $\mathrm{Osc}_2 F(u)$ is the plane passing through $F(u)$ spanning the first and second derivative vectors.

   It can be proven that the kth Bézier control point of a non-degenerate polynomial F can be constructed by intersecting Osculating flats:

$$P_k = \mathrm{Osc}_k F(0) \cap \mathrm{Osc}_{n-k} F(1).$$

   Further, if the expression

$$f(u_1^{<\mu_1>}, \ldots, u_n^{<\mu_h>}) = \bigcap_{i=1}^{h} \mathrm{Osc}_{n-\mu_i} F(u_i)$$

   is always well defined. It should be clear that f is symmetric and that $f(u^{<n>}) = F(u)$. It can also be shown that f is symmetric. By the blossoming principle, f must be the blossom, and thus we have an alternate definition for the blossom of F.

3. This construction for the blossom fails if F is not a general polynomial. I.e., if F is degree n but lies in a space of dimension strictly less than n, then the osculating flat intersections fail to have the desired properties.

   Note, however, that Universal splines were constructed so that the control points of each segment are in genearal position. Seidel exploited this fact to use the osculating flats of the Universal Spline to for the blossom of a geometrically continuous spline.

8.5.1 References:

Polar Forms for Geometrically Continuous Spline Curves of Arbitrary Degree, Hans-Peter Seidel, ACM Transcations on Graphics, Vol 12, No 1, January 1993.

Chapter 9

Wavelets

We shall begin with an intuitive development of wavelets taken from Stollnitz et al.

9.1 Intuition

1. Take a discrete signal (image).

   Decompose it into a sequence of frequencies.

   Use compact support basis functions rather than infinite support sin and cos.

   Construct a vector space using "unit pixels" as basis elements (piecewise constant functions).

   Average to get low frequencies.

   Construct "detail functions" to recover detail.

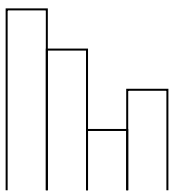
   Unit pixel bases form nested sequence of vector spaces, with the detail functions (wavelets) being the difference between these spaces.

9.2 1-D Haar

1. Suppose we have the coefficients

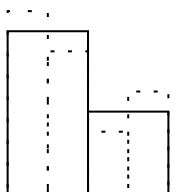
   [9 7 3 5]

   where we think of these coefficients as 1-D pixel values



The simplest wavelet transform averages adjacent pixels, leaving

$$[8\ 4]$$



Clearly, we have lost information. Note that 9 and 7 are 8 plus or minus 1, and 3 and 5 are 4 minus or plus one. These are our detail coefficients:

$$[1\ -1]$$

We can repeat this process, giving us the sequence

Resolution	Averages	Details
4	[9 7 3 5]	
2	[8 4]	[1 -1]
1	[6]	[2]

2. The Wavelet Transform (wavelet decomposition) maps from [9 7 3 5] to [6 2 1 –1] (i.e., the final scale and all the details).

   The process is called a Filter Bank.

   No data is gained or loss; we just have a different representation. In general, we expect many detail coefficients to be small. Truncating these to 0 gives us a lossy compression technique.

3. Vector spaces.

   We can think of our image as a vector space.

   The pixels are the elements of the space.

   Let [0,1) be constant. Let V^0 be the space of all such functions.

   Divide the interval in half to get [0,1/2) and [1/2,1) be constant over each interval. Call this V^1.

   Continue in this fashion, creating space V^j which includes all piece-wise constant functions over intervals of size 2^-j over [0,1).

   Note that $V^0 \subset V^1 \subset V^2 \subset \ldots$

4. Basis.

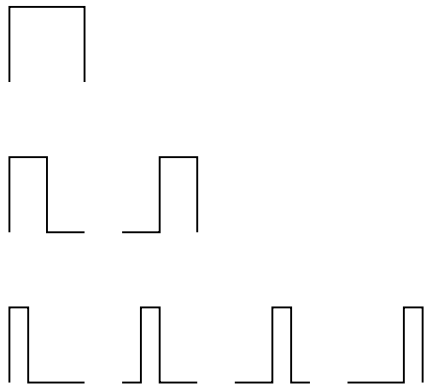
   We need a basis for each vector space.

   Basis function for V^j are called scaling functions.

   For Haar, can use

   $$\phi_i^j(x) = \phi(2^j x - i)$$

   where $\phi(x) = 1$ for $0 \leq x < 1$ and 0 otherwise.



   Support of a function refers to the region over which the function is non-zero. Note that the above basis functions have *compact support*, meaning they are non-zero over a finite region.

5. Inner Product.

   Next, we need an inner product:

   $$< f|g >= \int_0^1 f(x)g(x)dx$$

   Two vectors are said to be orthogonal if $< u|v >= 0$.

   Define a new vector space W^j that is the orthogonal complement of V^j in V^{j+1}. I.e., W^j is the set of all functions in V^{j+1} that are orthogonal to V^j.

   A basis for W^j are called wavelets. Important properties:

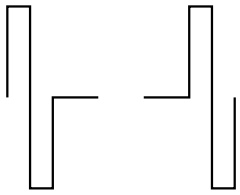
   - W^j together with V^j for V^{j+1}
   - Every ψ_i^j of W^j is orthogonal to every ϕ_i^j of V^j.

6. Haar Wavelets.

   $$\psi_i^j(x) = \psi(2^j - i)$$

   where

   $$\psi(x) = \begin{cases} 1 & \text{for } 0 \leq x < 1/2 \\ -1 & \text{for } 1/2 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$



7. Example again.

   Originally, w.r.t. V^2,

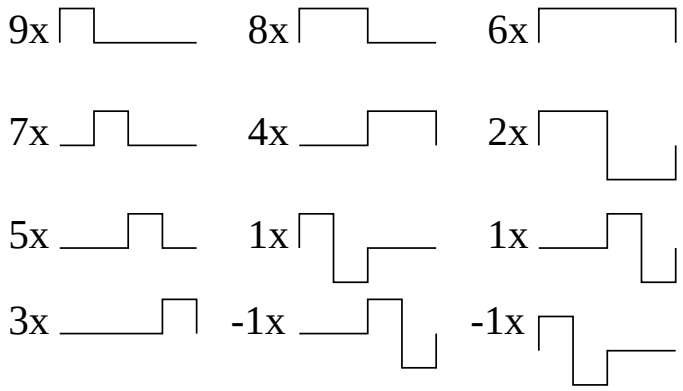
$$I(x) = 9\phi_0^2(x) + 7\phi_1^2(x) + 3\phi_2^2(x) + 5\phi_3^2(x)$$

   Rewriting w.r.t. V^1 and W^1,

$$I(x) = 8\phi_0^1(x) + 4\phi_1^1(x) + 1\psi_0^1(x) + (-1)\phi_1^1(x)$$

   Rewriting the ϕs in terms of V^0, W^0,

$$I(x) = 6\phi_0^0(x) + 2\psi_0^0(x) + 1\psi_0^1(x) + (-1)\phi_1^1(x)$$



8. Orthogonality and Normality

   Note that the basis functions in any one basis are mutually orthogonal. Such a basis is said to possess the property of orthogonality.

   A second useful property is normalization. This property requires that $< u|u >= 1$. We can modify our Haar basis to obtain normalization:

$$\begin{aligned} \phi_i^j(x) &= \sqrt{2^j}\phi(2^j x - i) \\ \psi_i^j(x) &= \sqrt{2^j}\psi(2^j x - i) \end{aligned}$$

   This changes the coefficients by a factor of $\sqrt{2^j}$. In our example, the normalized coefficients become

$$[6 \; 2 \; 1/\sqrt{2} \; -1/\sqrt{2}]$$

   .

9.3 Wavelet Compression

1. Lossless image compression – how do we represent an image using as few bits as possible?

   Pigeon hole principle tells us we've lost before we start. Have to have expectations about image before you can achieve any compression.

   Lossy compression involves two main steps: Loosing part of the data, and then performing lossless compression on what's left.

2. Can use wavelets to perform simple lossy part of compression.

   Need a goal function. For example, try to minimize the square of the L_2 error. For normalized Haar, this is easy. Let f be our image in wavelet form, let $\hat{f}$ be our wavelet approximation where we only use m of the wavelet coefficients. Order our coefficients in some order to be determined later. Then

$$\begin{aligned} ||f(x) - \hat{f}(x)||_2^2 &= < f(x) - \hat{f}(x) | f(x) - \hat{f}(x) > \\ &= \left\langle \sum_{i=m+1} c_i u_i | \sum_{j=m+1} c_j u_j \right\rangle \\ &= \sum_{i=m+1} \sum_{j=m+1} c_i c_j < u_i | u_j > \\ &= \sum_{i=m+1} c_i^2 \end{aligned}$$

   If we order our coefficients in decreasing magnitude, then error is minimized.

3. Notes/Questions:

   - Is L_2 a good metric?
     Do people see L_2 error?
     No. But it's hard to write a formula for the human visual system's error function that we want to minimize.
   - To re-compose our image, we still need to know which coefficients we keep. Either tag each coefficient with its index, or keep complete set of coefficients with lots of zero coefficients. Then apply lossless compression.

9.4 2D Haar and Image Compression

- Standard decomposition: Apply 1-D Haar to each row. Then apply 1-D Haar to each column.

  This gives us all combinations of the ϕ and ψ functions.

- Nonstandard decomposition: Apply one step of 1-D Haar to each row. Then apply one step of 1-D Haar to each column. Repeat on quadrant containing averages in both directions.

  This gives us new ϕ and ψ functions

$$\begin{aligned}
\phi\phi(x,y) &= \phi(x)\phi(y) \\
\phi\psi(x,y) &= \phi(x)\psi(y) \\
\psi\phi(x,y) &= \psi(x)\phi(y) \\
\psi\psi(x,y) &= \psi(x)\psi(y)
\end{aligned}$$

  The first of these is a scaling function, the last three are wavelet functions.

  We can now define our basis functions:

$$\begin{aligned}
\phi\phi^0_{0,0}(x,y) &= \phi\phi(x,y) \\
\phi\psi^j_{k.\ell}(x,y) &= \phi\psi(2^j x - k, 2^j y - \ell) \\
\psi\phi^j_{k.\ell}(x,y) &= \psi\phi(2^j x - k, 2^j y - \ell) \\
\psi\psi^j_{k.\ell}(x,y) &= \psi\psi(2^j x - k, 2^j y - \ell)
\end{aligned}$$

- Image compression in similar fashion, except it's expensive to sort coefficients.

9.4.1 References:

Wavelets for Computer Graphics, Eric Stollnitz and Tony DeRose and David Salesin

Index

Index