

# **NURBS-Based Adaptive Finite Element Analysis**

Toby Mitchell

PhD Student and Graduate Research Assistant  
Structural Engineering and Mechanics of Materials  
Civil and Environmental Engineering  
University of California at Berkeley

## **Introduction:**

Finite element analysis is ubiquitous in engineering design. The complex geometries of typical design make closed-form analysis impossible, and finite element analysis provides a means of computing approximate solutions, based on a set of **basis functions** that approximate the exact solution over spatially-limited domains (elements), that can be proven to approach the unknown exact solution with mesh refinement. This technique has become indispensable in automotive and aerospace engineering as well as in architecture and structural engineering, where it has enabled analysis of complex forms that could not be designed to bear loads and survive damage through any other method, at least not with the same degree of confidence afforded by FEA.

The idea of using NURBS as the basis functions, instead of the traditional Lagrange polynomials, seems to be relatively recent. Cottrell, Hughes, and Bazilevs put forth the notion in 2004, and followed with a series of papers that applied NURBS-based finite element analysis to various test problems, typically showing improved numerical performance and efficiency. These results were recently collected with extended commentary in book form. The authors argued that the ability of NURBS to exactly represent a given geometry at low order, coupled with the capability of refining the mesh to achieve higher accuracy without changing the geometry, made NURBS an ideal basis with which to build finite element codes.

## **Project Description:**

The goal of this project was twofold: I needed to teach myself how NURBS work and how to integrate them into finite element analysis for the sake of my research, and I hoped to make good on the promise of NURBS in **adaptive analysis**. Adaptive analysis refers to the process of automatically refining a finite element mesh to achieve a target level of accuracy at the minimum possible computational cost. Although adaptive analysis has clear benefits to industry in terms of potentially making analysis much less time-consuming, the practical difficulty of iteratively refining a conventional finite element mesh without butchering the geometry has effectively confined adaptive analysis to the academic realm. In particular, adaptive analysis with ordinary Lagrange-basis-function finite elements would require a direct computational link between CAD and FEA software. Since this link is typically only realized quasi-manually in practice, adaptive analysis cannot be done in industry without unreasonable overhead.

The geometrical exactness of NURBS basis functions coupled with the relative ease of refinement makes them naturally suited to making adaptive analysis more practical. No external link to the geometrical model is required with NURBS, since the CAD and coarsest FEA model are one and the same. This makes automated adaptive analysis much more feasible, and also makes such techniques as shape optimization – a true union of design and analysis – much more accessible. The prospect of making these powerful labor-, cost-, and material-saving techniques less academic and more readily available in structural design motivated this project.

I had hoped to accomplish the project in two phases. In the first, I would concentrate on building a NURBS-based finite element analysis code in MATLAB. In the second, I would focus on a straightforward two-dimensional test problem presented in [1] where the analytical solution is known, and would develop an iterative code that would compare the analysis results on a particular mesh to the exact solution and then would decide how to refine the mesh until a desired target accuracy was achieved. Ultimately the code would be modified to use an a-posteriori error estimation routine (of which many exist in the literature), eliminating the need for a known exact solution and rendering the code fully adaptive for any given problem.

### **Concept and Structure of Code:**

It is worth mentioning that although the code was written in MATLAB, the algorithms I wrote were intended to stand on their own. The code is ultimately intended to be incorporated into the popular and powerful open-source academic finite element analysis program FEAP, which my advisor and co-advisor maintain and distribute. I therefore set myself several limitations that the code needed to satisfy before I would be satisfied with the code. Specifically:

1. It had to be arbitrary-order, i.e. it had to be able to generate and analyze a NURBS mesh of any given polynomial order. Ideally, it should also be able to produce any order of derivative (up to the polynomial order).
2. It had to be general, in the sense of not relying on any assumptions about the input. Code that assumed open knot vectors, assumed uniform knot vectors, assumed no repeated internal knots, etc, was therefore disqualified.
3. It had to be reasonably efficient. In particular, no arrays could be dynamically allocated. Of course, the requirement of efficiency was balanced against the requirement of actually getting the code to work under a deadline, so I wasn't fanatical about this, just careful.
4. It had to be exportable. As much as possible was done with only for loops and if statements, and minimal use was made of the canned MATLAB functions for matrix manipulation. The code was written with an eye for future translation into heavier-duty languages.

Of course, no project works out exactly like it was planned. When I started I had done a literature search that turned up no papers on adaptive NURBS-based FEA, so I thought I had a shot at getting something publication-worthy. I later found some preprints and conference presentations showing researchers working on exactly that, which gave me some insight into the difficulty of the problem but also killed my get-published-quick scheme. More to the point, the learning curve involved in writing a NURBS-based FEA code from scratch stymied my attempts to arrive at an adaptive code. I nonetheless was able to get results which compared well with the exact solution, so the effort was still a qualified success.

Although the results thus far are sadly not new, I would argue that the particular implementation and the underlying philosophy of the code is superior to the published results I had used as references. In particular, Cottrell, Hughes, and Bazilevs rely heavily on the algorithms published in Piegl and Tiller, which, while certainly efficient, tend to obscure the underlying geometrical structure of NURBS by replacing it with rather opaque algebraic formulae. I spent quite a bit of time reading Farin, which was a much better guide to the underlying geometrical concepts (although it is chock full of typographical errors), and was able to get some good insights into algorithm design from Prautzsch, Boehm, and Paluszny. In the end, I arrived at a code design that I believe is simpler and more transparent than that presented by Hughes et al.

The main philosophical consideration guiding the code was really something I picked up from this class – that the geometry of NURBS is more fundamental than the algebra. Rather than relying on basis function formulae to construct curves, as in Piegl and Tiller, it is much more intuitive to build the recursive linear interpolation algorithms to construct the curves directly. Basis functions can then be recovered from these algorithms in exactly the same way as they were originally derived – by expanding out the recursive linear interpolation. Specifically, every time a basis function is needed, it is evaluated as a one-dimensional curve with a unit control point at the selected function and zero control points everywhere else, effectively filtering out the function that would be multiplied onto that control point in evaluating a point on the curve.

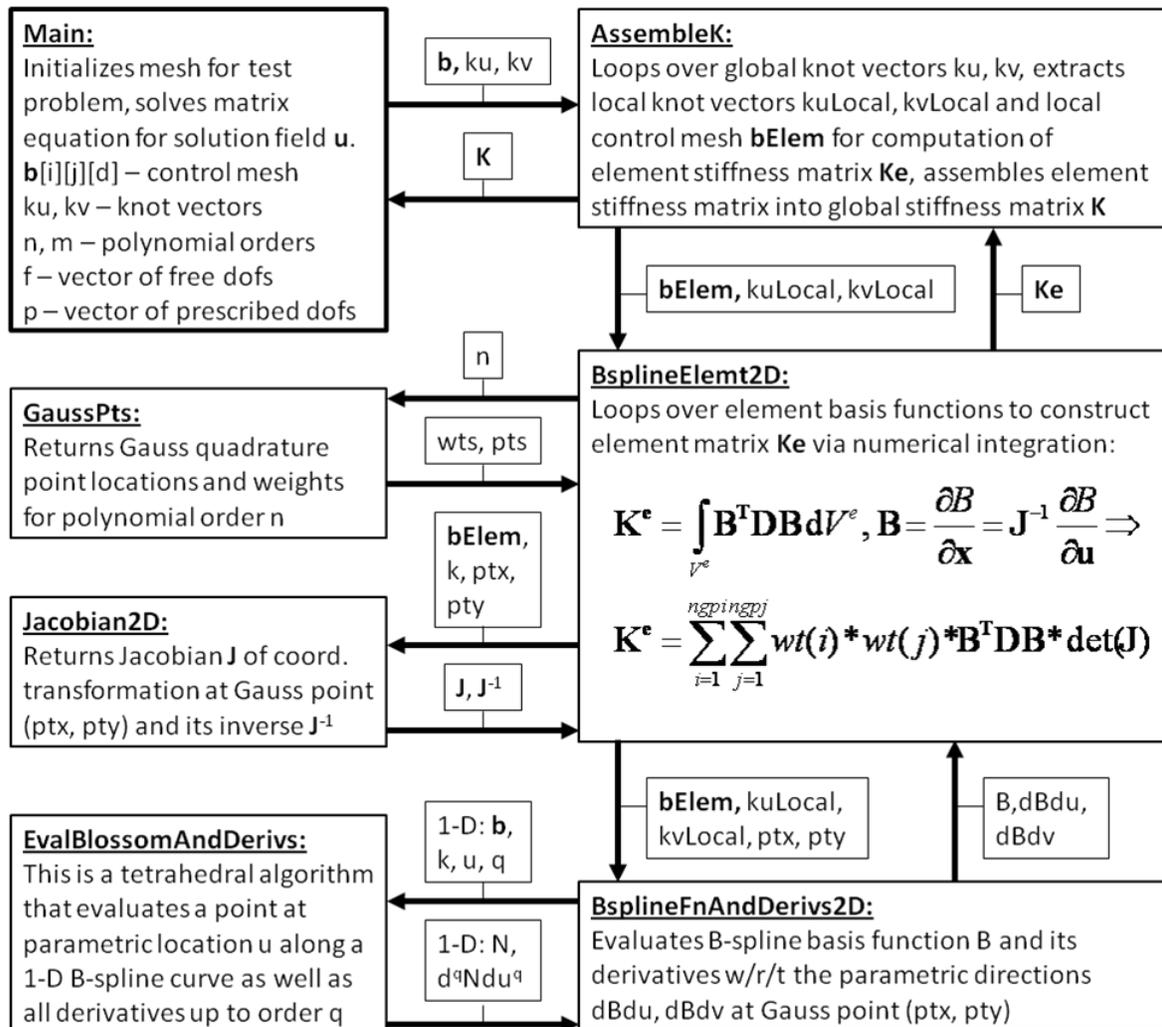
The concept of blossoming, presented in Rockwood and Chambers as well as in other references, was very useful in concisely representing the recursive linear interpolation that is the essential building block of the code. Rather than having different algebraic algorithms for each geometric operation (as in Piegl and Tiller), every separate algorithm is constructed by some combination of this one fundamental atomic unit. The only exception is order elevation, which requires a separate routine to elevate Bezier curve segments. I ended up replacing the simple blossoming routine in Rockwood and Chambers with a tetrahedral algorithm, as discussed in Prautzsch, Boehm, and Paluszny, that can take derivatives of the curve by successively differencing intermediate points of the recursive linear interpolation.

I can't say that my code is more efficient than others – at this point in its development that is impossible to expect – but it **is** more transparent and it **does** keep the geometry closer to the surface. The structure of the code is presented in **Figure 1**. In addition to this main routine, several routines (**GraphSurf**, **GraphStress**, **GraphExactStress**, **GraphStressError**) were written to display the displacement field results  $\mathbf{u}$  from the finite element solution, the corresponding stresses, the stresses of the exact solution, and the RMS error of the stresses from the finite element solution compared with the stresses from the exact solution.

Other than the general issues of presentation I've mentioned, I was able to eliminate one minor unnecessary complexity from the implementation presented in Hughes et al. The numerical integration of element stiffness matrices is typically conducted in a normalized bi-unit "parent domain" which is mapped to the physical

geometry by a Jacobian coordinate transformation. This is done so that the integration points can be pre-computed rather than being computed separately for each element, which would introduce unacceptable computational expense. However, this parent domain does not match the parameterization of the NURBS knot vectors, so in Hughes et al., the authors transform the integration points into the knot vector space. In this approach, a separate mapping is required for the derivatives of basis functions and for the Jacobian determinant used in Gauss integration, a result that differs from standard form.

Rather than do this, it is sufficient to do the opposite and simply translate and stretch each local knot vector that is collected in evaluating each element such that it fits the standard parent domain parameterization. This requires no modification of the Jacobian from its standard form, eliminating a minor but ultimately unnecessary matrix calculation. Results of both methods were found to give equivalent results to  $10^{-10}$  digits, so the additional simplicity of my method comes at no cost in accuracy.



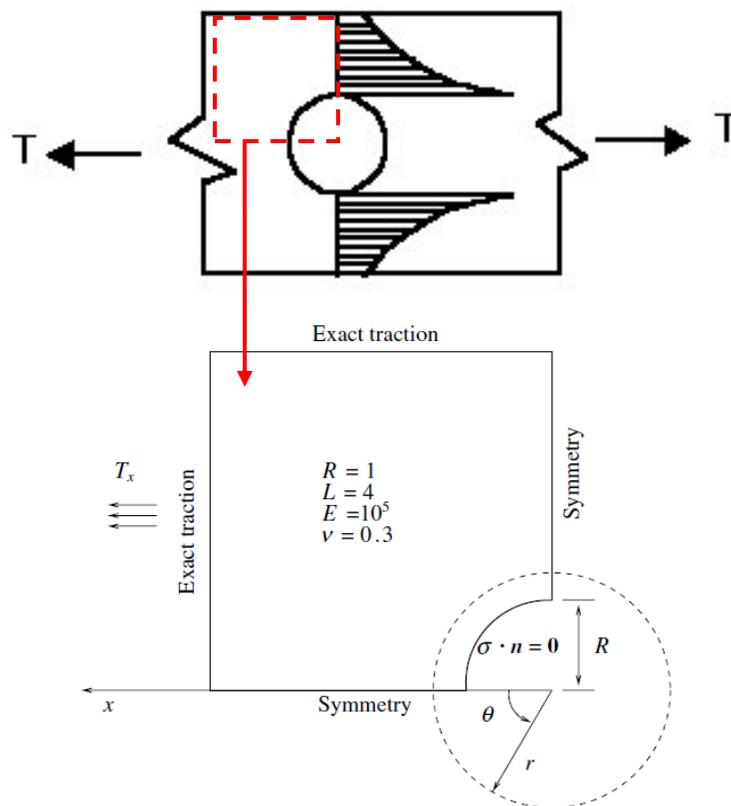
**Figure 1:** Structure of Code

One additional improvement was the use of a separate knot multiplicity vector  $r$  to deal with repeated knots. The implementation presented in Hughes et al. simply checks if two adjacent knot values are equal, indicating a degenerate knot span, and in that case does not compute an element over the degenerate span. I implemented an improvement suggested in various references, namely the use of a separate vector that indicates the multiplicity of each knot value in the knot vector. This improvement results in a modified knot vector with no repeated values, coupled with a separate knot multiplicity vector, and eliminates the need to check if the current span is degenerate, since no repeated knot values exist in the modified knot vector and all knot spans are therefore a priori non-degenerate. It also allows the knot vectors to be a vector of doubles while the knot multiplicity vector is a separate vector of ints.

It must be noted that the final code only employed B-splines. NURBS-based code is not significantly difficult, but it will have to wait to be implemented.

### The Test Problem:

The problem that the code was tested on is a classic result from mechanical engineering involving the stress concentration of a two-dimensional flat plate with a hole under tension, where the hole is small with respect to the dimensions of the plate. The smallness of the hole allows one to model the plate as effectively infinite in size.



**Figure 2.** Test problem: Segment of infinite plate with hole under tension (from [1])

This problem is commonly employed in industry to model the stress concentration in the vicinity of a small hole. The exact stresses in the plate are given by

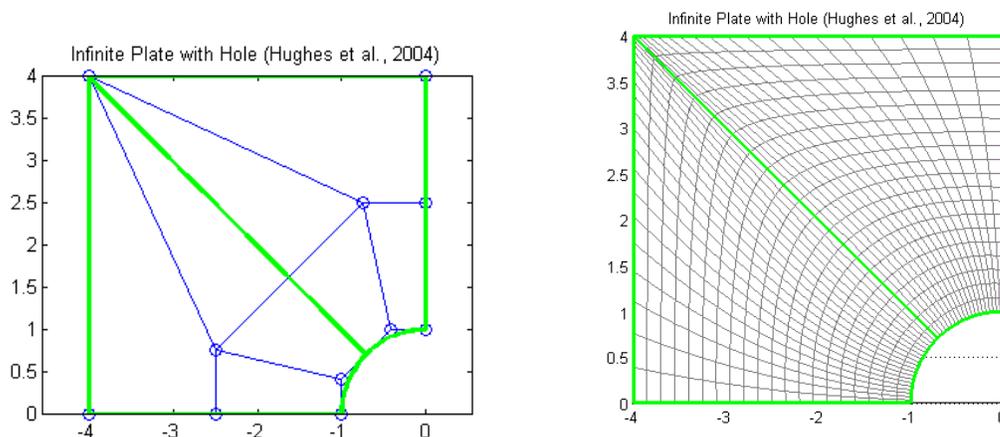
$$\sigma_{rr}(r, \theta) = \frac{T_x}{2} \left( 1 - \frac{R^2}{r^2} \right) + \frac{T_x}{2} \left( 1 - 4 \frac{R^2}{r^2} + 3 \frac{R^4}{r^4} \right) \cos 2\theta,$$

$$\sigma_{\theta\theta}(r, \theta) = \frac{T_x}{2} \left( 1 + \frac{R^2}{r^2} \right) - \frac{T_x}{2} \left( 1 + 3 \frac{R^4}{r^4} \right) \cos 2\theta,$$

$$\sigma_{r\theta}(r, \theta) = -\frac{T_x}{2} \left( 1 + 2 \frac{R^2}{r^2} - 3 \frac{R^4}{r^4} \right) \sin 2\theta,$$

where  $T_x$  is the magnitude of the applied tensile stress and  $R$  is the radius of the hole. The maximum tensile stress at the edge of the hole is three times the applied tensile stress. This is referred to as the **stress concentration factor** of this particular geometry, and design engineers must size the plate to accommodate these stress concentrations.

Since the domain of the problem is infinite but the code can only model finite domains, a trick is required in order to render the problem amenable to computational analysis. First, the problem is symmetric about the origin, so only one quadrant is modeled, with appropriate symmetry displacement boundary conditions. Second, a small subdomain of the infinite plate in the immediate vicinity of the hole is selected, and the stresses from the exact solution are applied at the edges of this subdomain (**Figure 2**). This trick enables the finite element solution to approximate the exact solution. Ideally, the finite element solution will converge to the exact solution as the mesh is refined.



**Figure 3:** Two-element quadratic NURBS mesh. Blue = control net, Green = elements.

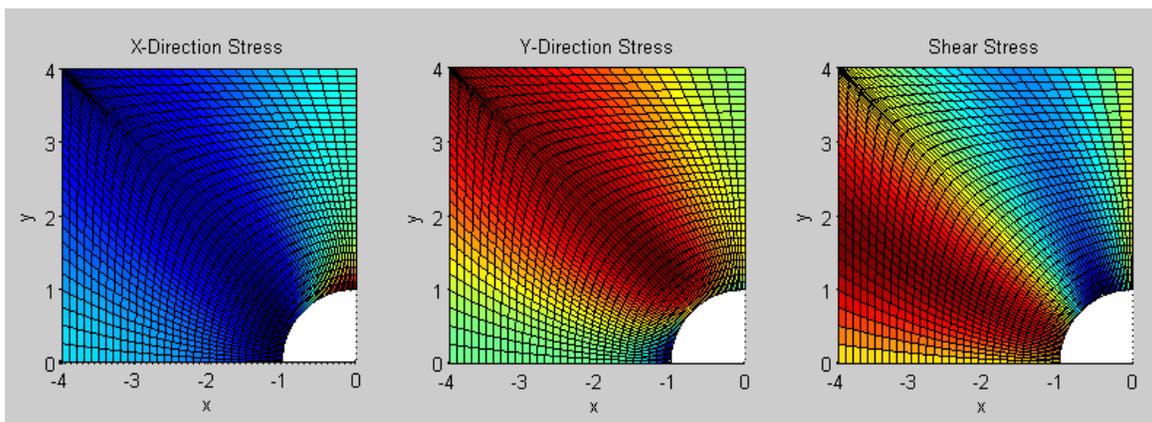
This particular geometry can be described exactly with one two-element quadratic NURBS patch that is  $C^1$ -continuous across the element boundary (**Figure 3**). This

particular approach results in a singular point at the upper left corner, where two control points must overlap to force the patch to have straight sides. This geometry can also be modeled by stitching two separate quadratic NURBS patches together with C0 continuity at the 45-degree boundary. However, this approach will result in stresses that are discontinuous at the C0 patch boundary, so the first approach is advocated in [1] as producing a more physically reasonable result. The same approach was adopted here.

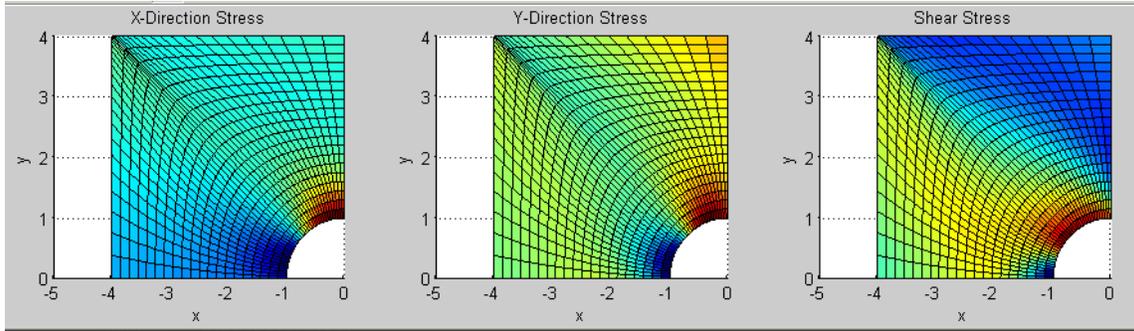
An additional trick was adopted in order to simplify the code. In standard finite element analysis, displacement boundary conditions are applied in strong form by directly prescribing displacement values at nodes, while stress boundary conditions are applied in weak (integral) form. Applying non-constant stress boundary conditions on a finite element mesh (as we are doing here) requires a separate subroutine to numerically integrate the stresses along the boundary in order to produce consistent element loads. Rather than build this additional routine to apply the exact stresses at the boundary, I simply solved for the exact displacement and applied this displacement directly along the boundary at the control points. This clever trick completely backfired, introducing a systematic error into the solution, but teaching an important lesson about a major conceptual difference between NURBS and conventional finite element analysis.

## Results:

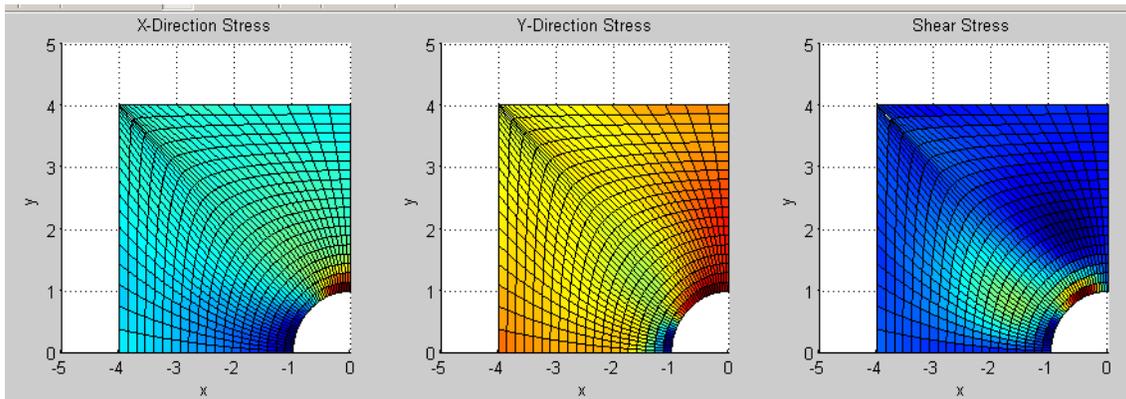
The analysis was conducted on a two-element mesh, beginning with quadratic polynomial order and increasing up to tenth order. A stress of 10 units was applied on the plate, leading to an expected maximum value of stress at the hole of 30 units. As expected, results were less accurate at low polynomial order (in terms of both the maximum stress at the hole and the spatial distribution of RMS stress error) and became more accurate with increased polynomial order. Although knot insertion would have also improved the solution quality, and is in fact essential in order to provide enough elements to achieve reasonable accuracy, the routines to insert knots across the mesh and subdivide it into more elements were not successfully implemented before the project deadline.



**Figure 4.** Exact Solution



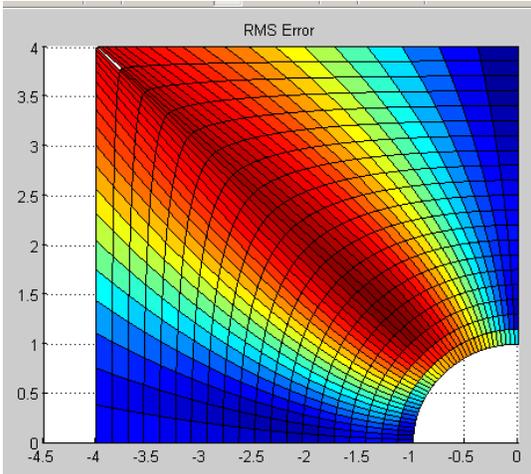
**Figure 5.** Two-element quadratic solution.



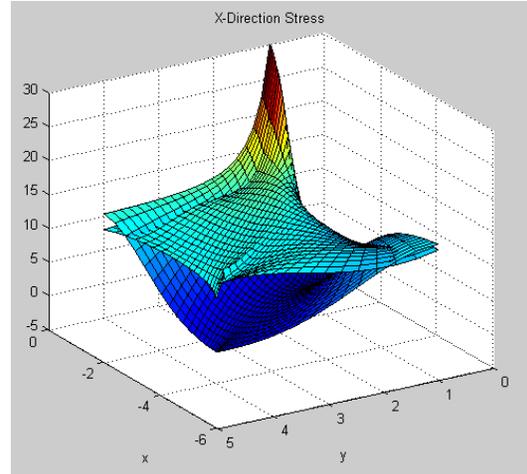
**Figure 6.** Solution on fourth-order mesh

The exact solution is shown in **Figure 4**, the FEA solution on the quadratic mesh in **Figure 5**, and a refined solution at fourth order on **Figure 6**. Although the stress concentration at the hole is fairly accurate even for the coarse mesh (approximately 26 units versus 30 units exact), and becomes more accurate with refinement, the stress **away** from the hole along the 45-degree element boundary is considerably different, as is visible in the high RMS stress error along the 45-degree line (**Figure 7**). It is expected that the stress at element boundaries will be less accurate; however, this degree of inaccuracy is more than should be expected, and seems to indicate systematic error – see **Figure 8** for a depiction of the discrepancy in the x-component of stress. The exact stress lies **below** the surface of the finite element stress, particularly along the 45-degree line.

As mentioned previously, there is a convincing source of systematic error that I missed due to a misunderstanding of the difference between conventional and NURBS-based finite element analysis. The strong imposition of displacement boundary conditions used to simplify the problem relies on the fact that, at the linear boundaries of the mesh, the control points become interpolatory. This allows us to directly prescribe the displacement in this case, even though control points are not generally interpolatory. However, due to the overlapped control points at the corner of the mesh (**Figure 9**), the spacing of the nodes along the boundary is distorted, and the displacement boundary conditions are not applied at the correct interpolatory locations. Rather, the displacement boundary conditions are “smeared out”, rendering the results systematically inexact.

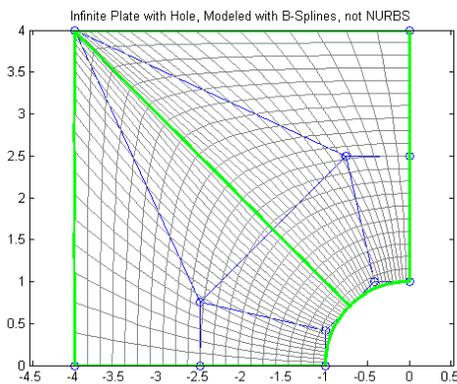


**Figure 7.** RMS stress error

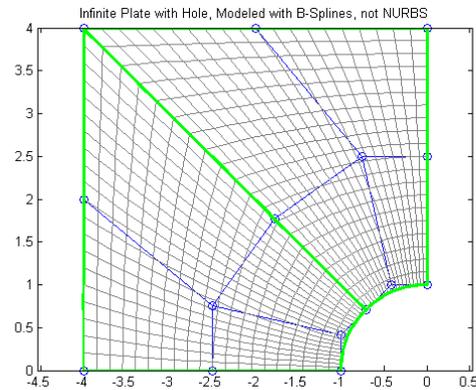


**Figure 8.** Exact vs. finite element x-stress

The original strategy of strongly imposing displacement boundary conditions would have worked had the boundary between the elements been  $C0$ , as in **Figure 10**. Unfortunately this strategy was rejected on physical grounds without my recognition of its impact on the solution strategy, and it was not possible to implement it in time. To complete the study, it would be necessary to either impose the traction boundary conditions, as in [1], or to move to a  $C0$  mesh so that the control points become interpolatory at the straight edges of the mesh.



**Figure 9.** Original  $C1$  mesh.



**Figure 10.**  $C0$  mesh. Note how the control points become interpolatory at the boundary.

### Conclusions:

A B-spline-based finite element code was implemented and used to analyze a classic stress analysis problem. The code architecture was designed for robustness, transparency, and reasonable efficiency, goals that I think were met successfully. However, several conceptual difficulties related to the test problem chosen were encountered which prevented the full success of the final code. These difficulties do not represent particularly difficult hurdles to be overcome, requiring only more time to

implement the required fixes. Results thus far encourage expectations that the code will function as expected once these issues are addressed.

Specifically, it is necessary to include NURBS basis functions instead of B-splines (which merely requires a NURBS-based derivative evaluation routine – I already have routines to evaluate NURBS that work perfectly, just not routines to evaluate their derivatives), the ability to impose general stress boundary conditions needs to be implemented, and knot insertion across the entire mesh must be available. Ultimately an a-posteriori error estimate and a suitable adaptive feedback routine must be designed. Future work will finish these tasks.

Overall, the experience of coding this routine was invaluable in building my understanding of NURBS and NURBS-based finite element analysis. The task turned out to be more difficult than I anticipated, largely due to habits of mind that, like most other finite element analysts, are programmed for the behavior of Lagrange polynomials. The non-interpolatory nature of the boundary conditions and the overlapping support of the functions that must be accounted for in the matrix assembly data structures required a bit of mental adjustment to understand. I anticipate that the results will be worth the adjustment, as evidenced by the ability of the code to analyze meshes at polynomial orders much higher than those possible via Lagrange-based standard FEA.

One last note: to run the code, it is necessary only to run FemTestBsplinePElev.m. However, the current implementation is quite research-y and user-hostile, so no user's manual was included in this report. This will eventually be addressed by including the routines in FEAP, which is much more user-friendly in comparison.