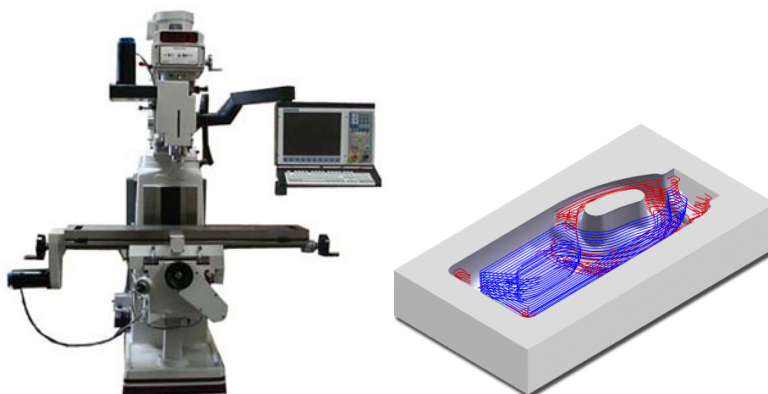


MATH 320 Fall 2010, Homework 2, Computer Aided Machining

Due October 4

1 Motivation

Computer controlled machine tools allow complicated shapes to be cut from a block of material with surprising ease. There are lots of types of machine tools out there, some of them pretty complicated. In this project, we'll devise some of the instructions for a milling machine. (Do a google search for “cnc milling machine” for some neat videos of milling machines in action!)



The photo on the left above is a typical milling machine. On the right is a CAD model that's marked with the path (in blue and red) that the cutting bit will follow. As you may be able to see (much better in the videos), a milling machine works by grinding away little bits of material along a curved path, called the *toolpath*. Your job in this project is to define the toolpath needed to cut out a simple part. In order to do this job, you need to know where the surface of the part is, in relation to the cutting bit.

CAD systems usually store a description of the outside of a part as a collection of parametric surfaces. A parametric surface is a function

$$S(u, v) = \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix}.$$

To keep from getting confused, we call the u, v -coordinates the *surface coordinates*, and x, y, z the *space coordinates*. The tool is controlled in x, y, z coordinates, so you'll use Newton's method to figure out where on the surface (in surface coordinates) the cutting bit is.

Of course, once a toolpath has been defined, one needs to command the milling machine itself. This is usually done using a special language called "G-codes" which have become somewhat standard. We won't generate G-codes in this project as that would take us farther afield than I would like.

2 Objectives

After this assignment, you should...

- Be able to write Newton's method solvers for arbitrary dimensions
- Understand 3d plotting in Octave
- Continue to refine your use of unit tests

3 Your tasks

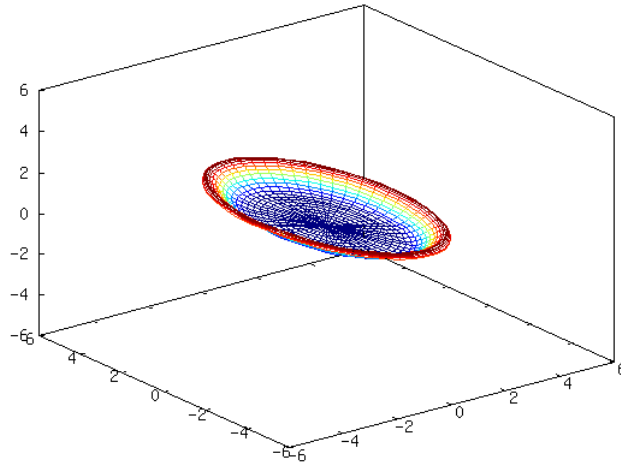
What you are given: I'll give you the following files:

- `partSurface.m` (a function that generates the xyz coordinates, given uv coordinates)
- `plotPart.m` (draws the part for you)
- `withinPart.m` (a function to tell you if a given xy coordinate is within the part)
- `xy2uv_tester.m` (tester for Problem 3)
- `xy2uv_tester2.m` (another tester for Problem 3)
- `toolPath_tester.m` (tester for Problem 4)

The first three files define the part that I want you to instruct a milling machine to cut out. If you call the function

```
octave>> plotPart
```

you should get a plot like the one below. It's a 3d plot that you can rotate and examine.



This part consists of one surface, and we'll be machining the top of it.

Problem 1: Your first task is to specify the xy -coordinates of the toolpath. The way a milling machine usually works is that the horizontal coordinates of the toolpath are specified by the user, and the machine chooses the height of the cutting bit. The toolpath will need to be a back-and-forth sweeping motion across the part. For this problem, write a function *xyToolPath* that has the following format:

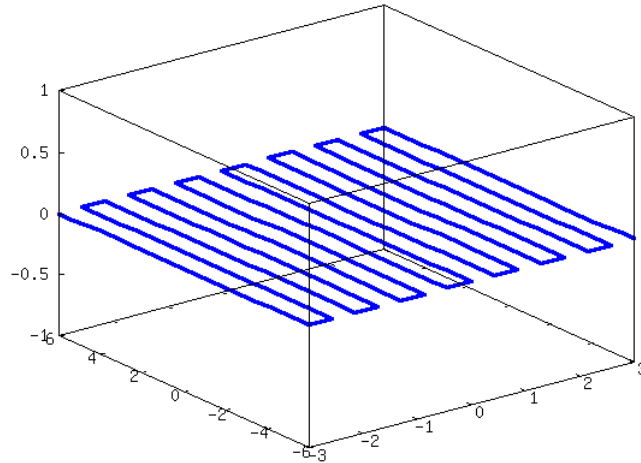
```
function [xs,ys]=xyToolPath(numXPts,numYPts)
% Construct x,y coordinates for toolpath. Sweep in y first, then x.
%
% function [xs,ys]=xyToolPath(numXPts,numYPts)
%
% Input: numXPts = number of paths to sweep
%        numYPts = number of points along a sweep
% Output: xs,ys = vector of points in the path
% Note: It's OK to hard-code the size of the workspace
```

You will want to ensure that the output of the *xyToolPath* covers the entire size of the part (use the function *plotPart* to look at the part, and rotate it to figure out how big it is).

If you execute the following commands,

```
octave>> [xs,ys]=xyToolPath(10,20);
octave>> plot3(xs,ys);
```

you should see a plot like the one below.



Problem 2: Newton's method requires computing a matrix of partial derivatives. See page 113 in your book for a discussion of the formulas. You'll therefore need a function that computes this matrix. So please write *surfJacobian* whose first few lines are below:

```
function J=surfJacobian(u,v)
% Compute the Jacobian matrix at a point on the parametric surface
%
% function J=surfJacobian(u,v)
%
% Input: u,v = surface coordinates to evaluate at
% Output: J = 2x2 matrix Jacobian:
% [ dx/du dx/dv ]
% [ dy/du dy/dv ]
% Note: Call the function partSurface, and used a fixed step size for u, v
```

You can just use the definition of partial derivatives to get an estimate... Also, remember that in Octave, the elements of a matrix J are $J(i, j)$ where i is the row number and j is the column, both starting at 1. Your function will call my function *partSurface* several times to figure out the x and y coordinates, given u and v .

Problem 3: (This is the most important part) Write the function *xy2uv*, which converts a spatial location to surface coordinates.

```
function [u,v]=xy2uv(x,y)
% Compute u,v surface coordinates on part from x,y coordinates
%
```

```

% function [u,v]=xy2uv(x,y)
%
% Input: x,y = spatial coordinates
% Output: u,v = surface coordinates
% Note: Call partSurface and surfJacobian to implement a Newton's method solver

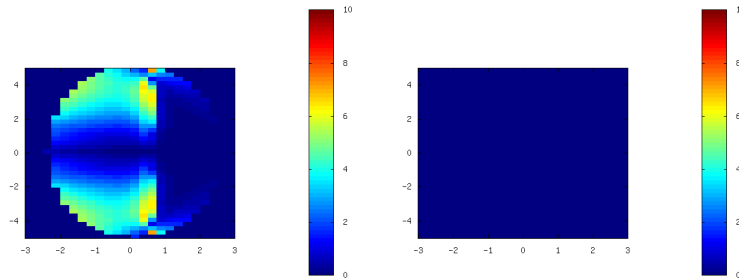
```

You'll need to make a number of design choices in implementing Newton's method:

- How many iterations to use. You might want to try to detect convergence, so that the program runs faster.
- What the initial point should be. You can either try to guess based on looking at my function *partSurface*, or you might try a fixed initial guess (that can work, but it's slower).

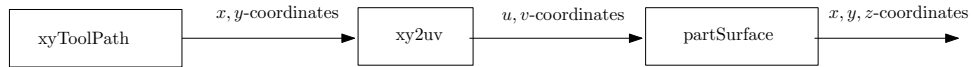
You'll probably run into some bugs when you try to write this function. (I spent about 3 hours working on it when I was making this project; it turns out I was taking the Jacobian incorrectly.) So to help you, I have written two unit testers. The first is *xy2uv_tester*. This function simply calls your *xy2uv* many times with random points, and makes sure that the surface coordinates you give back are consistent with the original input. If your function doesn't work, the tester tells you which points failed.

Another tester is *xy2uv_tester2*, which does much the same thing, but gives graphical output. It plots the distance between the *xy* point we used as input and the *xy* point that corresponds to your output. Ideally this should be pretty small, and the plot should look like the one at right, below. But if your function isn't working, it might look like the one at left, below. (That particular example is what happens when my *xy2uv* doesn't use enough iterations.)



If you are having trouble getting this to work, please ask for help!

Problem 4: Finally, we put all the pieces together. You'll need to write a function that starts out by generating the *xy* coordinates of the toolpath by calling *xyToolPath*. Then, it will solve for the *uv* coordinates for each of these points using *xy2uv*, and pass these into *partSurface* to figure out the correct *z* coordinate. All of these get produced as output.

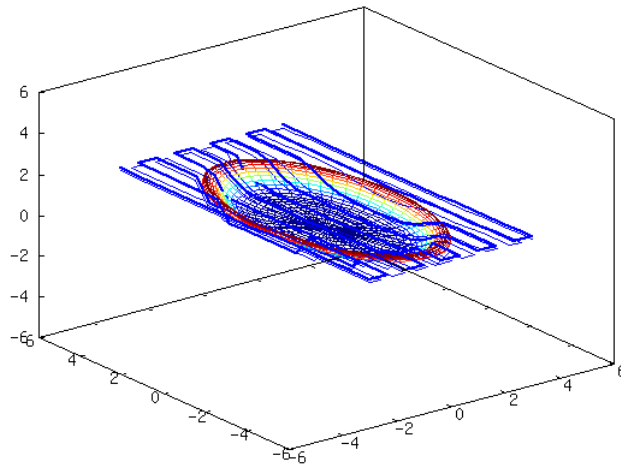


So the following few lines ought to get you started.

```

function [xs,ys,zs]=toolPath(numXPts,numYPts)
% Generate toolpath from surface data
%
% function [xs,ys,zs]=toolPath(numXPts,numYPts)
%
% Input: numXPts = number of paths to sweep
%        numYPts = number of points along a sweep
% Output: xs,ys,zs = vector of points in the path
% Note: Should use xy2uv, then call partSurface to compute the zs.
% You can use withinPart to move the tool out of the way once you're off the part
  
```

If you use the *toolPath_tester* that I supply, you'll be given some output like the following plot.



As I hope you can see, there are two toolpaths (one thick, one thin) that just run along the surface of the part.

Collect all of your functions into a directory (“Folder” for the Windows users!), zip it up, and submit using Blackboard. If you want, you can put scans or electronic documents in that zip file, or you can give them to me in class on paper, but the functions must be submitted electronically.