Torsion of Prismatic Beams of Piecewise Rectangular Cross Section

By

C.H. von Kerczek

1. Introduction:

I present in this note a finite difference method and Scilab computer programs to numerically solve the Saint-Venant theory for torsion of prismatic beams (shafts, bars) of piecewise rectangular cross section. The purpose of this note is mostly educational, for I believe that it is quite instructive to not only solve these problems analytically whenever possible but also explore solutions numerically of common configurations for which one cannot readily obtain analytical solutions. The same kind of problems occur in heat transfer, fluid mechanics, electricity and magnetism, and indeed in any physical situation in which the two dimensional Poisson (Laplace) equation with Dirichlet boundary conditions must be solved. The enclosed programs can be used for problems in these fields with very little, if any, modifications.

Solving the Poisson equation with Dirichlet boundary conditions is not new, to say the least. In fact there are several methods, each of almost endless variety, that have been developed for this purpose. As regards the applications in elasticity theory the dominant method is finite elements, which most students of solid mechanics will come to know by way of widely available computer programs. Then why am I presenting this note, and especially by reverting to the perhaps limited technology of finite differences? For two reasons: The first reason is that the finite difference method transparently mimics the nature of the Poisson equation. The finite difference formulas on the finite difference grid clearly illustrate how the Poisson equation functions. Without the forcing function of the Poisson equation one has the Laplace equation which basically just averages the boundary values into the domain. It is then easy to discern such theorems that the maximum value of the solution of Laplace's equation must occur on the boundaries. Furthermore, the finite difference method leads to the easily visualized mechanical analogy of calculating the equilibrium position of a nominally horizontal net that has weights attached at every intersection of the net strings. This is in fact the discrete version of the soap film analogy so often discussed in books about elasticity theory. The second reason for the finite difference method is that the solution of the Poisson equation on a basic rectangular domain is easily programmed from scratch if one solves the resulting algebraic equations by the relaxation method (Gauss-Seidel iteration). In fact, I think a student could probably write a finite difference program for solving the torsion problem for rectangular cross sections as a homework problem as easily as solving this problem analytically.

Here I will develop a nice little program for U, L, T and I shaped cross sections. Although I present the finished program here rather than have the student do the programming, at least the user should be able to read and understand all the programming steps even if he/she doesn't do the programming.

What follows below is based on the classic text **Theory of Elasticity** by Timoshenko and Goodier [1]. I think this is a great book. It combines insightful mathematical details with very profound physical explanations of the problems treated. It also describes the basic finite difference method applied to elasticity problems. The only thing that I am adding by this note is a nice, easy Scilab implementation of what Timoshenko and Goodier wrote almost 80 years ago. A more modern elasticity text that I like, from which I have taken some aspects of the formulations, is [2].

I first present, below, the basic formulation of the Saint-Venant torsion problem and its finite difference approximations [1]. I do this in two steps. The first step is a very easy program for rectangular cross sections. This program has limited application, but is a good vehicle to illustrate the basic ideas and programing. In the second step I modify this program to deal with U, L, T and I shaped cross sections.

2. Formulation of the Saint-Venant Torsion Theory:

There are two ways in which the torsion problem can be formulated. The first is by way of a warping function and the second is by way of a stress function. Both formulations are clearly explained and illustrated in references [1] and [2]. I will focus on the stress function formulation since it easier to implement. I give a brief outline of the formulation, leaving the details to be looked up in references [1] and [2].

I assume that the prismatic bar is parallel to the z axis with x,y coordinates lying in the nominal cross section of the undeformed bar. I assume there are equal and opposite torques, directed along the z axis, applied to the two ends of the bar. The Saint-Venant Principle allows us to assume that we need not be concerned with how these torques are applied as long as they are statically equivalent to the torque produced by the stresses in cross section planes of the bar. The approach is that there exists a stress function $\phi(x,y)$ from which the *nonzero* stresses are computed by the formulas

(1) $$\sigma_{xz}=\frac{\partial \phi}{\partial y} \qquad \sigma_{yz}=\frac{-\partial \phi}{\partial x}$$

By imposing compatibility on these stresses one can show that the stress function satisfies the equation

(2) $$\nabla^2 \phi=-2G\theta, \quad \nabla^2=\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}$$

where $G=\frac{E}{2(1+\nu)}$, ν is Poisson's ratio, G the torsional rigidity of the bar, E is the elastic constant of the material and θ is the twist angle per unit length of the bar.

The boundary conditions are derived from the fact that there are no tractions on the periphery of the cross section which results in the requirement that

(3) $$\frac{d\phi}{ds}=0$$ over the boundary B(x,y)=0,

where s is arc-length along the boundary, measured in the counter clockwise direction. This condition is equivalent to $\phi = 0$ over the boundary B .

Once the stress function ϕ is determined one can compute the stresses by formulas (1) and then the imposed torque T is balanced by the torque of the shear stresses over the cross section to determine the twist angle θ . The integration of the shear stress torque over the cross section is

(4) $$T = \iint_A [x\sigma_{yz} - y\sigma_{xz}]\,dA$$

where A is the area of the cross section. By substituting the stress equations (1) into equation (4) one obtains

(5) $$T = \iint_A [-x\frac{\partial \phi}{\partial x} - y\frac{\partial \phi}{\partial y}]\,dA \quad ,$$

which can be written

(6) $$T = 2\iint_A \phi\,dA - \iint_A [\frac{\partial x\phi}{\partial x} + \frac{\partial y\phi}{\partial y}]\,dA \quad ,$$

and applying Gauss' theorem to the second integration on the right of (6) we obtain

(7) $$T = 2\iint_A \phi\,dA - \oint_B [\vec{n}\cdot(x\phi\,\vec{i} + y\phi\,\vec{j})]\,dl \quad .$$

The line integral around the boundary B is zero when $\phi = 0$ on B. Thus the integration of the stresses (4) reduces to

(8) $$T = 2\iint_A \phi\,dA \quad .$$

The way that we will compute the stresses for T and I cross sections will require us to consider equation (7) without the line integral over B being zero.

We will compute T by both (4) and (5) (or 7) because this serves as a useful way to asses the accuracy of the calculation.

Before proceeding further, note that we can divide $\phi(x,y)$ by θG to make the problem independent of θG . Then the two integrations (4) and (5) or (7) will be called the J integrals and the torque-angle-of-twist relationship is simply $T = \theta G J$. Afterwards, the stresses computed by formulas (1) must be multiplied by θG to obtain the actual stresses. In this way the beam of given cross section shape need only be solved once. We note that for a beam with a circular cross section J is the polar moment of inertia of the cross section as one learns in elementary Mechanics of Materials.

3. Finite Difference Formulation:

The problems we seek to solve are ones with piecewise rectangular boundaries such as U, L, T and I cross sections. Such cross sections will be embedded in a *basic rectangle* of length a in the x direction and b in the y direction. I scale lengths by the length a so that b is a fraction (or multiple of a). Then the length of the cross section in the x direction is 1. I begin with the formulation for a simple rectangular cross section just to show how easy this is to program. Then I give the modifications for the U, L, T and I shapes in a later Section. The origin of the x,y coordinate system is assumed to be at the centroid of the basic rectangle.

(a) The Grid:

The rectangle is partitioned into a grid, uniform in the x and y directions respectively, but not necessarily with equal x and y increments as shown in Figure 1 below.

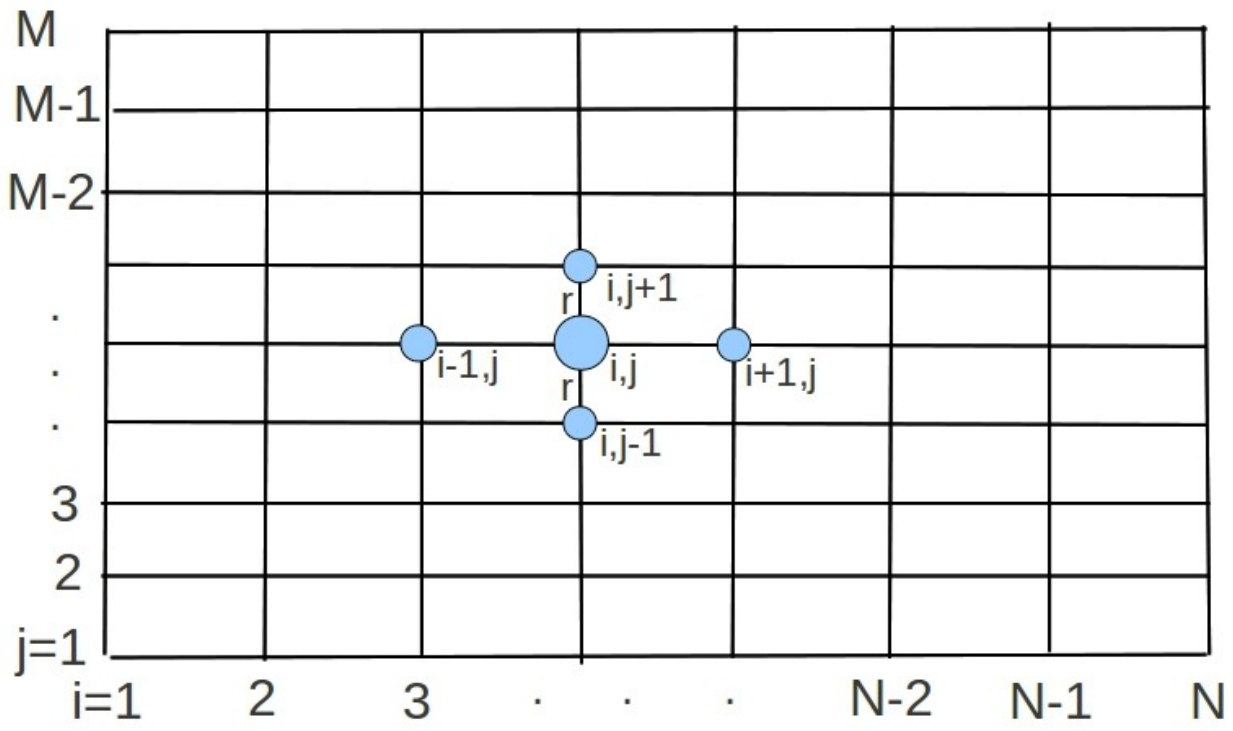

Figure 1. The finite difference grid for for the basic rectangle. The circles depict the *computational molecule (of equation 13 below).*

The N grid points in x and M grid points in y respectively have coordinates

(9) $[-0.5=x_1, x_2, x_3, \ldots, x_N=0.5]$ with $dx=x_i-x_{i-1}$, $i=2,\ldots,N$

and

(10) $[-b/2=y_1.y_2,\ldots,y_M=b/2]$ with $dy=y_j-y_{j-1}$, $j=2,3,\ldots,M$.

The ratio of grid spacing in x to grid spacing in y is r = (dx/dy).

This rectangular grid is always the base grid on which the problem will be solved even for U, L, T and I shaped cross sections.

(b) The finite difference equations:

In general the Poisson equation (3) is to be solved at every point i,j belonging to the *interior* R of the rectangle. The interior constitutes all the grid points with indexes i=2,...,N-1 and j=2,...,M-1. The values of ϕ are specified on the boundary points j=1 and M for $i\in[1,\ldots,N]$ and i=1 and N for $j\in[1,\ldots,M]$. We call these boundary points $B_{i,j}$. We let the net function $f_{i,j}=\phi(x_i,y_j)$ be the numerical values at the grid points of the sought after solution $\phi(x,y)$. The equations for $f_{i,j}$ are found by replacing the derivatives of equation (3) evaluated at each point $i,j\in R$ by finite difference formulas.

The second derivative finite difference formulas on a uniform grid are

(11) $$\left[\frac{\partial^2\phi}{\partial x^2}\right]_{i,j}\approx\frac{f_{i-1,j}-2f_{i,j}+f_{i+1,j}}{dx^2}$$

(12) $$\left[\frac{\partial^2\phi}{\partial y^2}\right]_{i,j}\approx\frac{f_{i,j-1}-2f_{i,j}+f_{i,j+1}}{dy^2}\quad .$$

These formulas for the approximate second derivatives can be derived in a variety of ways that I leave to the reader. Assuming we have divided out the factor θG in all of our problems, the right hand side of the Poisson equation (2) is then equal to -2. By substituting formulas (11) and (12) for the derivatives into (3), one obtains the finite difference set of algebraic equations that must be solved for every point $i,j\in R$.

(13) $$-(2+2r^2)f_{i,j}+[f_{i-1,j}+f_{i+1,j}+r^2(f_{i,j-1}+f_{i,j+1})]=-2\,dx^2\quad .$$

The required boundary conditions are $f_{i,j}=0$ for all values of i,j lying in $B_{i,j}$. Assume these boundary conditions have been set.

(c) Solution of the finite difference equations:

First rewrite equation (13) as

(14) $$f_{i,j}=\frac{1}{(2+2r^2)}[f_{i-1,j}+f_{i+1,j}+r^2(f_{i,j-1}+f_{i,j+1})+2dx^2] \quad .$$

Figure 1 illustrates a *computational molecule* for equation (13). The computational molecule shows how the value of $f_{i,j}$ at a central point i,j depends only on the four surrounding points. Of course the equations (computational molecules) for the neighboring values, say $f_{i,j}$ and $f_{i+1,j}$, overlap and thus it can be seen that the equations (14) for all the points in R are coupled. However, a glance at the computational molecule suggests the following iteration scheme for solving these equations. Make an initial estimate (guess) for $f_{i,j}$ at every point in R, the boundary values being established and held fixed. Then choose a pattern of repeatedly recalculating a new value of $f_{i,j}$ at each point in R according to equation (14), using in the right hand side of (14) whatever the values of the four surrounding points happen to be and repeating this process over and over. After a certain number of iterations, one might expect to home in on a solution. This, in fact, works. It is best for computer calculation to choose a systematic pattern for visiting each grid point in R for the recalculation of $f_{i,j}$ there. I choose simply to work *down the rows and up the columns*. That is, start with the *next to bottom row* j=2. Hold j fixed and sequentially compute $f_{i,2}$ using equation (14) for i=2,3,...,N-1. Next do the same for j=3, then j=4, etc until completing row j=M-1. This is called a *pass (or iteration)* over the field. Do this over and over until the solution emerges. The process does converge.

Here is a little more precise description of the iteration process. Let the superscript k indicate the pass number, with k=1, being an initial guess (almost anything will do, but I usually choose $f^1_{i,j}=0 \quad \text{for all } i,j\in R$ because of reasons that will emerge later). Then for k=1,2,...

(15) $$f^k_{i,j}=q(i,j)[f^k_{i-1,j}+f^{k-1}_{i+1,j}+r^2(f^k_{i,j-1}+f^{k-1}_{i,j+1})+2dx^2] \quad .$$

Carefully note that by choosing the pattern of marching over the field R which starts at j=2 and goes down the rows from i = 2 to M-1, the values of $f_{i-1,j}$ and $f_{i,j-1}$ have already been computed *in this pass* and hence get the superscript k, whereas the values of $f_{i+,j}$ and $f_{i,j+1}$ have not been reached *in this pass* and hence are values from the *previous pass* and have the superscript k-1. By repeating the passes a number of times one can see how the value of $f_{i,j}$ at each point i,j is updated. This method of solving the set of equations (15) is called the *relaxation method*, but is in fact nothing but the Gauss-Seidel method.

I look ahead a bit here to note that the multiplier q(i,j) has been constructed especially to deal in a very simple way with L and U shaped cross sections (to be explained in more detail later). Such cross sections will be defined by areas of the basic rectangle that are *computationally cut out* (called *cutouts*), the remaining grid points forming a U or L shaped section. q(i,j) is defined as follows:

(16) $q(i,j)=\frac{1}{2+2r^2}$ for points i,j *inside of R*, the actual cross section (U or L shape)

$q(i,j)=0$ for points i,j that lie *outside of R,* in the *cutout* regions of U and L..

By starting the iteration process with $f_{i,j}=0$ everywhere, the boundary conditions are automatically satisfied because even though equations (15) uses the boundary points, the value $f_{i,j}$ *on* the boundary $B_{i,j}$ is never computed by (15).

Just to illustrate how easy it is to implement this solution procedure, below is a program fragment that solves the finite difference equations.

```
for k=2:K
    for j=2:M-1
        for i=2:N-1
            f(k,i,j) = q(i,j)[f(k,i-1,j) + f(k-1,i+1,j) + r(f(k,j-1) + f(k-1,j+1)) - dx^2 W(i,j)]
        end
    end
end
```

For this illustration I have chosen to do a fixed number of K passes. How do we know the solution will have converged after K passes? We don't, but one can experiment to see how many passes are needed. I have found that for a rectangular region, which surprisingly is the worst case, about NM passes is much more than needed for convergence and does not really take too much computer time. Of course the accuracy of the solution depends not only on the degree of convergence, but also the smallness of dx and dy. I do not want to belabor the point here because in the actual implementation of this solution method I will include a convergence criteria to stop the iteration after the number of passes for which the criterion has been satisfied. The purpose of the above program fragment is only to illustrate the simplicity of the relaxation method for solving the equations (15), the heart of this whole method of solving the Poisson equation.

4. Specific Implementations:

(a) Torsion of a rectangular cross section bar:

The case of a rectangular bar is particularly simple, and the above explanation should be sufficient to implement the solution. The only thing that needs to be added is implementation of the computation of the stresses and the J integrals.

The numerical computation of the stresses and the J integral are obtained from the numerical stress function $f_{i,j}$ using the following formulas:

(17) $$\sigma_{xz,i,j}=\frac{f_{i,j+1}-f_{i,j-1}}{2dy} \qquad \sigma_{yz,i,j}=\frac{-f_{i+1,j}-f_{i-1,j}}{2dx}$$ for all points $i,j\in R$.

These 1st derivative formulas are order dy^2 and dx^2 accurate like the second derivative formulas (11) and (12), because they are *central difference* formulas derived from a quadratic fit to three consecutive points. However, we also need to compute the stresses at the

boundaries where formulas (17) do not apply because they require values of f outside the rectangle. For example at the boundary j=1, the value of $f_{i,j-1}=f_{i,0}$ and no such point exists. One could apply formulas like (17) at the boundaries by just making them one sided (forward or backward) difference formulas. For example at the boundary j=1, one could compute

$$\sigma_{xz,i,1}=\frac{f_{i,2}-f_{i,1}}{dy}$$

but such formulas are much less accurate, of order dy in the above example. To maintain the same level of accuracy for the boundary stresses as the interior stresses, we use a second order $(dy^2 \text{ or } dx^2)$ formulas, derived by fitting a quadratic polynomial to three consecutive points, differentiating it and evaluating this derivative at the desired end point of these three points rather than the middle point as for (17). The results are

(i) for the bottom and top of the rectangle, j=1, and j=M, i=1 to N respectively

(18a) $$\sigma_{xz,i,1}=\left[\frac{\partial \phi}{\partial y}\right]_{i,1}=\left[\frac{-3f_{i,1}+4f_{i,2}-f_{i,3}}{2dy}\right]$$

(18b) $$\sigma_{xz,i,M}=\left[\frac{\partial \phi}{\partial y}\right]_{i,M}=\left[\frac{f_{i,M-2}-4f_{i,M-1}+3f_{i,M}}{2dy}\right] ,$$

and of course $\sigma_{yz}=0$ on these two boundaries;

(ii) for the left and right ends of the rectangle, i=1, and i=N, j=1 to M, respectively

(19a) $$\sigma_{yz.1,j}=-\left[\frac{\partial \phi}{\partial x}\right]_{1,j}=-\left[\frac{-3f_{1,j}+4f_{2,j}-f_{3,j}}{2dx}\right]$$

(19b) $$\sigma_{yz.N,j}=-\left[\frac{\partial \phi}{\partial x}\right]_{N,j}=-\left[\frac{f_{N-2,j}-4f_{N-1,j}+3f_{N,j}}{2dx}\right] .$$

For ease of notation let

(20) $$H_{i,j}=[x\sigma_{yz}-y\sigma_{xz}]_{i,j} \quad \text{or} \quad =\phi_{i,j} .$$

Each grid rectangle of area dxdy, called a cell, is denoted by the i,j value at its *upper right hand corner*. Then computation of the J integrals is carried out by averaging the values of H at the corners of each cell, multiplying this average by dxdy, and then adding these up over the entire grid to obtain the value of the integral. This is just the two dimensional analog of the trapezoidal rule of integration. That is

(18) $$J=\iint_A H\,dA=\frac{dxdy}{4}\sum_{i=2}^{N}\sum_{j=2}^{M}[H_{i-1,j-1}+H_{i,j-1}+H_{i-1,j}+H_{i,j}] .$$

This completes the numerical formulation for the twisting of a rectangular cross

sectioned bar by an end torques. The implementation is shown in the Scilab program *TorqueR* below. The comments in the program, that include reference to the formula numbers in the text that are being implemented, should suffice for understanding the program.

```
                    Program TorqueR
clear
//*********************************************************************
//
//   Poisson Equation with Dirichlet boundary conditions on a
//   rectangle of length 1 and width b.
//
//*********************************************************************
//
//  1. Input
//
N=41; M=21;  // Overall # of Grid points in x and y respectively.
b=0.5;       // Rectangle width b.
//
//*********************************************************************
//
//  2. Data setup
//
x=linspace(0,1,N);           // The x grid points (9)
y=linspace(0,b,M);           // The y grid points (10)
dx=x(2)-x(1); dy=y(2)-y(1);  // The z and y grid increments
x=x-1/2; y=y-b/2;            // Origin at centroid of rectangle
r2=(dx/dy)^2;
c=(2+2*r2);                  // Coefficient in eq (15)
q=ones(N,M)/c;               // No cutouts in this program (15)
K=N*M;          // Maximum number of iterations (passes) allowed
w=2*dx^2;       // RHS of equation (16)
//---------------------------------------------------------------------
f=zeros(N,M);    // Starting values of the stress function. This
                 // also sets the boundary values of f.
//
//
//*********************************************************************
//
//  3. Solution procedure by iterations (See program fragment
//     for eq. (15)
//
for k=1:K       // loop1-----------------------------------
  for i=2:N-1   // loop2-----------------------------------
```

```
    for j=2:M-1 // loop3----------------------------------
      f(i,j)=q(i,j)*(f(i-1,j)+f(i+1,j)+r*(f(i,j-1)+f(i,j+1))+w);
                                                  // Eq (15)
      end       // End of loop3-----------------------------
    end         // End of loop2-----------------------------
  end           // End of loop1-----------------------------
end
//
//*****************************************************************
//
// 4. Calculation of stresses.
//
sxz=zeros(N,M); syz=zeros(N,M);   // Initialize the stresses to 0
//
// Stresses at the interior points R
//
for i=2:N-1
    for j=2:M-1
        sxz(i,j)=(f(i,j+1)-f(i,j-1))/(2*dy);     // 1st of (17)
        syz(i,j)=-(f(i+1,j)-f(i-1,j))/(2*dx);    // 2nd of (17)
  end
end
//
// Stresses on the outer boundary points  B_{i,j}
//
for i=1:N
    sxz(i,1)=(-3*f(i,1)+4*f(i,2)-f(i,3))/(2*dy);      // (18a)
    sxz(i,M)=(f(i,M-2)-4*f(i,M-1)+3*f(i,M))/(2*dy);   // (18b)
end
for j=1:M
    syz(1,j)=-(-3*f(1,j)+4*f(2,j)-f(3,j))/(2*dx);     // (19a)
    syz(N,j)=-(f(N-2,j)-4*f(N-1,j)+3*f(N,j))/(2*dx);  // (19b)
end
//
//*****************************************************************
//
//  5. Calculation of the J integrals:
//
// By stress integration
//
J=0;
for i=2:N
  for j=2:M
  t=x(i)*(syz(i,j)+syz(i,j-1))+x(i-1)*(syz(i-1,j-1)+syz(i-1,j));
  J=J+t-(y(j)*(sxz(i,j)+sxz(i-1,j))+y(j-1)*(sxz(i,j-1)+...
    sxz(i-1,j-1))); // (20) for H in terms of stresses in (17-19)
    end
```

```
end
J=dx*dy*J/4;
//
// By stress function integration
//
Jf=0;
for i=2:N
    for j=2:M
        Jf=Jf+f(i,j)+f(i,j-1)+f(i-1,j)+f(i-1,j-1); // (20) H=f
    end
end
Jf=2*dx*dy*Jf/4;
//
//************************************************************
//
//  6. Output
//
subplot(2,2,1)
plot3d(x,y,sxz)
title("The stress sxz")
subplot(2,2,2)
plot3d(x,y,syz)
title("The stress syz")
subplot(2,2,3)
plot3d(x,y,f)
title("The stress function")
disp(k,'# of iterations')
disp(J,'J integral by stress integration')
disp(Jf,'J integral by stress function integration')
```

A sample run of this program for a rectangle of width b=0.5 is shown in Figure 2 below.

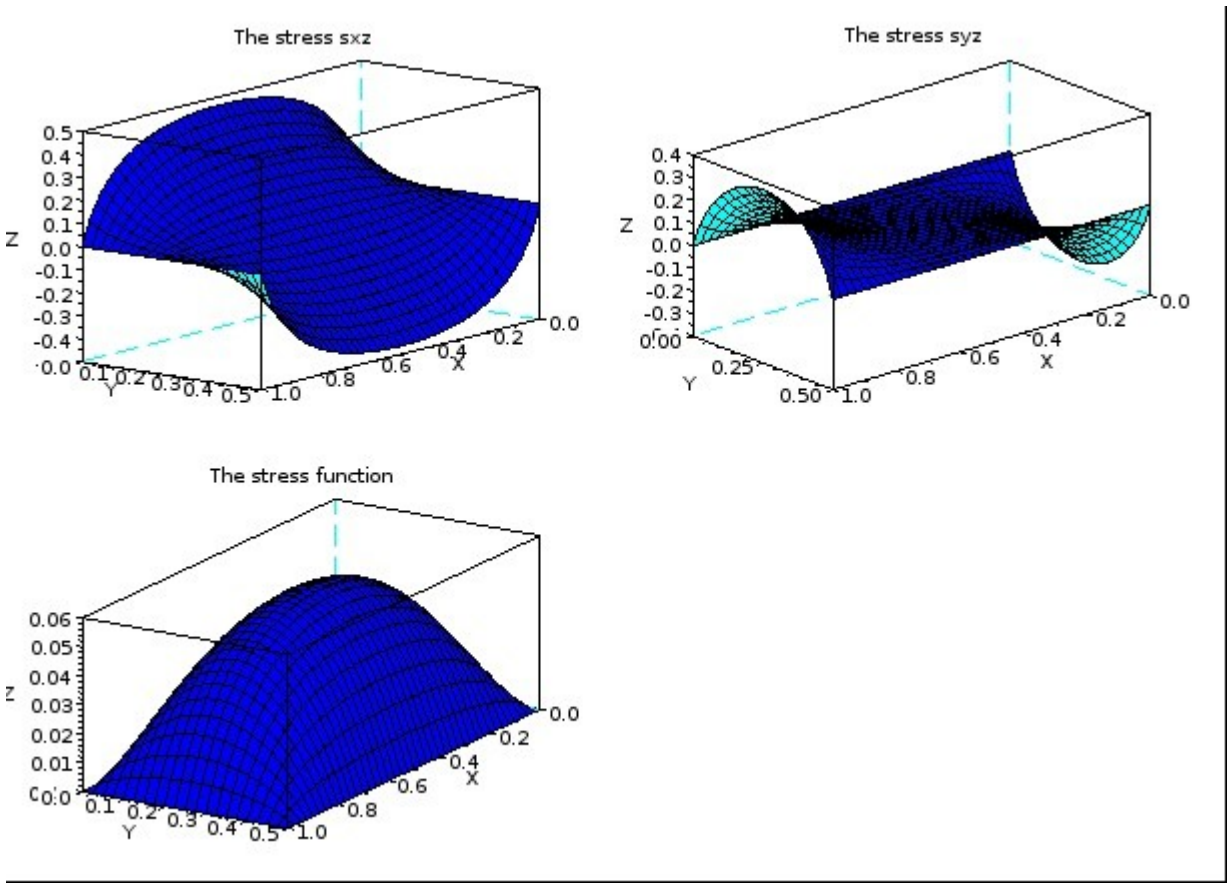

Figure 2. The stresses and stress function for a twisted prismatic rectangular cross section bar of width b=0.5. N=41, M=21. J (by stress)=0.0287, J (by stress function)=0.0284.

Notes about program TorqueR:

(i) I have kept this program as primitive as possible so that anyone with just very basic programming skills can easily understand, duplicate and modify it. In particular, the input to the program is not a separate input file since it is so easy to simply call up the program in Scilab's edit window, make the input changes directly in the program and hit the save and execute buttons. If one converts the program to Matlab, the same operating procedure would apply.
(ii) I usually choose N and M so that dx=1/(N-1) and dy=b/(M-1) are nice numbers like 0.01 or 0.025 etc.
(iii) The total number of iterations K=NM is way more than enough. In a later program (the one for the L and U sections) I have implemented a convergence criteria to limit the number of passes. Running this case on *ProgramR* required about 450 iterations and took about 45 seconds external clock time.
(iv) I have not saved the data arrays for the stresses and stress function. Only the final values of J, Jf and number of iterations have been printed in the command window. If one wanted the detailed stress and stress function data, one could add output files at the end of the program for this. (See the Scilab help button for the appropriate syntax.)

(v) You can easily convert this program to Matlab. Simply change the // comment indicators to % and change the graphical function syntax to Matlab's which can be found in Matlab's help file.

The student might be interested in using the program TorqueR to compare results with those of reference [1], Section 109, p.309. It should be noted that my a (a=1 here) and b are twice those of reference [1].

(b) Torsion of L and U shaped cross sectioned bars:

The torsion of rectangular cross section is somewhat limited. Of more interest are cross sections of complicated shapes such as L and U shapes. For such cross sections one can derive analytical solutions by way of complex variables and conformal mapping. But this is a somewhat more advanced and indirect method and not in keeping with the elementary method advocated here.

We deal now specifically with setting up the L and U shape geometries. As mentioned above, the geometries are obtained simply by cutting out or blanking out those regions of the basic rectangle that will result in an L or U shaped section of what remains. The U sections are called *even* if both legs of the U are the same (=b), and *uneven* if the right leg is <b. All we need to do is specify the boundaries of the L or U *cutouts*. Figure 2 below shows the basic rectangle and the cutouts that will form an uneven U shape. The cutout is the blue region.

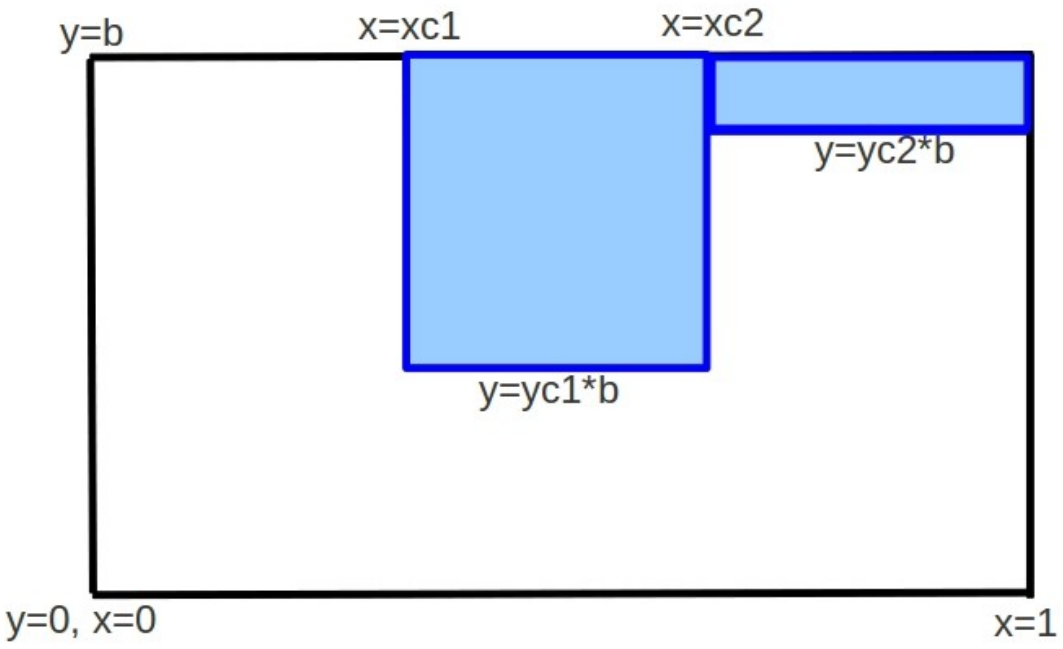

Figure 2. The basic rectangle with the blue portion cut out to form the uneven U cross section. To define the cross section we place the x,y coordinate system origin at the lower left of the basic

rectangle.

In Figure 2 the cutouts are defined by their boundaries as follows:

(i) The *bottom boundary of* the *U cutout* is specified by $y=y_{c1}$, where y_{c1} is a *fraction* of b. This boundary will be called the y_{c1} boundary.

(ii) The *top of the right leg of the U* is specified by $y=y_{c2}$, where y_{c2} is a *fraction* of b. This is called the y_{c2} boundary. Note Figure 2 shows an *uneven U section with yc2<yc1*. For an *even* U yc2 would be =yc1.

(iii) The *left side of the U cutout* is specified by $x=x_{c1}$ and this is called the x_{c1} boundary.

(iv) The *right side of the U cutout* is specified by $x=x_{c2}$ and this is called the x_{c2} boundary.

Figure 2 should make clear the boundaries of the U shaped cross section. The L shaped cross section is obtained simply by setting $x_{c2}=1$. The basic rectangle is obtained by making $x_{c1}=1$, $x_{c2}=1$ and $y_{c1}=1$.

T and I shaped cross sections are set up as L and U shaped sections, respectively, as above, but since T and I sections are symmetric about the y=0 line in Figure 2, we simply solve the L or U shape sections with a *symmetry boundary condition* applied at y=0. We will explain this further below.

The grid to be used for the L and U shaped sections (and the T and I sections by reflection about the bottom boundary) is shown in Figure 3 below

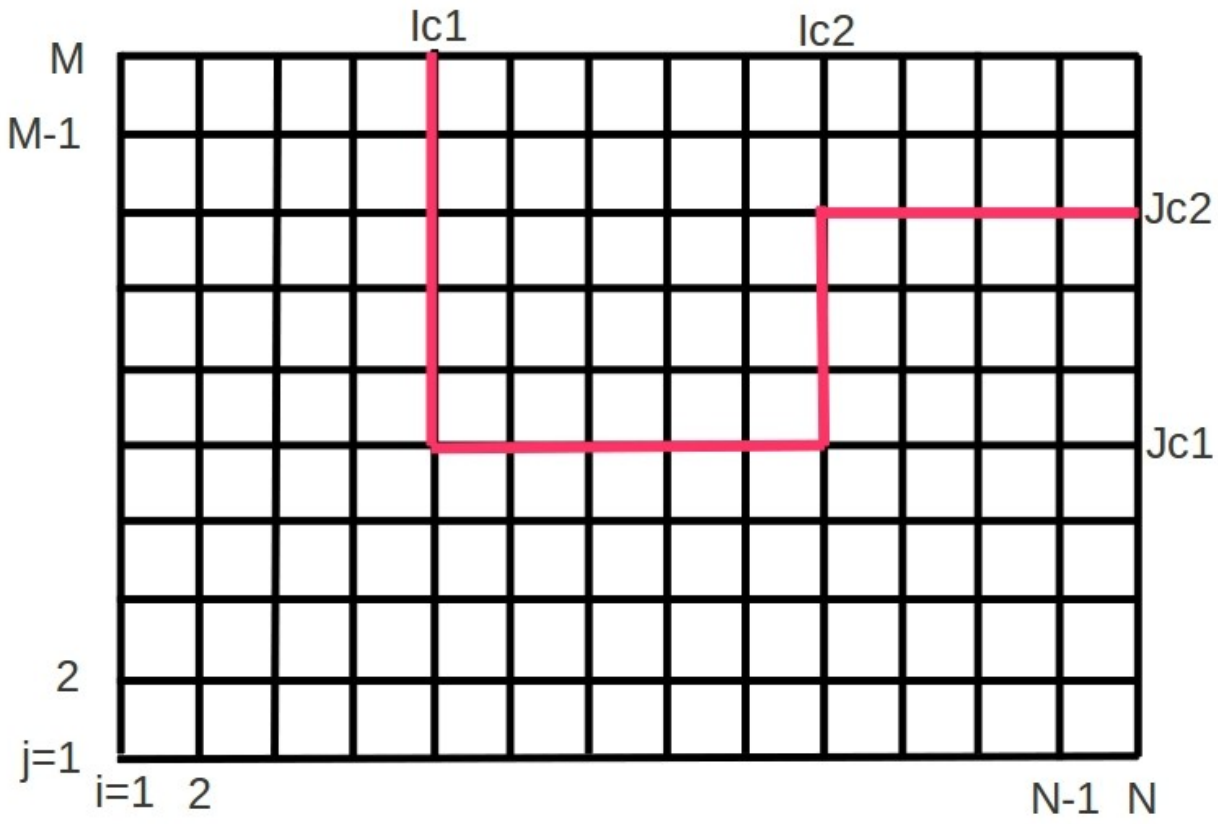

Figure 3. The grid for the L, U, T and I shaped cross sections. The part of the grid above the red lines are numerically cutout of the overall rectangle.

The grid boundaries indicated by the red grid lines i=Ic1, i=Ic2, j=Jc1 and j=Jc2, shown in red in Figure 3, are derived from the specified cross section shape boundaries xc1, xc2, yc1 and yc2, respectively, in the following way.

(i) The boundary yc1 is translated into the grid line Jc1 by Jc1=ifix(yc1*b/dy+0.1dy). The function ifix(z) truncates the argument to the nearest integer below the value of z. We add the term 0.1dy to the argument to guard against the possibility that truncation producing a whole integer value below z due to small roundoff error. The other grid boundary integers are derived in a similar way.

(ii) For boundary yc2, Jc2=ifix(yc2*b/dy+0.1dy).

(iii) For boundary xc1, Ic1=ifix(xc1/dx+0.1dx).

(iv) For boundary xc2, Ic2=ifix(xc2/dx+0.1dx).

The solution procedure of the finite difference equations (15) remains unchanged from

the way they were solved on the rectangular grid. All that needs to be done is to set q(i,j)=0 on all the boundary points and the points above the red boundaries. This is done simply as in the program fragment shown below:

```
q=ones(N,M);
for j=1:M
    q(1,j)=0; q(N,j)=0;
end
for i=1:N
    q(i,1)=0; q(i,M)=0;
end
for j=Jc1:M
    for i=Ic1:Ic2
        q(i,j)=0;
    end
end
for j=Jc2:M
    for Ic2:N
        q(i,j)=0;
    end
end
```

By employing this method of handling the cutout regions numerically, it may seem a bit wasteful to perform the iterations at all the grid points where the $f_{i,j}=0$, but this is compensated for by avoiding not only the programming difficulty of putting a lot of *if statement* logic inside the for loops of the iteration, but also avoiding the computational time penalty of such *if statement* logic. The number of processor cycles required to execute an if statement vastly exceeds the number of cycles required by additions and multiplications. So overall execution speed probably balances out.

The remaining modifications that need to be made to the *TorqueR* program only involves the calculation of the stresses on the new boundaries marked in red in Figure 3.

First, the stresses σ_{xz} and σ_{yz} are computed on the entire interior of the rectangle grid as in *TorqueR,* even over the grid points in the cutout regions where $f_{i,j}=0$. Since $f_{i,j}=0$ in the cut out region the stresses will be zero there. The stresses on the outer boundaries of the entire rectangle are computed exactly as in *TorqueR*, requiring no changes in the program. The only additions that are needed is to compute the boundary stresses on the red lines in Figure 3 using the appropriate formulas (18) and (19). Rather than describing these changes here we refer to the new program TorqueULS, listed below, where these new program steps are indicated by comments.

TorqueULS

```
clear
////////////////////////////////////////////////////////
//   Torque on a piecewise rectangular cross section
```

```
//
//-------------------------------------------------------------
//    Basic rectangle in the x,y plane. Scaled so
//    that length in x=1 and height in y=b.
//
////////////////////////////////////////////////////////////
//
//  1. Input
//
//  A Basic rectangular grid overlies the entire cross section.
//  The cross section can hace the following block shapes:
//  1. L; 2. U; 3. T; 4. I.
//  These shapes are obtained by numerically cutting out pieces
//  from the basic rectangle. This is accomplished by specifing
//  two points xc1 and xc2 along the x axis and then points yc1
//  and yc2 (fractions of b) which give the height from the
//  lower boundary to the length between xc1 and xc2 and
//  xc2 and 1 respectively. This gives the asymmetric shapes
//  of L and U. To obtain the symmetric shapes, T and I, about
//  the lower boundary y=0, we simply specify Ks=0 to make y=0
//  a line of symmetry and solve the top half of the section.
//-------------------------------------------------------------
//
N=41; M=21; // Overall # of Grid points in x and y.
b=0.5;      // dimension of the overall rectangle height.
xc1=0.3;  // Left boundary of the L or U cutout; fraction of 1.
xc2=0.7;  // Right boundary of the L or U cutout;fraction of 1.
yc1=0.5;  // Bottom location of L or U cutout between xc1 and
          //  xc2;
yc2=0.8;  // Bottom location of L or U cutout between xc2 and
          //  1;
       // Note: Setting xc1, xc2, yc1, yc2 =1 in an appropriate
       // combinations gives the various section shapes. For no
       // cutouts, set all =1.
Ks=1;  // Ks=0 gives T or I, Ks=1 gives L or U or rectangle.
K=2000;    // Maximum number of iterations allowed.
err=0.00001; // Convergence criteria of the stress function
             // iteration.
////////////////////////////////////////////////////////////
//
//  2. Data setup
//
x=linspace(0,1,N); y=linspace(0,b,M);  // x and y Grid points
dx=x(2)-x(1); dy=y(2)-y(1);
x=x-0.5; y=y-b/2;    // Put origin at center of the rectangle.
dx2=dx^2;
r2=(dx/dy)^2;
c=1/(2+2*r2);
```

```
//-------------------------------------------------------------
//  Set up the cutout limits based on grid points.
//
Jc1=fix(yc1*b/dy+0.1*dy)+1;  // Bottom cutout of U grid line.
Jc2=fix(yc2*b/dy+0.1*dy)+1;  // Top cutout right vertical of U.
Ic1=fix(xc1/dx+0.1*dx)+1;    // Left cutout of  U grid line.
Ic2=fix(xc2/dx+0.1*dx)+1;    // Right cutout of U grid line.
bcs=[y(Jc1),y(Jc2),x(Ic1),x(Ic2)];
disp('The actual values of yc1, yc2, xc1, xc2 referred to
     the center of R)
disp(bcs)                   // Print out the cutout boundaries
//-------------------------------------------------------------
//  Set up the RHS function for the Poisson equation.
//
W=2*dx2
//-------------------------------------------------------------
q=ones(N,M);
//
// Blank out the cutouts and boundaries.
//
for i=1:N
    q(i,1)=0; q(i,M)=0;
end
for j=1:M
    q(1,j)=0; q(N,j)=0;
end
for j=Jc1:M
    for i=Ic1:Ic2
        q(i,j)=0;
    end
end
for j=Jc2:M
    for i=Ic2:N
        q(i,j)=0;
    end
end
q=c*q;
//-------------------------------------------------------------
f=zeros(N,M);  // Starting values of the stress function.
//-------------------------------------------------------------
// By taking these as the starting values of f in the iteration
// procedure,
// homogeneous boundary conditions are automatically imposed.
//-------------------------------------------------------------
//
////////////////////////////////////////////////////////////
//
//  3. Solution procedure by iterations.
```

```
//
k=0;
em=1;
while em>err
    e=0; em=0;
    // The special j=1 row of equations(15) incorporats the
    // symmetry condition.
    if Ks==0
    for i=2:N-1
        f(i,1)=c*(f(i-1,1)+f(i+1,1)+2*r2*f(i,2)+W); // Eq. (22)
    end
    end
    //-----------------------------------------------------
   for j=2:M-1
      for i=2:N-1
      fo=f(i,j);
      f(i,j)=q(i,j)*(f(i-1,j)+f(i+1,j)+r2*(f(i,j-1)+f(i,j+1))
             +W);                  // Equation (15)
      end
    end
  end
  k=k+1;
  if k>K then
      break
  end
end
////////////////////////////////////////////////////////////
//
// 4. Calculation of stresses.
//
sxz=zeros(N,M); syz=zeros(N,M);
//
// Stresses at all the interior points of R.
//
for i=2:N-1
    for j=2:M-1
        sxz(i,j)=(f(i,j+1)-f(i,j-1))/(2*dy);
        syz(i,j)=-(f(i+1,j)-f(i-1,j))/(2*dx);
    end
end
//-----------------------------------------------------------
// Stresses at all the body boundary points.
//
if Ks==0
    // symmetry BC at the lower boundary of the rectangle:
for i=2:N-1
    sxz(i,1)=0; syz(i,1)=-(f(i+1,1)-f(i-1,1))/(2*dx);
    sxz(i,M)=(f(i,M-2)-4*f(i,M-1)+3*f(i,M))/(2*dy);
```

```
end
else
    // Zero BC at the lower boundary of the rectangle:
 for i=2:N-1
    sxz(i,1)=(-3*f(i,1)+4*f(i,2)-f(i,3))/(2*dy);
    sxz(i,M)=(f(i,M-2)-4*f(i,M-1)+3*f(i,M))/(2*dy);
end
end
    // Stresses at the left and right rectangle boundaries:
for j=1:M
    syz(1,j)=-(-3*f(1,j)+4*f(2,j)-f(3,j))/(2*dx);
    syz(N,j)=-(f(N-2,j)-4*f(N-1,j)+3*f(N,j))/(2*dx);
end
//---------------------------------------------------------------
// The stresses at the cutout boundaries.
//
    // Boundary Jc1;
for i=Ic1:Ic2
    sxz(i,Jc1)=(f(i,Jc1-2)-4*f(i,Jc1-1)+3*f(i,Jc1))/(2*dy);
end
    // Boundary Jc2;
for i=Ic2:N
    sxz(i,Jc2)=(f(i,Jc2-2)-4*f(i,Jc2-1)+3*f(i,Jc2))/(2*dy);
end
    // Boundary Ic1;
for j=Jc2:M-1
    if Ic2<N-1
    syz(Ic2,j)=-(-3*f(Ic2,j)+4*f(Ic2+1,j)-f(Ic2+2,j))/(2*dx);
    else
    end
    // Boundary Ic2;
    if Ic1<Ic2-1
    syz(Ic1,j)=-(f(Ic1-2,j)-4*f(Ic1-1,j)+3*f(Ic1,j))/(2*dx);
    else
    end
end
////////////////////////////////////////////////////////////
//
//  5. Calculation of the torque T:
//
// By stress integration
J=0;
for i=2:N
    for j=2:M
       t=x(i)*(syz(i,j)+syz(i,j-1))+x(i-1)*(syz(i-1,j-1)+syz(i-
           1,j));
    J=J+t-(y(j)*(sxz(i,j)+sxz(i-1,j))+y(j-1)*(sxz(i,j-1)+sxz(i-
           1,j-1)));
```

```
    end
end
J=dx*dy*J/4;
//
//-------------------------------------------------------------
// By stress function integration
Jf=0;
for i=2:N
    for j=2:M
        Jf=Jf+f(i,j)+f(i,j-1)+f(i-1,j)+f(i-1,j-1);
    end
end
Jf=2*dx*dy*Jf/4;
if Ks==0
B=0;
for i=2:N
    B=B+0.5*dx*(f(i,1)+f(i-1,1));  // The line integral (23)
end
Jf=Jf-y(1)*B;
end
////////////////////////////////////////////////////////////////
//
//  6. Output
//
subplot(2,2,1)
plot3d(x,y,sxz)
title("The stress sxz")
subplot(2,2,2)
plot3d(x,y,syz)
title("The stress syz")
subplot(2,2,3)
plot3d(x,y,f)
title("The stress function")
disp(k,'# of iterations')
disp(J,'J integral by stress integration')
disp(Jf,'J integral by stress function integration')
```

Below are a couple of sample cases:

Example 1. An L shaped region where the cutout is the upper right quadrant.

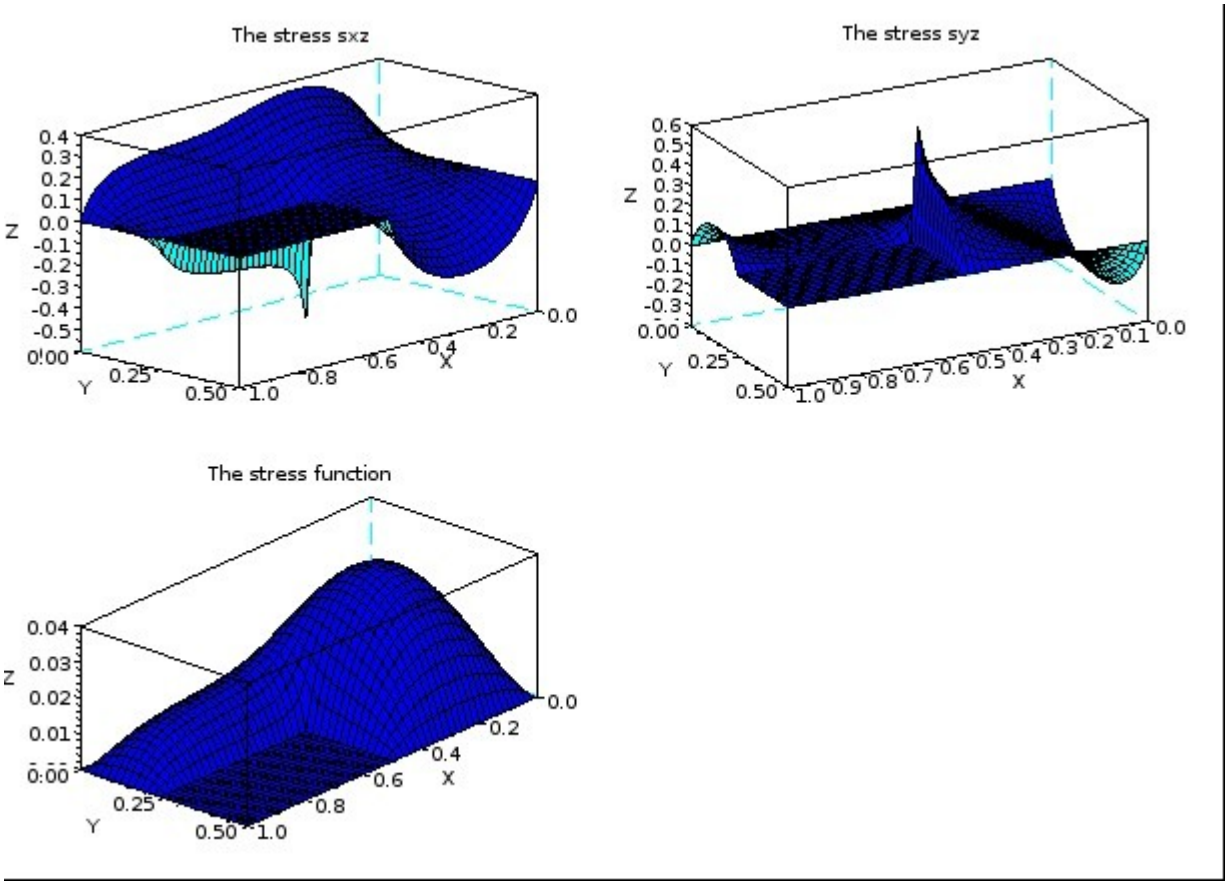

Figure 4. The torqued L cross section. b=0.5, $y_{c1}=0.5b$, $y_{c2}=b$, $x_{c1}=0.5$, $x_{c2}=1$. M=51, N=26. Required k=429 iterations.

The J integral by stress integration = 0.0125 and J by stress function integration = 0.0117. It should be observed that theoretically the stresses are singular at the interior corner (here at the center of the basic rectangle), but numerically this will only be simulated by large values of stress there. The stress function, however, is not singular there. Because of the singular nature of the stresses at the interior corner, it is to be expected that the J integral by stress integration is somewhat different than by stress function integration. The later value should be more reliable. The agreement between the values of the J integrals for stress and stress function integrations becomes smaller as the grid is refined (N and M are made larger). The calculation required about 1 minute and 7 seconds by my wrist watch.

Example 2. This case demonstrates the solution for an *even* U shaped cross section.

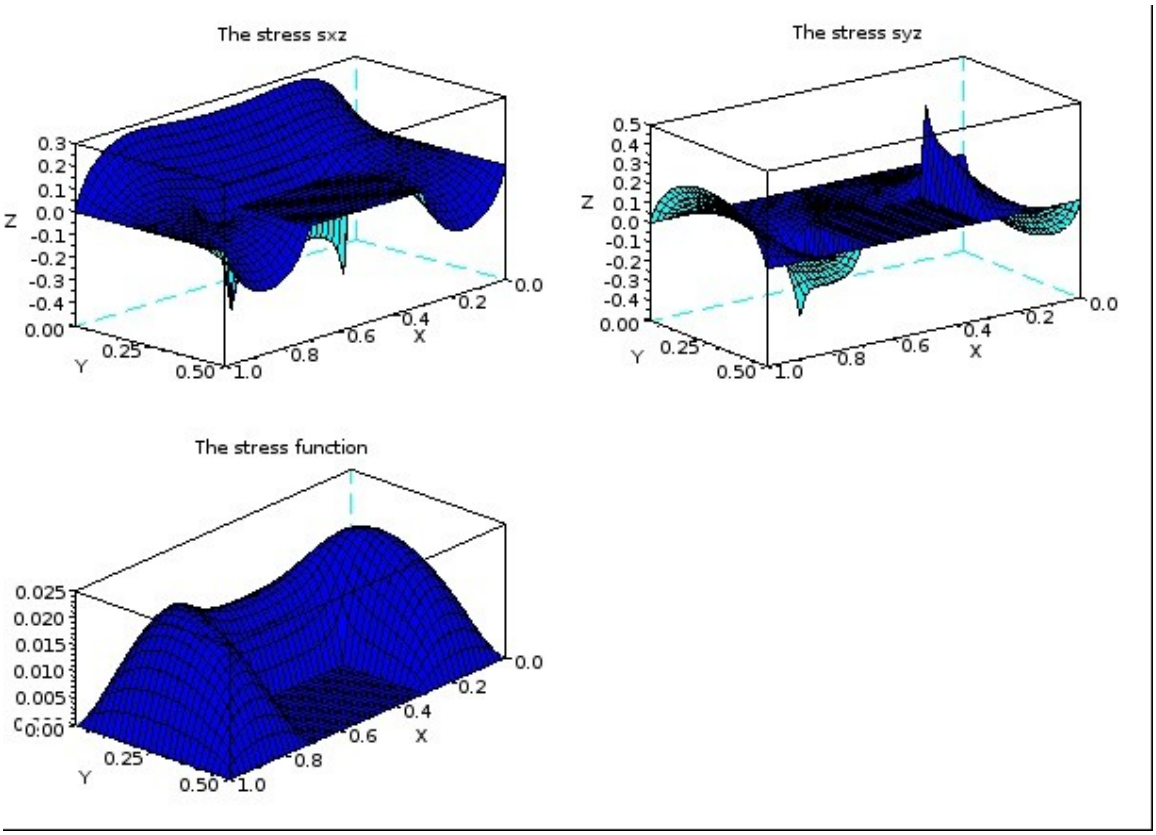

Figure 5. Torqued even U shaped cross section. $y_{c1}=0.5b$, $y_{c2}=b$, $x_{c1}=0.3$, $x_{c2}=0.7$. M=51, N-26, # of iterations k=253.

The J integral by stress integration = 0.00900 and J integral by stress function integration = 0.00883. Here the difference between the two ways of calculating the J integrals is more pronounced because there are now two singular points in the stress distributions.

Example 3: This example is for a *uneven* U beam where we use exactly the same input as Example 2 except yc2=0.8b (see Figure 2). The shear stresses and stress function are plotted in Figure 6 below.

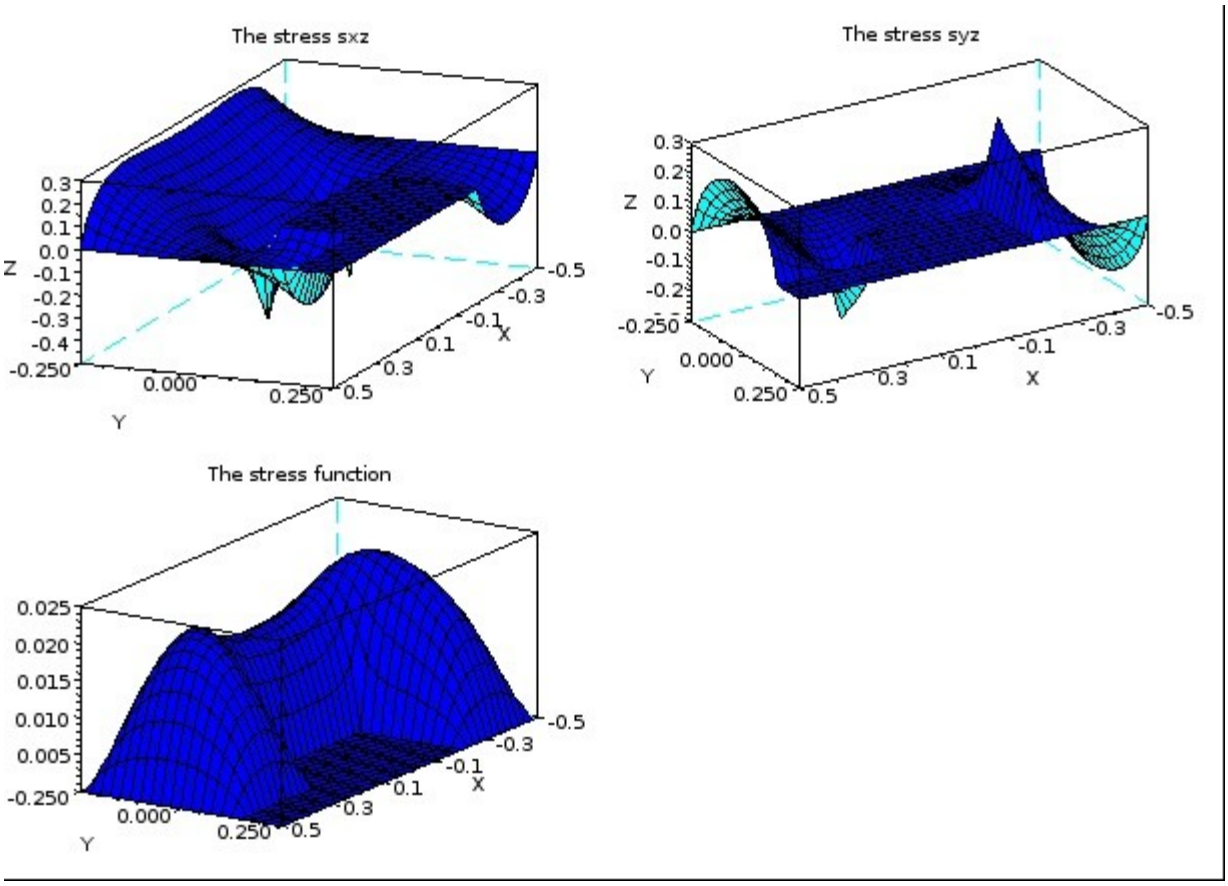

Figure 6. Torqued uneven U shaped cross section. $y_{c1}=0.5b$, $y_{c2}=0.8b$, $x_{c1}=0.3$, $x_{c2}=0.7$. M=41, N-21, # of iterations k=120.

For the uneven U shaped sections with data given in Figure 6 the resulting J integrals are = 0.00846 for the stress integration and = 0.00859 for the stress function integration. Note that in this and the following example we have reduced the number of grid points in both x and y. This is not quite as accurate and explains the slightly larger relative difference between the two J integral calculations. But it is still accurate enough for illustrative purposes. Do remember that for higher accuracy one has to both reduce the convergence criteria for the relaxation process and reduce dx and dy (increase N and M) appropriately.

Example 4: This example is for an uneven I beam. The I beam is represented by the U shaped section of Figure 2 and Figure 3 turned 90° clockwise. Then the lower flange of the I beam is represented by the right leg of the U. By reflecting the U across the lower boundary j=1, one obtains the I beam cross section. Mathematically the reflection is accomplished by simply imposing the symmetry condition

(19) $$\left[\frac{\partial \phi}{\partial y}\right]_{y=-0.5}=0$$

instead of the condition $\phi(x,-0.5)=0$ for the U and L shaped cross sections. Condition (19) translates into

(20) $$f_{i,2}=f_{i,0} \quad .$$

But $f_{i,0}$ does not exist in the finite difference grid. It represents a point below the boundary y=-0.5. Thus we evaluate equation (14) on the boundary j=1 (y=-0.5) yielding

(21) $$f_{i,1}^{k}=\frac{1}{2+2r^2}[f_{i-1,1}^{k}+f_{i+1,1}^{k-1}+r^2(f_{i,0}^{k}+f_{i,2}^{k-1})+2dx^2] \quad .$$

$f_{i,0}$ in equation (21) is eliminated by equation (20) to yield

(22) $$f_{i,1}^{k}=\frac{1}{2+2r^2}[f_{i-1,1}^{k}+f_{i+1,1}^{k-1}+2r^2 f_{i,2}^{k-1}+2dx^2] \quad .$$

The only modification that must be made in the Program to the solution procedure for solving equations (15) is to add, before the *two for loops* that execute a relaxation pass, the calculation of equations (22) for i=2:N-1 and j=1. This is shown in the relaxation iteration part of *ProgramULS* and is selected by the input variable Ks=0. The only other modification that needs to be made is to include the boundary integration in formula (7). This boundary integration reduces to the single line integration along the boundary y=-0.5 as

(23) $$\int_{x=-0.5}^{x=0.5} y\,\phi\,\vec{n}\cdot\vec{j}\,dx$$

where y=-0.5 and $\vec{n}=-\vec{j}$ because ϕ is not zero only along the boundary y=-0.5. The numerical integration of (23) should be evident in *ProgramULS* at the end of the stress function integration.

In this example we assume for the I cross section the same dimension as the basic uneven U section of Example 3, but now we really have an uneven I section laid on its side. Figure 7 below shows the stresses and the stress function for this case. Note in this Figure 7 that the stress function is not zero at the boundary y=-0.5. The stress σ_{xz} is now zero along the boundary y=-0.5 as it should be if the stress function is an even function across y=-0.5. The stress σ_{yz} is now not zero along this boundary as expected. In this case the J integrals are 0.00884 by stress integration and 0.00859 by stress function integration. But recall that these values are only for half the I beam cross section because we have only computed over half the cross section by use of symmetry. These values should be doubled for the actual I cross section. Note also that the calculation required 330 iterations compared to the U cross section case of the same dimension. This is because of the symmetry condition at the boundary y=-0.5. This is actually a Neumann boundary condition that always slows the convergence rate of the relaxation procedure. (Algebraically in terms of Gauss-Seidel iteration it makes the system a bit less diagonally dominant, which hurts convergence rates.)

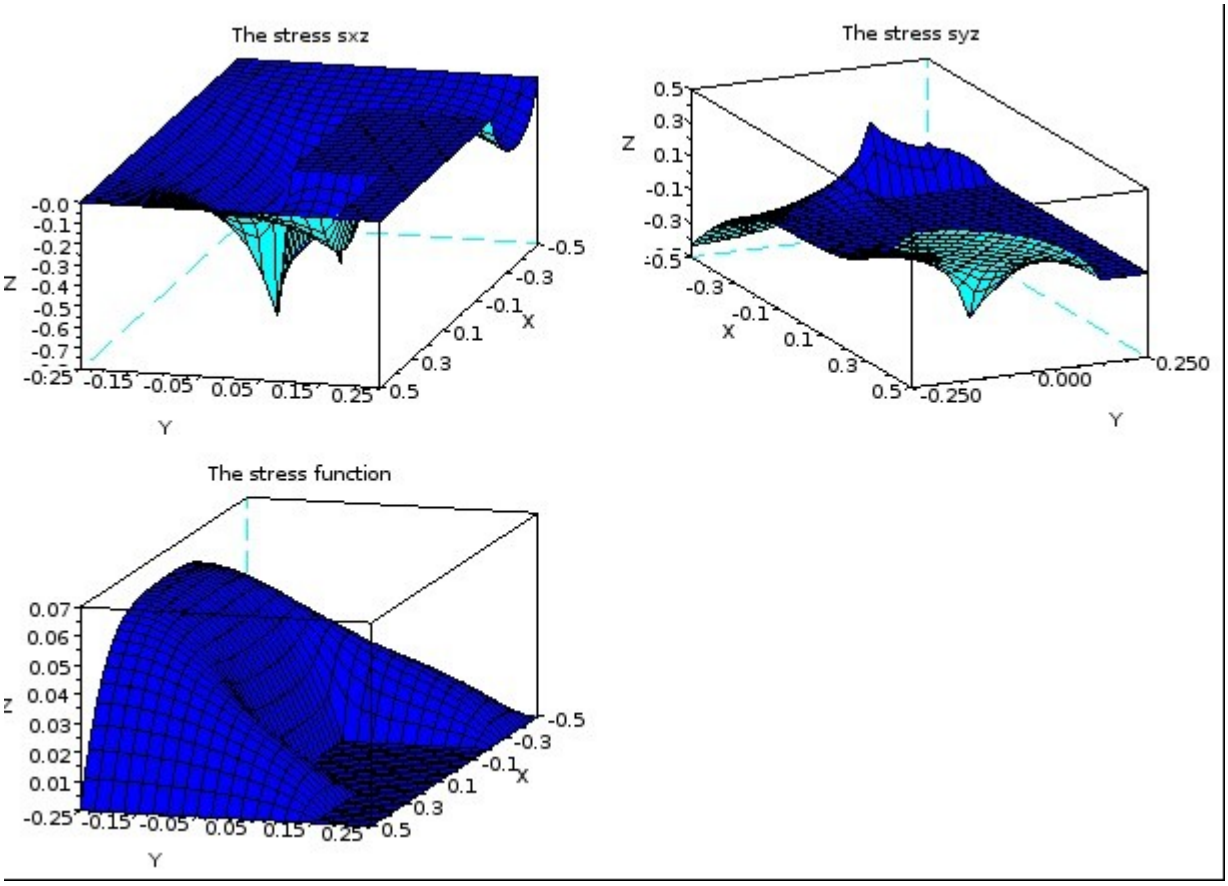

Figure 7. The uneven I beam cross section (reflect it about the y=-0.5 axis). Here yc1=0.5b, yc2=0.8b, xc1=0.3, xc2=0.7 . N=41, M=21.

5. Concluding Remarks:

The program *ProgramULS* can be used to compute a variety of interesting situations that can be quite instructional. For example, I have not presented a T beam example. I leave that to the reader to explore. Furthermore, an interesting thing to do with this program is to examine thin walled cross sections and compare the results to the approximate theories that have been developed for such cross sections and are the stuff of elementary Mechanics of Materials. Just remember to use a fine enough grid that will result in enough grid points inside the thin walls to resolve the stresses accurately.

There are some relatively simple modifications that can be made to explore some other geometries. I *challenge* the reader to start with the simple rectangle program, *ProgramR*, to examine trapezoidal cross sections as shown in Figure 8 below.

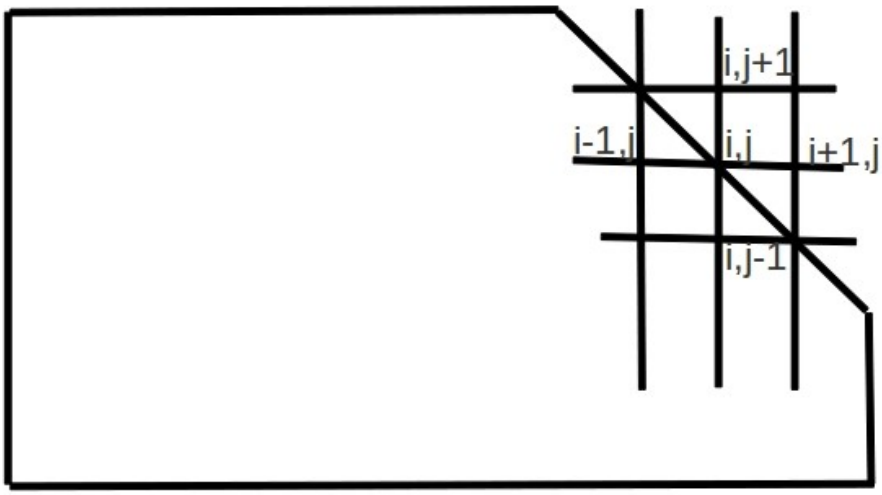

Figure 8. A trapezoidal cross section showing a few boundary grid points on the diagonal boundary. In particular the grid points surrounding the point i,j on the diagonal boundary.

If one arranges the the grid to lie on the diagonal boundary as in Figure 8, the iteration method to solve the finite difference equations (15) remains the same. Only the blanking array q(i,j) has to be altered in an evident way. Also the boundary stress calculations on the diagonal boundary must be included, but the boundary formulas for the derivatives in the y and x directions remain those of (18) and (19) applied at the appropriate diagonal grid points. One need not have any trepidation about achieving the diagonal boundary lying directly on the grid points since one can adjust the relative x to y grid spacing r to accommodate any angle of the diagonal. But be careful about to large or small angles of the diagonal because this might result in a large or small value of r that reduces the accuracy of the finite difference formulas. Of course you can also use the symmetry condition to have a more complicated trapezoid.

Have fun playing around with the above, and I am sure you will also find it interesting and educational.

Reference:

[1] Timoshenko, S.P. and Goodier, J.N. **Theory of Elasticity**, McGraw-Hill Book Company, New York, (1970).
[2] Soutas-Little, **Elasticity**, Dover Publications, Inc. Mineola, New York. (1999).