Hydrostatic and Stability Calculations with MATLAB

Adrian B. Biran[1]

August 29, 2006

Abstract

This paper shows how MATLAB can be used in Naval Architecture for practical hydrostatic and stability calculations and for teaching these notions. The applications exemplified in the paper include software for digitizing curves of statical stability or borders of plane figures, and for calculating the properties of the areas bounded by the digitized curves.

1 Introduction

1

MATLAB is a high-level programming language and a computing environment. The name is an acronym for *Matrix Laboratory.* Initially meant for solving problems in Linear Algebra, MATLAB soon proved its efficiency in Control and Electrical Engineering, and shortly afterwards in Mechanical and practically all other branches of engineering. The basic data type is the array. Operations on the elements of arrays can be performed directly, without the need to write loops as in other programming languages. Operations between arrays can be performed as simply as between numbers. The package provides many mathematical functions. In addition there are many functions that allow immediate plotting, and there are functions that permit very sophisticated plots. Programming loops or conditional branching is simple. All these features enable rapid prototyping, but also the writing of complex programs. In addition to the basic package there are toolboxes with powerful functions for specialized fields. A few of these additions are SIMULINK, a toolbox for simulating systems described by block diagrams, and toolboxes for control systems, optimization, signal processing, statistics, and symbolic mathematics.

MATLAB has been used in Naval Architecture. There is ample evidence of hydrodynamic and ship-control calculations in MATLAB. As an outstanding example, Thor Fossen (1994) uses MATLAB throughout his book on *Guidance and Control of Ocean Vehicles.* The author of this paper has used MATLAB

[1]Technion, Israel Institute of Technology, Faculty of Mechanical Engineering

for many years for quick hydrostatic and stability calculations, and for post-processing the output of university or commercially available hydrostatic programs. Such applications are illustrated throughout the book *Ship Hydrostatics and Stability* (Biran, 2005) and the set of corresponding files can be found on the sites of the book provided by *Elsevier*, the publisher of the book, and by *The Mathworks*, the distributers of MATLAB.

Programs for Naval-Architectural calculations have been available for many years, both commercially and in universities. Such programs enable the user to design a hull surface, or at least to input the offsets of an already designed hull surface, and to perform hydrostatic calculations for a ship having that hull. As happens with all programs, they can only do what the programmer 'taught' them to do. In certain situations a Naval Architect may be interested in carrying on other calculations than those for which the software was programmed. Then, MATLAB can help. Examples of its uses are the reading and pre-processing of offsets, or the post-processing of the output for comparison with stability criteria that were not included in the program.

Another category of users that may benefit from using MATLAB is that of students that have to solve exercises, carry on projects and especially those that cannot afford a commercially available program. A student's version of MATLAB is available and it can be used for most courses delivered in a technical university. As this article shows, students can write in MATLAB their own programs for hydrostatic calculations. Years ago, this author gave this assignment as a two-semester project. The students who carried it on were enthusiastic and the results calculated by them compared well with those yielded by a program well proven in universities and industry. As the MATLAB facilities have been considerably enhanced since then, today it should be possible to achieve much better performances.

Some of the advantages of using MATLAB in Naval Architecture are:

- the possibility of adding functions and programs written by the user;
- easy accessibility of data;
- the transparency of MATLAB programs.

Usually, commercially-available programs are black boxes for the user, while programs written in MATLAB are transparent. The user can understand the mathematics behind them and the coding, and, if necessary, modify the software and add new features. This possibility can be important in universities, both for students and for researchers.

We begin this article with a simple example of stability calculation that presents immediately the basic features of MATLAB and shows how simple is its use. Next, we use MATLAB for standard integrations, an ubiquitous subject in Naval Architecture. We continue by showing how to write a short program for checking compliance with a code of stability. We go on to show how MATLAB enables us to write demos and *Graphic User interfaces*, shortly *GUIs*, that can be used as teaching aids. We give two examples, one that explains the notion of *metacentric evolute*, the other details the meaning of the *curve of statical stability.* We continue by showing how MATLAB can be used to digitize and to integrate over the digitized data. A first example is that of a program for digitizing curves of statical stability and calculating the areas under the curve. As a second example,, we describe a program that digitizes the border of a plane figure and calculates its area, centroid, principal axes, and moments of inertia.

2 A simple example of stability calculation

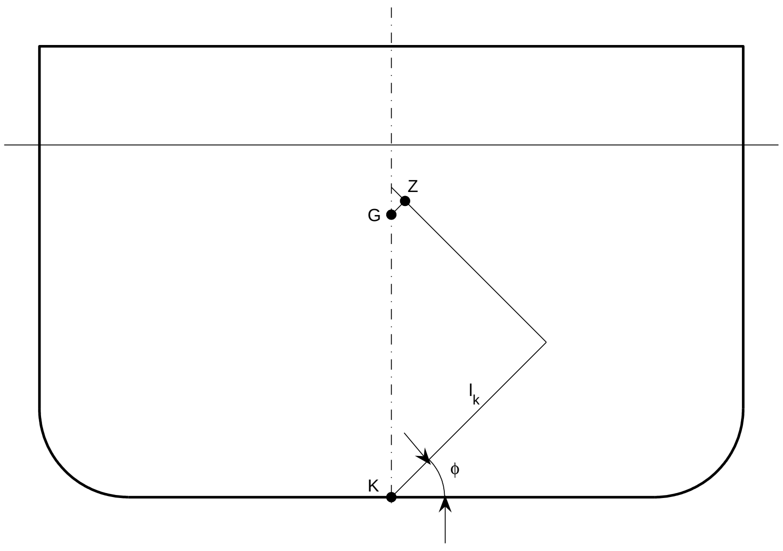


Figure 1: Definitions used in stability calculations

The following example is similar to that presented in Biran and Breiner (2002); it allows us to introduce a few of the characteristic features of MATLAB and to show how easy it is to use this software for routine calculations and plotting. Let us assume that for a given ship we have the data listed below and we are asked to calculate and plot the curve of statical stability and the curve of the heeling arm in turning. The given displacement is $\Delta = 3900$ t, the length between perpendiculars, $\mathrm{L_{pp}} = 75.95$ m, the mean draft, $\mathrm{T_m} = 5.96$ m, the arm of free-surface effect, FS = 0.03 m, the metacenter above baseline, $\overline{KM} = 5.35$ m, the vertical center of gravity, $\overline{KG} = 4.78$ m, and the cross-curves values at the given displacement (arms of stability of form), l_k, are shown in Table 1. Figure 1 explains how some of these points and the lever l_k are defined.

We first write the data at the command prompt, $\gg$:

```
Lpp = 75.95;
Tm  =  5.96;
```

Table 1: Cross-curves values

10^o	20^o	30^o	45^o	60^o	75^o	90^o
0.936	1.823	2.610	3.708	4.540	4.931	4.931

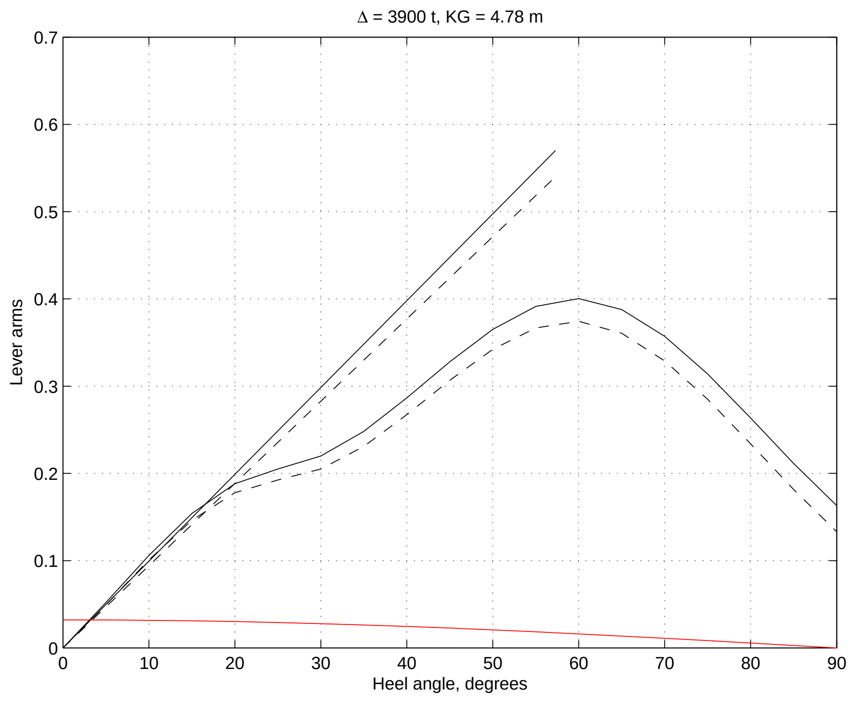


Figure 2: Checking stability in turning

```
Displ = 3900;
KM = 5.35;
KG = 4.78;
FS = 0.03;
```

If we do not enter a semi-column, ';', after each input, MATLAB returns us the input. This may be desirable for more complex data, such as matrices. In continuation we enter the heel angles in an *array* delimited by square brackets, '[ ]', and called `heel`,

```
heel = [ 0 10 20 30 45 60 75 90 ];
```

Similarly we enter the cross-curve values in an array `lk`,

```
lk = [ 0 0.936 1.823 2.610 3.708 4.540 4.931 4.9431 ];
```

We calculate the righting arm ,$\overline{GZ}$, with the command

```
GZ = lk - KG*sind(heel);
```

Here we call the new MATLAB function `sind` that yields sine values for arguments measured in degrees. In the statement shown above we use a remarkable feature of MATLAB, *vectorization.* In a single line we defined the multiplication of an array by a scalar, and the subtraction of two arrays. In this case the arrays are *row vectors.* In other computer languages, these operations are carried on repetitively, in *loops.*

The separation of the values stored in the arrays `heel` and `GZ` is too wide to obtain a good plot. Therefore, we use the MATLAB function `spline` to carry on a cubic-spline interpolation over the given values. To do so we first define an axis of closer-spaced angle values

```
 heeli = 0: 5: 90;
```

where the result of the command is an array of numbers beginning with 0, with the increment 5, and ending with 90. Now we call the function `spline` with three arguments

```
  GZi = spline(heel, GZ, heeli);
```

and plot the interpolated righting-arm values against the axis of closely-separated angle values

```
 plot(heeli, GZi, 'k-'), grid
```

Above, the plotting argument `'k-'` says that the curve should be plotted in black as a solid line. This argument is supplied between quotes, '', that is as a *string argument.* The command `grid` superimposes a grid over the plot. We add now a title and label the axis with the commands

```
  title('\Delta = 3900 t, KG = 4.78 m')
  xlabel('Heel angle, degrees')
  ylabel('Lever arms')
```

The command `\Delta` is taken from the LaTeXdesk-publishing package and produces the Greek letter Δ, that is the displacement symbol. We want to add now the tangent in origin. To tell MATLAB that we want to superimpose this line over the existing plot, and not begin a new plot, we enter the command

```
   hold on
```

Next, we calculate the metacentric height and we plot a line with one $x-$coordinate in the origin, 0, and the other at 1 radian, that is at $180/\pi$. We use here the MATLAB command `pi` that yields the value of π :

```
 GM = KM - KG;
 plot([ 0 180/pi ], [ 0 GM ], 'k-')
```

Until now we neglected the free-surface effect. We take it into consideration with the following commands that calculate the *effective righting-arm*, the *effective metacentric height*, and plot the respective curve and its tangent in origin in black, dashed lines:

```
GZeff = GZi - FS*sind(heeli);
plot(heeli, GZeff, 'k--')
GMeff = GM - FS;
plot([ 0 180/pi ], [ 0 GMeff ], 'k--')
```

To calculate the heeling arm in turning, l_T, we begin by converting the service speed of 16 knots to ms^{-1}, apply the formula prescribed by the IMO code of safety, and plot in red the heeling-arm curve over the curve of statical stability.

```
V0 = 0.5144*16;
lT = 0.02*(V0^2/Lpp)*(KG - Tm/2)*cosd(heeli);
plot(heeli, lT, 'r-')
```

The complete plot appears in Figure 2. Instead of trying to appreciate visually the value of the angle of statical stability, at the intersection of the l_T and $\overline{GZ}_{eff}$ curves, we can use another MATLAB feature, the `ginput` function

```
  [ phi_st GZst ] = ginput(1)
```

A crosshair appears over the curve. We drag it with the mouse over the intersection of the curves and click the left-button. We read an angle of 3.2 degrees. The readers may find slightly different values. If the reader wants to continue with other plots, it is necessary to stop plotting over the same figure. To do this we enter the command

```
hold off
```

3 Numerical integration

Naval Architects have to integrate many functions defined by a small number of tabulated values and not by analytic expressions. Among the numerical methods used for this, the trapezoidal and Simpson's rule are the favorites. The subject is so characteristic to Naval Architecture that traditional books usually begin with it. In Biran (2005) we preferred to delay the treatment to Chapter 3, after motivating the reader by showing why numerical integrations are necessary. MATLAB has several functions that perform integration. To exemplify one of them, `trapz`, and also a few additional features of MATLAB, let us suppose that we have to calculate the area of the design waterline of a given ship, in this case the model called *Carena 752 bis* in INSEAN (1963). We read the offsets of the waterline in the body plan. The length between perpendiculars is divided into 20, with a few additional intermediary stations. To calculate the interval of integration we enter the length between perpendiculars and divide it by 20

```
  Lpp = 120.395;
   dL  = Lpp/20;
```

Next, we enter the numbers of the stations in an array called `stations`. Thus, for station -1/2 we enter -0.5, for the station corresponding to the aft perpendicular we enter 0, for the station corresponding to the forward perpendicular we enter 20.

```
 stations = [ -0.5 0 0.5 1 1.5 2:18 18.5 19 19.25 19.5 19.75 20 ]
```

To spare work, by the middle of the above array we wrote `2:18`. This expressions produces the numbers 2, 3,...18. Now we calculate the distances of stations from AP, that is their $x-$coordinates,

```
  x = dL*stations;
```

To obtain the values of the offsets measured in the given body plan we must calculate the scale. The given molded beam, B, is 17.078 m, while its value measured in the body plan is 203 mm. We calculate

```
  scale = 17.078/203
```

We enter the measured offsets, in mm, in an array called `measured` and we obtain the offsets, in m, by multiplying this array by `scale`

```
measured = [ 10.5, 28, 42, 55, 66, 75, 88.5, 96.5, 100, 101.5,...
   101.5, 101.5, 101.5, 101.5, 101.5, 101.5,...
   101.5, 101.5, 101.5, 100, 92.5, 75, 62, 46, 37.5, 28, 19, 10 ];
offsets  = scale*measured
```

Above the array was to long to be entered in one line. Therefore, we used the *continuation device*, '...'. If we want to check for input errors we may plot the offsets against their distances from AP. To force the same scale on the two axes we use the command `axis equal`

```
 plot(x, offsets), axis equal
```

Finally, we use the MATLAB function `trapz` to calculate the area of the design waterline by means of the trapezoidal rule. In this case we call the function with two input arguments, the distances from AP, that is the $x-$coordinates, and the offsets, that is the $y-$coordinates.

```
  Aw = 2*trapz(x, offsets)
  Aw = 1.7966e+003
```

and obtain the area 1796.6 m^2. To check the result we may calculate the waterplane coefficient

```
    Cw = Aw/(Lpp*17.078)
```

where 17.078 is the given molded beam. Our result is 0.8738, the one appearing in the INSEAN publication is 0.839.

We spent some time in measuring and calculating the offsets. The more calculations we have to perform with these numbers, the more our investment is worth. For example, let us suppose that we must calculate also the *Longitudinal center of flotation*, *LCF*. We calculate the moments of the various stations, we integrate them over the ship length and divide by the corresponding area

```
Moments = x.*offsets;
LCF = trapz(x, Moments)/trapz(x, offsets)
LCF = 59.4923
```

We can check our result by calculating *LCF/Lwl* and comparing it with the value published by INSEAN

```
Lwl = 125.14;
LCF/Lwl
ans =
  0.4754
```

INSEAN gives the value 0.490.

Above we used the trapezoidal rule that does not impose restrictions on the spacing of stations. When the offsets are given for pairs of equally-spaced stations, one can use Simpson's rule. A function that performs integration by Simpson's rule, for functions defined in tabular form, is described in Biran and Breiner (2002, pp. 496–7).

4 Checking compliance with codes of safety

The stability of ships and other floating bodies must be checked in accordance to regulations issued or adopted by the bodies that approve their design or permit them to sail. CAD systems for ship design can include procedures for doing the job. If the available program does not contain such a procedure, or it is necessary to use a criterion differing from that provided, the Naval Architect has to check the stability manually. As MATLAB allows quick prototyping, it is easy to write programs for any given criteria of stability. Biran (2005), pp. 233-6 , describes an example tailored for the stability regulations BV 1003 of the German Navy. In this section we show how to write a *script file* that checks the stability of sail vessels according to the UK Maritime and Coastguard Agency (Maritime 2001). The philosophy behind this code is explained in Biran (2005, pp. 192-4).

To explain the code of practice we refer to Figure 3, which shows the $\overline{GZ}$ curve of the same training yacht that is exemplified in Biran (2005). The code assumes that the wind arms are proportional to $\cos^{1.3}\phi$, where ϕ is the heeling angle. Given the flooding angle, ϕ_d, we assume that this angle is reached under

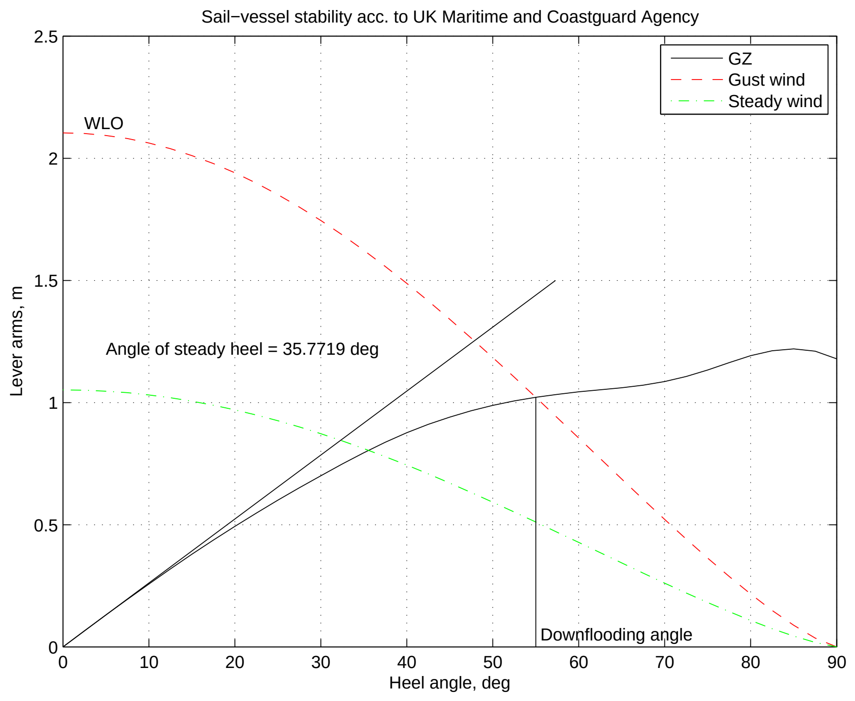


Figure 3: Checking the stability of a sail vessel

a gust wind. If the downflooding angle is not given or is greater than 60°, it is assumed equal to 60°. The gust-wind arm at zero heel is calculated as

$$WLO = \frac{GZ_f}{\cos^{1.3} \phi}$$

where GZ_f is the righting arm at the angle ϕ_d.

Further, the code assumes that the gust-wind speed does not exceed 1.4 times the steady-wind speed. Therefore, the steady-wind arm at zero heel is $WLO/2$. The resulting curve

$$\frac{WLO}{2} \cos^{1.3} \phi$$

intercepts the $\overline{GZ}$ curve at an angle that we call *angle of steady heel*. If the master of the ship sails so that the heeling angle is less than the angle of steady heel, the heel angle under a wind gust will not reach the downflooding angle.

Listing 1, at the end of this article, shows a function that checks stability according to the above criterion. The listing should be written on a file `SailCode.m`. The function can be called with one output and three or four input arguments

```
SteadyHeel = SailCode(stability, KG, GM, downflood)
```

Obviously, the user can choose other argument names, but must respect their order. For the training yacht exemplified in Figure 3 the first input argument is a *2-by-n* array of heel angles, in degrees, and cross-curve values, in m, that is written in MATLAB as

```
stability = [ 0     5       10      ...     85      90
              0     0.308   0.611   ...     3.249   3.216 ];
```

The other data are *KG = 2.037* m, *GM = 1.5* m, and the downflooding angle 55^o.

The first line of the code states that the file contains a function, that it has one output argument, that the name of the function is `SailCode`, and that it can be called with four input arguments. In continuation, a number of lines are marked as *comments* by means of the percent symbol, '%'; they form the *help* part of the function. Typing `help SailCode` on the command line produces a display of the help.

The statement

```
heel   = stability(1, :);
```

creates an array, `heel`, consisting of the first line, all columns of the array `stability`, that is an array of heel-angle values. The next statement creates an array, `l_k`, consisting of the second line, all columns of the array `stability`.

The command

```
heeli  = 0: 2.5: heel(end);
```

creates an array of interpolating heel angles, from 0 to the maximum heel angle, with 2.5-degree intervals.

The command

```
GZi    = spline(heel, l_k, heeli);
```

interpolates over the function *l_k(heel)* at the points defined in the array `heeli`.

In continuation we see a *conditional construct*. It says that if the number of input arguments is smaller than 4, that is if the dowflooding angle is not defined, then its value is set by default to 60^o.

The plotting and labelling statements may be self-evident. Let us explain the line

```
[ phi_st, GZst ] = ginput(1);
```

A haircross appears on the screen and the user has to bring it over the intersection of the righting-arm and steady wind-arm curves. By pressing the left mouse button the user picks up the angle value, `phi_st`, and the righting-arm value at this angle. The former is displayed on the plot by a `text` statement. As the picked-up value is numerical, it is first converted to a string of characters by the command `num2str` and is next displayed on the plot by the command `text`.

5 Teaching aids

5.1 A demo that describes the B and M_curves

Two great features of MATLAB can be used for writing teaching aids. One is the possibility of animating graphs. The other is the possibility of writing *Graphic User Interfaces*, shortly *GUIs*.

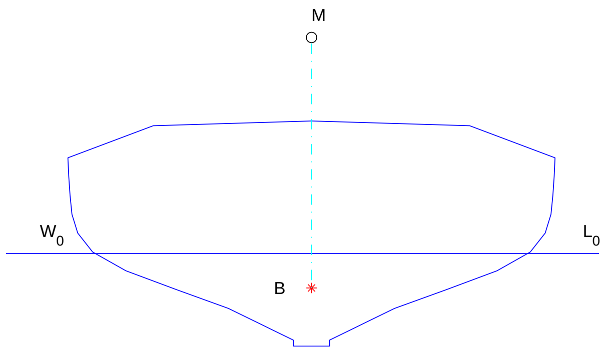


Figure 4: Displaying B and M in upright condition

In this section we describe a demo that shows the evolution of the center of boyancy, B, and of the metacenter, M, while the ship heels at constant displacement, between 0 and 90 degrees. The full program, `B_Curve`, and two auxiliary files are shown in Subsection 10.2 as *Listings 2*. We call *B-curve* the projection on a transversal section of the trajectory of the center of buoyancy, and *M-curve* the projection on the same plane of the trajectory of the metacenter. The data that produced the Figures 4 to 6 belong to an actual vessel that was also exemplified several times in Biran (2005). Figure 4 shows the initial display, that is the situation in upright condition. Pressing `Enter` starts a sequence of displays of the water line and of the points B, and M for 15^o, 30^o, $\ldots 90^o$, heel angles. The complete picture appears in Figure 5. Pressing once again `Enter` connects the various metacenters by a yellow line, as in Figure 6.

Let B_ϕ be the center of buoyancy, and M_ϕ the metacenter corresponding to the heel angle ϕ. The sequence of displays shows that the normals $\overline{B_\phi M_\phi}$ to the B-curve in B_ϕ are tangent to the M-curve in M_ϕ. In other words, the M-curve is the *evolute* of the B-curve and, therefore, it is called *metacentric evolute*.

The program `B_Curve` must be written on a file `B_Curve.m`. It calls two other files, `lido9.m` and `rotate.m` that must stay in the same folder as the calling file. The file `lido9.m` stores the data of the vessel exemplified by the demo. The file `rotate.m` contains a function that performs rotation of the coordinate axes. It is possible to include the vessel data in the main file. We preferred to use a separate file and allow thus to run the demo with the data of other ships. As to the file `rotate.m`, this a function of general use and it is worth keeping it separately. In fact, we reuse this function in the GUI described in the next

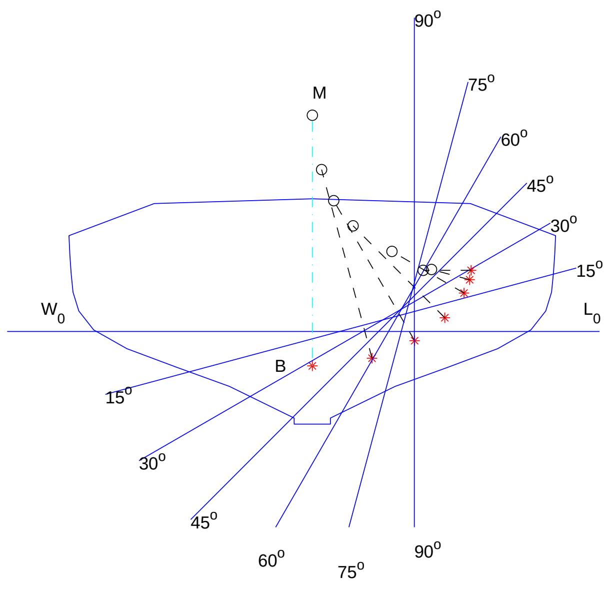


Figure 5: Displaying the B_curve and the corresponding metacenters

subsection.

In the listing `B_Curve` we meet a new MATLAB feature. The line

```
Hf_fig = figure
```

retrieves the *handle* `Hf_fig` assigned to the *object* called *figure*. This handle is used in the next line to assign to the property *NumberTitle* of the object *figure* the value *off*, and to the property *Name* the value *B and M curve*. This means that instead of the default title *Figure 1*, the figure window will display the title *B and M curve*.

The command `lido9` *loads* the data stored in the file `lido9.m`. The command that separates the display into several frames is `pause`. We meet it at the end of the program segment PLOT MIDSHIP SECTION. The program run stops here enabling the user to view the display corresponding to the upright condition. To go further the user has to press `Enter`. Then, the program enters a *For-loop* that is repeated $k - 1$ times, where k is the number of heel angles stored in the array `heel`. The lines to be executed are contained between the statements `for l = 2:k` and `end`. At each repetition, $l = 2,\ 3\ \ldots k$, the program performs operations involving the l-th elements of the called arrays, `x0, yo, heel,` etc. The command `pause(1)` written before `end`, causes the program to stop for one second before executing the next repetition. After completing l repetitions, a new command `pause` freezes the display until the user presses again `Enter` Then, the last program segment displays the metacentric evolute and the text 'The M-curve is the evolute of the B-curve.'

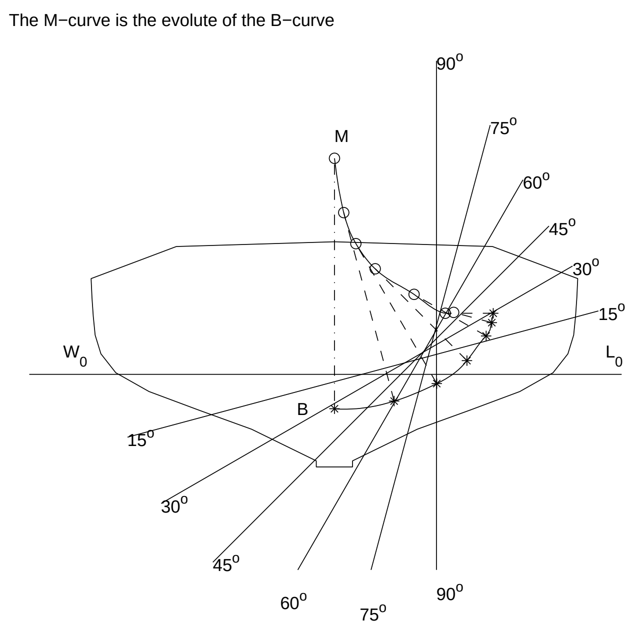


Figure 6: Displaying the B and M_curves

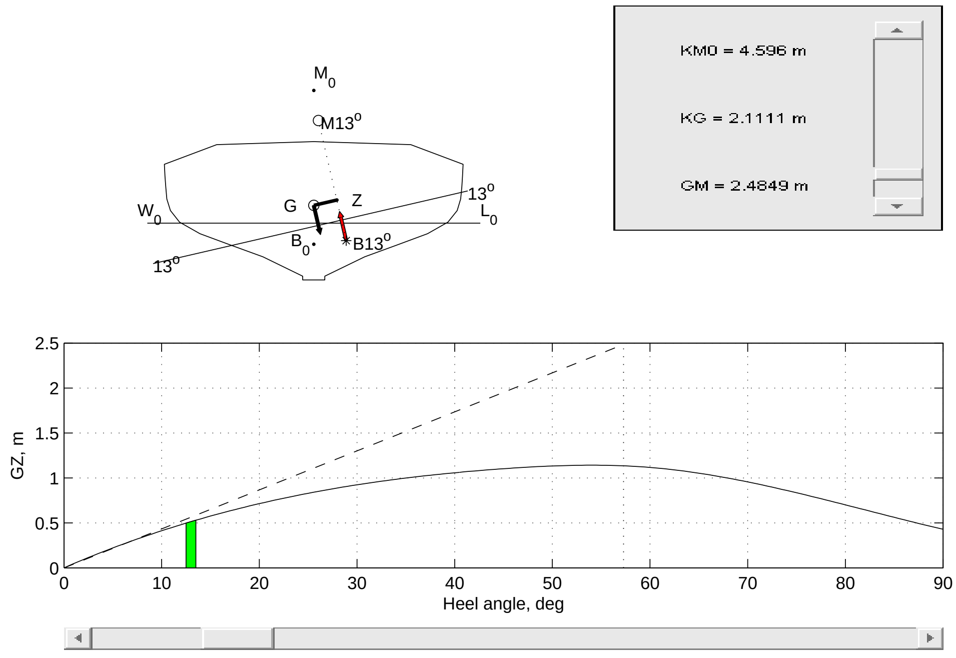
5.2 A GUI that explains the curve of statical stability

With the tools of the basic MATLAB package it is possible to build graphical user interfaces composed of control and display units. The interactive control is carried on by means of menus, buttons, sliders and editable text boxes. The display blocks are plotting windows and text boxes.

As an example we describe a GUI, `GZdemo`, that explains the meaning of the curve of statical stability. One instance appears in Figure 7. The plot in the upper, left-hand side of the figure shows the midship outline of the ship exemplified in the preceding subsection, the waterlines in upright condition and at 13° degrees of heel, and the center of buoyancy in upright condition, B_0, the center of buoyancy at 13° heel, B_{13^o}, the center of gravity, G, metacenter in upright condition, M_0, and the metacenter at 13° heel, M_{13^o}. The righting arm at 13° is $\overline{GZ}$.

The lower part of the figure shows a plot of the curve of statical stability and its tangent in the origin. A colored *patch* shows the value of the righting arm at 13°.

Under the curve of statical stability we see a *slider*. To move this control element the user can either pick up the slider with the mouse and drag it, or click on the arrows at the two slider extremities. By doing so the user changes the heel angle. Figure 8 shows the display after dragging the slider to 30°. The position of the waterline, of the center of buoyancy, of the metacenter and of the point Z changes also in the upper, left-hand plot.



KM0 = 4.596 m

KG = 2.1111 m

GM = 2.4849 m

Figure 7: A GUI that explains the curve of statical stability, 13-degree situation

Another slider is shown at top, right. By dragging this slider or clicking on the arrows at its extremities, the user can raise or lower the center of gravity. The display on the left of the slider shows the height of the metacenter above baseline, in upright condition, $\overline{KM}_0$, the vertical center of gravity, $\overline{KG}$, and the metacentric height, $\overline{GM}$.

To appreciate the effect of a $\overline{GM}$ change, we refer to Figure 9 in which the heeling angle is again 30^o, but the metacentric height is reduced to 1.616 m. The reader is invited to examine the changes.

It is possible to program a GUI by writing explicitly its code. The MATLAB package, however, includes also a *GUI Layout Editor* appropriately called *GUIDE* that simplifies the work.

6 Developing a digitizer

6.1 Why digitizers

Many engineering problems require the coordinates of points on a drawing, map or graph. Typical examples can be found in Naval Architecture where one needs the offsets of a ship to perform hydrostatic calculations, or points on a curve of statical stability to calculate the area under it. Before the advent of computers such data were measured manually. When computers were introduced in engi-

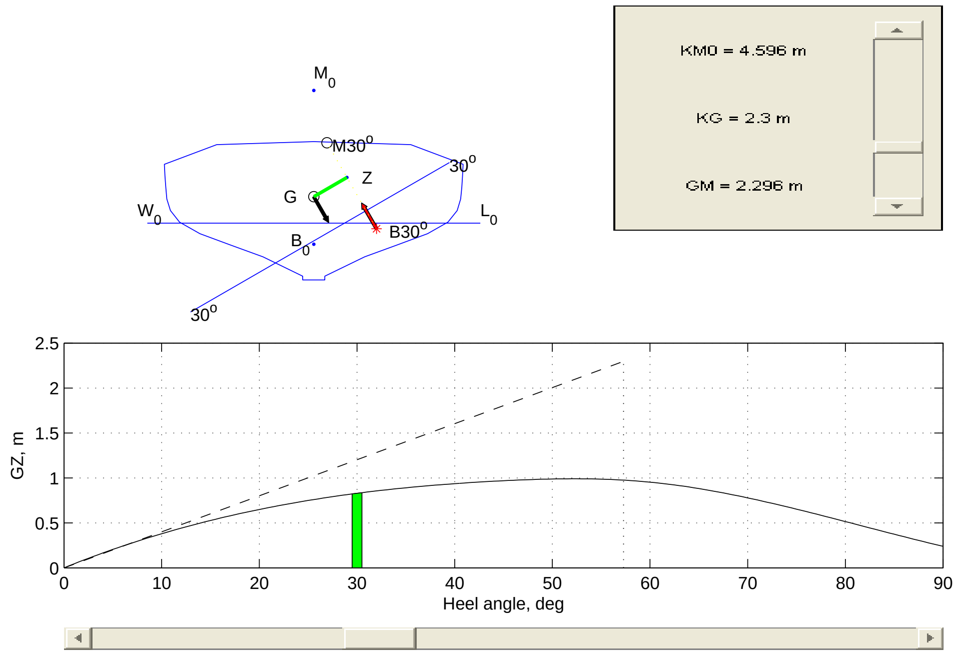


Figure 8: A GUI that explains the curve of statical stability, 30-degree situation

neering, the task was performed with the aid of instruments called *digitizers*, or *digitizing tablets*, and appropriate software. This software was part of many CAD packages.

The web describes or advertises software designed for digitizing features appearing in graphic files. To mention just two examples we can refer to the *Engauge Digitizer - Digitizing software* (http://digitizer/sourceforge.net/), or to *Digitize* (http://www.rockware.com/catalog/pages/digitize.html).

MATLAB enables us to import a graphic file containing the lines or graph that must digitized and to pick up the coordinates of points on the lines we are interested in.

6.2 Importing and processing images in MATLAB

Graphic files can be produced by scanning drawings, maps or graphs. The resulting file can be read into the MATLAB workspace by using the command `A = imread('myfile.jpg')`, where instead of '`myfile`' the actual file name will appear. We obtained good results with the format `jpg`. Therefore, we use this file type. When the file is in the workspace it can be displayed with the command `imag(A)`.

MATLAB provides functions for processing the imported file. What interests us is that we can pick up points of the image and we can print on the image. The next subsections show how to do this.

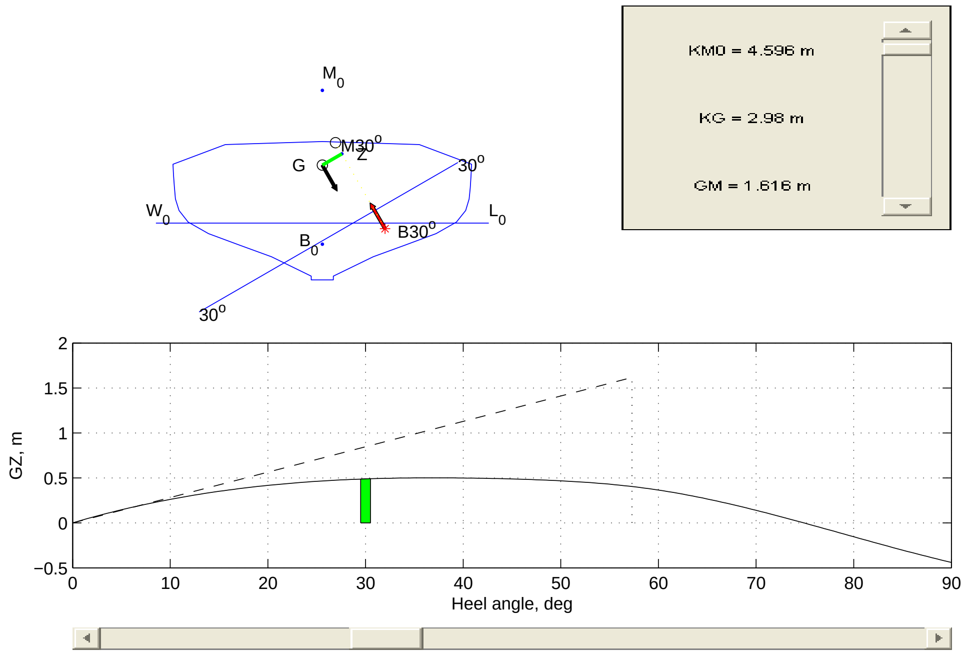


Figure 9: A GUI that explains the curve of statical stability, the situation after reducing the metacentric height

6.3 The digitizer

Let us suppose that we want to digitize a body plan. After displaying its image we start by *calibrating* it. In this phase we pick up, with the mouse, three non-collinear points whose actual coordinates are known and input these coordinates. The computer uses the above information to find the axes and the scale of the figure. Next, we pick up the points we want to digitize. To explain the mathematics involved we use the qualifier '*real-world*' for the true coordinates of a point, and '*screen*' for the coordinates of the same point measured in the image to be digitized.

Let $x_i,\ y_i,\ i\ \in\ [1,\ 2;\ 3]$ be the real-world coordinates of the calibrating points, and $\xi_i,\ \eta_i$ their screen coordinates. Assuming that the sets x_i, y_i and $\xi_i,\ \eta_i$ are related by affine transformations, namely translation, rotation and scaling, we write

$$\begin{aligned} a_1x_1 + a_2y_1 + a_3 &= \xi_1 \\ b_1x_1 + b_2y_1 + b_3 &= \eta_1 \\ a_1x_2 + a_2y_2 + a_3 &= \xi_2 \\ b_1x_2 + b_2y_2 + b_3 &= \eta_2 \\ a_1x_3 + a_2y_3 + a_3 &= \xi_3 \\ b_1x_3 + b_2y_3 + b_3 &= \eta_3 \end{aligned} \tag{1}$$

Putting these equation in matrix form we write

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \\ a_3 & b_3 \end{vmatrix} = \begin{vmatrix} \xi_1 & \eta_1 \\ \xi_2 & \eta_2 \\ \xi_3 & \eta_3 \end{vmatrix} \tag{2}$$

or, shorter,

$$\mathbf{XC} = \mathbf{S} \tag{3}$$

where **X** is the matrix of homogeneous real-world coordinates, **C** is the matrix of transformation coefficients, and **S**, the matrix of screen coordinates.

Knowing the screen and the real-world coordinates of three non-collinear points, the coefficients a_i, b_i can be found by Gaussian elimination. In MATLAB this operation is written as

```
C = X\S
```

where the *backslash* operator, ‘ \’, symbolizes division at left.

After calibration, the user may digitize as many points as necessary, let's say n. The screen coordinates of those points are stored in a 2-by-n array and are related to the real-world coordinates by

$$\begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} \begin{vmatrix} x_1 & x_2 & \dots & x_n \\ y_1 & y_2 & \dots & y_n \end{vmatrix} = \begin{vmatrix} (\xi_1 - a_3) & (\xi_2 - a_3) & \dots & (\xi_n - a_3) \\ (\eta_1 - b_3) & (\eta_2 - b_3) & \dots & (\eta_n - b_3) \end{vmatrix} \tag{4}$$

where we reuse the notations x_i, y_i, ξ_i, η_i, but we refer now to the points digitized for future use, and not to the calibration points.

Using MATLAB notation we define

$$A2 = \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}$$

and obtain it in MATLAB as a submatrix of C

```
A2  = C(1:2, :);
```

which means rows 1 to 2 of C, all columns. The screen coordinates of the digitized points are collected in a matrix called `DIG`in MATLAB

$$DIG = \begin{vmatrix} \xi_1 & xi_2 & \dots & \xi_n \\ \eta_1 & \eta_2 & \dots & \eta_n \end{vmatrix}$$

We create a matrix of the same size as `DIG`, with n identical columns

$$\begin{vmatrix} a_3 \\ b_3 \end{vmatrix},$$

using the MATLAB command

```
C2 = C(3, :)*ones(1, n);
```

Then, the right-hand side of Equation 4 is obtained in MATLAB with

```
B2 = DIG - C2;
```

Our aim is to obtain the matrix of real world coordinates of the digitized points. We call this matrix `REAL` and calculate it with the help of the backslash operator

```
REAL = A2\B2
```

The calibrating and digitizing processes are guided interactively by *dialog boxes* predefined in MATLAB. In our digitizer we use the following dialog boxes:

uigetfile - that allows the user to choose a graphic file from a given folder;

msgbox - that tells the user what is the next step;

inputdlg - that prompts the user to enter the real-world coordinates of the calibrating points.

A full digitizing function is shown at the end of this article, in the first part of *Listing 3*, that is up to and including the program segment called 'CALCULATE THE REAL-WORLD COORDINATES OF DIGITIZED POINTS'.

7 A digitizer-integrator for curves of statical stability

The digitizer can be combined with an integrator. To give a simple example let us assume that we are given a curve of statical stability and are asked to check if the areas under this curve satisfy the requirements of some code of stability. In the trivial solution we could measure the coordinates of a few points on the curve and use a rule for numerical integration. To 'mechanize' this process and obtain more results, we developed the function `scdigitizer` whose full listing is shown as *Listing 3* at the end of this article. To use the function it is necessary to scan the given curve of statical stability and to invoke the function together with the graphic file. The listing begins with instructions of use written as comments, that is preceded by the percent symbol '%'. Together, these comments form the *help* part of the file. To read these instructions it is sufficient to type on the command line `help scdigitizer`.

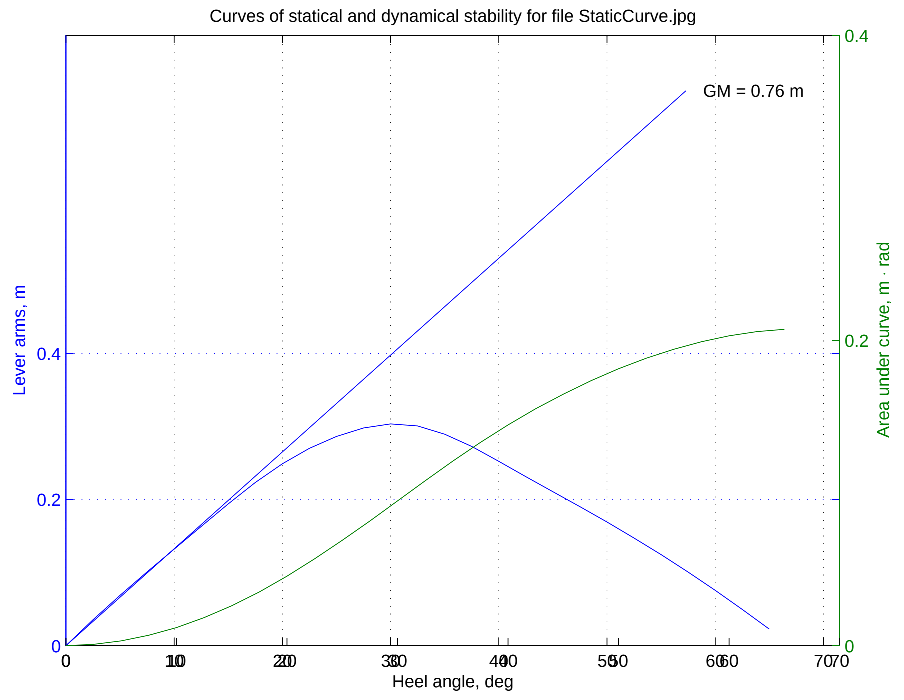


Figure 10: Curves of statical and dynamical stability plotted by the function `scdigitizer`

The first part of the function is a digitizer built as described in the preceding section. To calibrate the curve the user has to pick up three points whose real-world coordinates are known. Most suitable will be a point on the lever-arm axis, the origin of the axes, and a point on the heel-angle axis. After calibration the user is asked to pick up points on the curve of statical stability. In continuation, if the value of $\overline{GM}$ is known, the user is prompted to enter its numerical value. Alternatively, the user can digitize the point on the tangent in origin that corresponds to a heel angle of 1 radian. The mode of entering the metacentric height is chosen interactively with the help of the dialog box `listdlg`.

The program uses the MATLAB function `spline` to interpolate points on the curve, at 2.5^{o} intervals and store them in an array *GZi*. Next, the program uses the MATLAB `cumsum` function that returns the *cumulative sum* of the elements of the argument array. In our case, given an array of righting-arm values , $\overline{GZ}$, and a corresponding array of heeling-angle values, ϕ_i, we use the `cumsum` function to calculate in an economic way the array of areas under the curve of statical stability, from zero to each angle ϕ_i,

$$S_i = \int_0^{\phi_i} \overline{GZ} d\phi$$

The algorithm is described in Biran (2005, pp. 80-3) and is implemented in the program segment called AREA UNDER THE GZ CURVE.

The last program segment prints the results on a file `scdigitizer.out`. The line

```
fid = fopen('scdigitizer.out', 'w')
```

opens a file named `scdigitizer.out`, with permission for writing only, and assigns a *file identifier* to `fid`. The commands `fprintf` belong to the C programming language. Below we see an example of results printed on the file `scdigitizer.out`:

```
Curves of statical and dynamical stability calculated with the
function SCDIGITIZER for file  StaticCurve.jpg
  Heel angle     GZ        Area
    Degrees      m         m.rad
       0.0       0.00      0.000
       2.5       0.04      0.001
       5.0       0.07      0.003
       7.5       0.10      0.007
      ...        ...       ...
      62.5       0.05      0.206
      65.0       0.02      0.207
```

8 Developing a digital integrator

Usually, the digitized points define a plane figure whose geometrical properties must be calculated. This task can be performed by numerical integration using, for example, the trapezoidal or Simpson's rule. Since the invention of the planimeter by Amsler, around 1854, Naval Architects used mechanical analog computers, specifically planimeters, integrators and integraphs to calculate the geometrical properties by following a closed contour on the lines drawing or on intermediate, specially-prepared graphs. In this section we show how to develop functions that imitate the integrators.

8.1 Green's theorem

In hydrostatic calculations we are interested in the geometrical properties of plane figures, such as *waterplanes* or transversal sections known as *stations.* The geometrical properties — areas, moments and moments of inertia — are defined as double integrals over the areas of the figures. Textbooks of Ship Hydrostatics (see, for example, Lewis, 1988, Rawson and Tupper, 2002, Chapter 2, Biran 2005, Chapter 4) show how to approximate these area integrals in Naval Architecture by using methods like the trapezoidal or Simpson's rule. When the figure we are interested in is defined by points on its border, it may be more convenient to convert the double integral into a line integral by means of *Green's theorem.* A common statement of this theorem, as can be found in many textbooks (see, for example, Finney, 1988, Kreyszig, 2005, O/Neil 2003) is

$$\iint_S \left(\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y} \right) dxdy = \oint_C (F_1 dx + F_2 dy) \tag{5}$$

where S is the area of the plane figure and C, its border. This theorem, which has very important applications in physics, has been proposed and used for purposes such as those described in this section. By an appropriate choice of the functions F_1 and F_2 it is possible to calculate all the geometrical properties of a given figure. We use here some of the functions proposed by Turkowski (1997). Thus, to calculate the area, A, we choose $F_1 = -y,\ F_2 = 0$, and transform the double integral into a line integral as follows

$$A = \iint_S (0+1)dxdy = \iint_S dxdy = -\oint_C ydx \tag{6}$$

To calculate the first moment about the $y-$axis we let $F_1 = 0,\ F_2 = x^2/2$ and obtain

$$M_y = \iint_S (x-0)dxdy = \iint_S xdxdy = \frac{1}{2}\oint_C x^2 dy \tag{7}$$

For the first moment about the $x-$axis we choose $F_1 = y^2/2;\ F_2 = 0$, and write

$$M_x = \iint_S (0-y)dxdy = -\iint_S ydxdy = \frac{1}{2}\oint_C y^2 dy$$

and take

$$M_x = -\frac{1}{2}\oint_C y^2 dy \tag{8}$$

For the moment of inertia about the $y-$axis we choose $F_1 = -y^3/3, \ F_2 = 0$ and obtain

$$I_x = \iint_S (0 + y^2) dxdy = \iint_S y^2 dxdy = -\frac{1}{3} \oint_C y^3 dx \tag{9}$$

For the moment of inertia about the $y-$axis we choose $F_1 = 0, \ F_2 = x^3/3$ and have

$$I_y = \iint_S (x^2 - 0) dxdy = \iint_S x^2 dxdy = \frac{1}{3} \oint_C x^3 dx \tag{10}$$

To calculate the product of inertia about the x and $y-$axes we assume $F_1 = 0, \ F_2 = x^2y/2$ and get

$$I_{xy} = \iint_S (xy - 0) dxdy = \iint_S xydxdy = \frac{1}{3} \oint_C x^2 ydx \tag{11}$$

8.2 Parametrization of the border curve

To carry on the integration of the equations described in the previous section we need an analytic expression of the boundary C. In principle any curve, for example splines of various kinds, can be used. Examples and relevant references can be found in Sheynin and Tuzikov (2003). As in Turkowski (1997), for the purposes of this article we choose to approximate the actual boundary by straight-line segments. Such an approximation may appear as rough, but it avoids the disadvantages of more sophisticated approximations. For example, if we want to describe by splines a body plane of a boat with chine lines, we must specify the points where a spline ends and another one begins. Obviously, this is no serious problem, but it may obscure the presentation of the solution. In picturesque language, we prefer to reduce the number of trees to let the reader view the forest.

We assume that the border C is defined by a number of appropriately chosen points. The order of the points is defined by running along the border so as to have always the enclosed area, S, on the left. Let the coordinates of the first point be $x_1, \ y_1$ and the coordinates of the $i-$th point be $x_i, \ i_i$. The last segment must end at point 1, or, in other words, we must close the border. We approximate the curve segment between the point i and the point $i+1$ by a straight-line segment whose parametric equations are

$$\begin{aligned} x &= x_i(1-t) + x_{i+1}t \\ y &= y_i(1-t) + y_{i+1}t \end{aligned} \tag{12}$$

The *parameter* t runs from 0 to 1. Differentiating Equations 12 we obtain

$$\begin{aligned} dx &= (x_{i+1} - x_i)dt \\ dy &= (y_{i+1} - y_i)dt \end{aligned} \tag{13}$$

Substituting Equations 12 and 13 into Equations 6 to 11 we develop formulae valid for the successive straight-line segments that approximate the border. The sums of the results for individual segments yield the geometrical properties of the enclosed area.

As an example, let us use of the above relationships to find the basic formula for calculating the area according to Equation 6:

$$A_i = \frac{1}{2}\int_{x_i}^{x_{i+1}} ydx = \frac{1}{2}\int_0^1 \left[y_i(1-t) + y_{i+1}t\right](x_{i+1}-x_i)dt = \frac{x_{i+1}-x_i}{2}\cdot\frac{y_{i+1}+y_i}{2} \tag{14}$$

In this result we may recognize the trapezoidal rule. We do not show how the other basic formulae are derived, but they can be recognized in the listing of the function `digint` shown as Listing 4 in Subsection 10.4.

8.3 A general digitizer-integrator

Subsection 10.4 contains part of the listing of a general digitizer-integrator based on the theory explained above. The listing should be written on a file called `digint.m`. The first part of the program, immediately after the help part, is not shown because it is identical to the segment of `scdigitizer` contained between the statement `clf` and the title `POST-PROCESSING`.

After completing the digitizing process, the user is prompted to choose one of five possibilities of postprocessing the data, or exit the program. Following this choice the program enters the corresponding branch of a conditional construct called in MATLAB jargon *switch yard*. Schematically, this structure can be described as

```
switch  criterion
    case 1
        statements
    case 2
        statements
    ...
    case 6
        statements
end
```

The available possibilities and the resulting output are described in the help part of Listing 4. As an example, we show in Figure 11 a Z-shaped figure

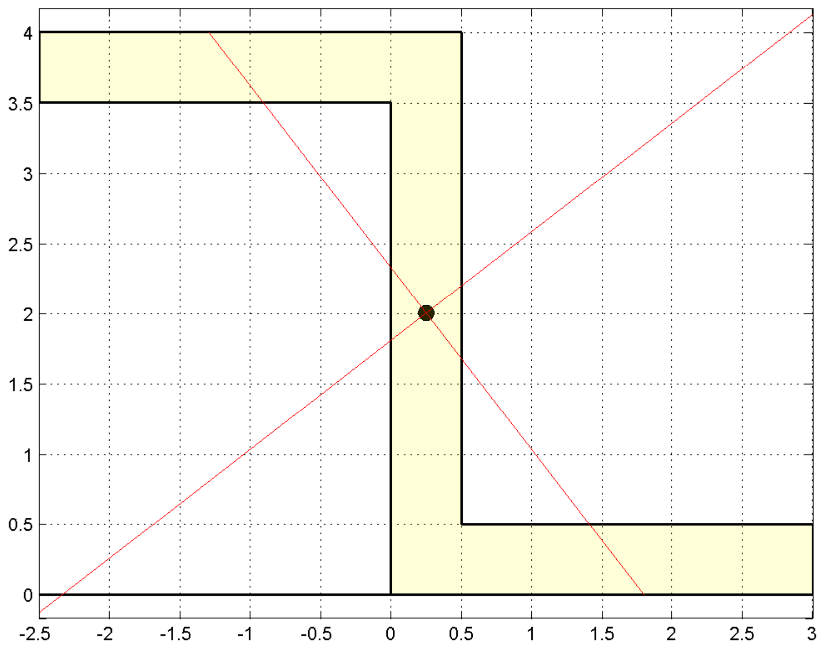


Figure 11: Figure produced by the function `digint`

used in the textbook of Beer and Johnston (1998). The dimensions are in m. After digitizing the figure and choosing the option ‘Geometric properties’, the program plots the digitized figure and shows on it the centroid and the principal axes. An instance of the output file is

```
Properties of geometric figure calculated with the function DIGINT
Area, ........................................... 4.5460
Perimeter .......................................19.0093
First moment about the y axis .................. 1.0890
First moment about the x axis .................. 9.0933
x-coordinate of centroid ....................... 0.2395
y-coordinate of centroid ....................... 2.0003
Moment of inertia about given y-axis ........... 7.2975
Moment of inertia about given x-axis ...........28.7790
Product of inertia about given axes ............-4.5082
Moment of inertia about barycentric y-axis ...... 7.0366
Moment of inertia about barycentric x-axis ......10.5899
Product of inertia about barycentric axes .......-6.6865
Radius of gyration about barycentric y-axis ..... 1.2441
Radius of gyration about barycentric x-axis ..... 1.5263
Angle of rotation of principal axes ............37.5599 deg
Maximum principal moment .......................15.7317
```

```
Minimum principal moment .......................  1.8948
Maximum radius of gyration about principal axis .  1.8603
Minimum radius of gyration about principal axis .  0.6456
Check tensor invariants
Trace before rotation ..........................17.6265
Trace after rotation ...........................17.6265
Determinant before rotation .....................29.8082
Determinant after rotation ......................29.8082
```

For comparison, the exact area is 4.5 m^2, the $x-$coordinate of the centroid, 0.25 m, the $y-$coordinate of the centroid, 2 m, the angle of the principal axes is 37.726 degrees, and the principal moments of inertia are 15.452 and 1.892 m^4. Obviously, another digitization can result in slightly different results.

9 Simulating roll

SIMULINK is probably the most famous MATLAB toolbox. It is the only one supplied in the students' package. SIMULINK enables the user to simulate complex dynamical systems that are described by block diagrams. The software transforms the diagrams into computer code that can be run easily to produce various outputs.

To give a simple example of how SIMULINK works, let us simulate the roll motion, under a wind gust, governed by the equation

$$\ddot{\phi} + \frac{g}{i^2}\overline{GZ} - \frac{W_w(z_A - 0.5T_m)}{g\Delta} \cdot p_w(0.25 + 0.75\cos^3\phi = 0 \tag{15}$$

where A_w s the sail area in m^2; z_A, the height coordinate of the sail-area centroid, in m, measured from the same line as the mean draft; T_m, the mean draft, in m; p_w, the wind pressure in kN/m^2; $g\Delta$, the ship displacement in kN. In Equation 15 we take into account a non-linear restoring-force arm and a non-linear heeling arm, but we neglect added masses and damping. The assumed wind arm is that prescribed by the stability regulations BV 1033 of the German Navy (see Biran, 2005, pp. 221–37). We consider this assumption more realistic than others because it yields a non-zero heeling arm at 90°.

References

Beer, F.P., and Johnston, E.R. Jr 1998, *Vector mechanics for engineers*, 3d SI metric edition, McGraw-Hill Ryerson, Toronto, Canada.

Biran, A. 2005 *Ship hydrostatics and stability*, revised reprint with a Spanish index and amendments by Rubin Lopez-Pulido, Butterworth-Heinemann, Oxford, England.

Biran, A., and Breiner, M. 2002 *MATLAB 6 for Engineers*, 3d edition, Pearson Education, Harlow, England. There are German, French and Greek translations.

Finney, R.L. 1988, *Calculus and analytic geometry*, 7th ed., Addison Wesley publishing Company, Reading-Massachusetts, U.S.A.

Fossen, T.I. 1994 *Guidance and control of ocean vehicles*, John Wiley & Sons.

Kreyszig, E. 2005 *Advanced engineering mathematics*, 9th edition, John Wiley & Sons.

INSEAN, 1963 *Carene di petroliere*, Quaderno n. 2, Roma: INSEAN (Vasca Navale).

Lewis, E.V. (ed) 1988 *Principles of Naval Architecture – Second Revision, Vol. I – Stability and Strength.* Jersey City, N.J.: The Society of Naval Architects and Marine Engineers.

Maritime and Coastguard Agency 2001 *The code of practice for safety of large commercial sailing & motor vessels*, 4th impression, London: The Stationery Office.

O'Neil, P.V. 2003 *Advanced engineering mathematics*, 5th ed., Thomson Brooks/Cole, Pacific Grove - CA, U.S.A.

Rawson, K.J. & Tupper, E.C. 1996 *Basic Ship Theory*, Vol. 1, Harlow, England: Longman.

Sheynin, S. and Tuzikov, A. 2003 *Moment computation for objects with spline curve boundary*, in IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 25, No. 10, October, pp. 13-17–

Turkowski, K. 1997, *Computing 2D polygon moments using Green's Theorem*, http://www.worldserver.com/turk/computergraphics/Moments.pdf.

10 Listings

10.1 Listing 1 - Function SailCode

```
function  SteadyHeel = SailCode(stability, KG, GM, downflood)

%STEADYHEEL Checks stability of sail vessel in accordance to the ‘Code of
%           practice for safety of small commercial sailing & pilot boats’
%           issued by the UK Maritime and Coastguard Agency. Call as
%                SteadyHeel = sailcode(stability, KG, GM, downflood)
%   INPUT
%           stability - 2-by-n array of heel angles, in deg, and
%           cross-curves , in m, in the format
%                       [ heel angles; levers ]
%           KG - vertical centre of gravity, m
%           GM - metacentric heigth, m
%           downflood - downflooding angle, in deg. This argument is
%           optional. If not entered, the default value is 60 deg.
%   OUTPUT
%           SteeadyHeel - maximum admissible angle of heel under steady
%           wind, deg.
%           This output is obtained after pointing the haircross over
%           the intersection of the GZ and steady-wind curves.
%           Written by Adrian Biran as complementary software to the book
%           'Ship Hydrostatics and Stability', revised reprint 2005,
%           Butterworth Heinemann. Copyright 2006 by Adrian B. Biran.

% separate heel angles and lever arms
heel   = stability(1, :);              % heel angles, deg
l_k    = stability(2, :);              % lever arm of form stability (cross-curves)
GZ     = l_k - KG*sind(heel);
% calculate curve of statical stability at 2.5 deg intervals
heeli  = 0: 2.5: heel(end);            % interpolation axis
GZi    = spline(heel, GZ, heeli);     % interpolated righting arms
% calculate gust arm
if nargin < 4
    downflood = 60;
end
GZf    = spline(heel, GZ, downflood);  % righting arm at downflood angle
WL0    = GZf/cosd(downflood)^1.3; gust   = WL0*cosd(heeli).^1.3;
% calculate steady-wind arm
steady = gust/2;
% plot curves
plot(heeli, GZi, 'k-', heeli, gust, 'r--', heeli, steady, 'g-.')
grid
t = 'Sail-vessel stability according to UK Maritime and Coastguard Agency';
title(t)
xlabel('Heel angle, deg')
ylabel('Lever arms, m')
legend('GZ', 'Gust wind', 'Steady wind')
hold on
plot([ 0 180/pi ], [ 0 GM ], 'k-')
[ phi_st, GZst ] = ginput(1);
SteadyHeel       = phi_st;
text(5, max(GZi), [ 'Angle of steady heel = ' num2str(SteadyHeel)
' deg' ])
text(heeli(2), 1.02*gust(2), 'WL0')
plot([ downflood downflood ], [ 0 GZf ], 'k-')
text(1.01*downflood,
0.05*GZf, 'Downflooding angle')
hold off
```

10.2 Listings 2 - B and M-curves demo

```
%B_CURVE plots B and M curves for vessel Lido 9.
```

```
%        Calls Lido9.m and rotate.m
%        During the demo there are two pauses
%        where you have to press ENTER.
%        This is a companion file to Biran, A. (2003), Ship Hydrostatics
%        and Stability, Oxford: Butterworth-Heinemann (See Example 2.6
%        in this book).

Hf_fig = figure;
set(Hf_fig, 'NumberTitle', 'off',...
          'Name', 'B and M curves');

clf
lido9               % load ship data
x0 = -T.*sin(heel); y0 =  T.*cos(heel); k = length(heel);
Bcenter = zeros(2, k); Bcenter(:, 1) = [ 0; KB(1) ];
M       = zeros(2, k); M(:, 1) = [ 0; KM(1) ];
%%%%%%%%%%%%%%%%%%% PLOT MIDSHIP SECTION %%%%%%%%%%%%%%%
f = max((B/0.8), (B^2/(12*T(1))+T(1)/2)); % estimate frame dimensions
plot(station(1, :), station(2, :), 'b-')
axis([ -f/2 f/2 -T(1)/2 (f-T(1)/2) ])
axis('square'), axis('off')
hold on
text(-B/1.8, T(1)+0.25, 'W_0')   % mark waterline in upright condition
text( B/1.8, T(1)+0.25, 'L_0')   % mark waterline in upright condition
text(-B/13, KB(1), 'B') text( 0, KM(1)+0.3, 'M')
% coordinates of initial WL ends
xl =  x0(1) - B/1.5; yl =  y0(1); xr =  x0(1) + B/(1.8-1/10);
yr = y0(1);
plot([ xl xr ], [ yl yr ])
% plot centre of buoyancy in upright condition
plot(0, KB(1), 'r*')
% plot initial metacentre
plot(0, KM(1), 'ko')
plot([ 0 0 ], [ KB(1) KM(1) ], 'c-.')

pause

for l = 2:k
       %%%%%%%%%%%%%%% PLOT WATERLINES %%%%%%%%%%%%%%%
       % coordinates of WL ends
       xl =  x0(l) - B*cos(heel(l))/(1.5+l/1.7);
       yl =  y0(l) - B*sin(heel(l))/(1.5+l/1.7);
       xr =  x0(l) + B*cos(heel(l))/(1.8-l/10);
       yr =  y0(l) + B*sin(heel(l))/(1.8-l/10);
       plot([ xl xr ], [ yl yr ])
       % write heel angle in degrees
```

```
        if l ~= 1
                string = [ num2str(phi(l)) '^o' ];
                text(xl, yl, string)
                text(xr, yr, string)
        end
        %%%%%%%%%% PLOT CENTERS OF BUOYANCY %%%%%%%%%%%
        Bcenter(:, l) = rotate(heel(l))*[ w(l); KB(l) ];
        plot(Bcenter(1, l), Bcenter(2, l), 'r*')
        %%%%%%%%%%%%% PLOT METACENTERS %%%%%%%%%%%%%%%
        M(:, l)       = rotate(heel(l))*[ w(l); KM(l) ];
        plot(M(1, l), M(2, l), 'ko')
        %%%%%%%%%%% PLOT NORMALS TO B CURVE %%%%%%%%%%%
        plot([ Bcenter(1, l), M(1, l) ], [ Bcenter(2, l), M(2, l) ],...
        'k--')
        pause(1)
end

pause
% spline interpolation of B coordinates
Bx = 0: Bcenter(1, k)/50: Bcenter(1, k);  % interpolation axis
By = spline(Bcenter(1, :), Bcenter(2, :), Bx);
plot(Bx, By, 'm-')
% spline interpolation of M coordinats
Mx = 0: M(1, k)/50: M(1, k);              % interpolation axis
My = spline(M(1, :), M(2, :), Mx);
plot(Mx, My, 'y-') text(-B/1.5, 1.4*KM(1),...
    'The M-curve is the evolute of the B-curve')

hold off
```

The file `B_Curve` calls the data of the ship *Lido 9*. Therefore, the following file must stay in the same folder as the file `B_Curve`.

```
%LIDO9  Data of vessel Lido 9.
%       Companion software to Adrian Biran, Ship Hydrostatics and
%       Stability, 2003, Oxford Butterworth Heinemann.

%%%%%%%%%%%%%%%%%%%%  SHIP OFFSETS  %%%%%%%%%%%%%%%%%%%%%%%%
P1  = [ 0.000; 0.50 ]; P2  = [ 0.240; 0.50 ];
P3  = [ 0.240; 0.58]; P4  = [ 1.100; 1.00 ]; P5  = [ 1.787; 1.25 ];
P6  = [ 2.460; 1.50 ]; P7  = [ 2.902; 1.75 ]; P8  = [ 3.100; 2.00 ];
P9  = [ 3.176; 2.25 ]; P10 = [ 3.200; 2.50 ]; P11 = [ 3.218; 2.75 ];
P12 = [ 3.230; 3.00 ]; P13 = [ 3.230; 3.36 ]; P14 = [ 2.099; 3.425 ];
P15 = [ 0.000; 3.489 ];
% Form starboard outline
starb    = [ P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P14 P15 ];
```

```
% Form port outline
port     = [ -starb(1, :); starb(2, :) ];
port     = fliplr(port);
% Form station outline
station  = [ port starb ];
%%%%%%%%%%%%%%%%%  HYDROSTATIC DATA  %%%%%%%%%%%%%%%%%%%%%
phi  = 0: 15: 90;                 % heeling angles, degress
heel = pi*phi/180;                % heeling angles, rad
B    = 6.480;                     % molded breadth, m
T    = [ 1.729 1.575 1.163 0.600 -0.012 -0.693 -1.354 ];
                                  % draft, m
w    = [ 0 1.122 1.979 2.595 2.945 2.874 2.539 ];
                                  % lever arm of buoyancy force, m
KB   = [ 1.272 1.121 0.711 0.107 -0.625 -1.393 -2.108 ];
                                  % height coordinate of centre of buoyancy
KM   = [ 4.596 3.711 2.857 1.830 0.479 -0.869 -1.471 ];
                                  % metacentre above baseline
```

The file B_Curve also calls the file rotate.m. Therefore, this file too must stay in the same folder as the file B_Curve.

```
function        r = rotate(phi)
%ROTATE rotation about the origin
%       rotate(phi) rotates counterclockwise by angle phi
%       This file is called by B_curve.m.
%       For details about coordinate transformations see
%       Biran, A., and Breiner, M. (2002), MATLAB 6 for Engineers.
%       Companion file to Biran, A. (2003), Ship Hydrostatics and
%       Stability, Oxford: Butterworth-Heinemann.

r = [ cos(phi) -sin(phi); sin(phi) cos(phi) ];
```

10.3 Listing 3 - Function scdigitizer

```
function     [ SSCURVE, AREAS ] = scdigitizer(filename)

%SCDIGITIZER  Digitizes the coordinates of points in an imported image of a
%         curve of statical stability and calculates the area under the curve
%         as a function of heeling angle.
%          INPUT
%         The image can be presented in a format admitted by the IMREAD
%         function, for example jpeg. The filename and filetype of the image
%         can be supplied, between quotes, as an argument of the function. Then,
%         the function can be called as
%                     [ SSCURVE, AREAS ] = scdigitizer('filename.filetype')
%         If the function is called without an input argument the user will be
%         prompted to choose the file type and the file name from a dialog box.
%          CALIBRATION
%         To calibrate the image the user has to pick up three noncollinear
%         points of the image and input their real-world coordinates. This will
%         enable the function to find the origin of coordinates, the axes of
%         coordinates and the scale of the image. After picking up a point the
%         user is prompted to introduce its real-world coordinates. The dialog
%         box has two fields, the angle value in degrees, and lever values
%         in m. To switch from the first field to the second use the
%         mouse or Ctrl+Tab. We recommend to choose digitizing points on
%         the axes of the curve.
%          DIGITIZATION
%         In the next phase the user can digitize as many points as necessary.
%         The process is guided by dialogue boxes. As long as the user
%         wants to continue digitizing the answer is YES. After picking
%         up the last point on the curve the user has to answer NO.
%          OUTPUT
%         The output consists of two arays, SSCURVE and AREAS, a file
%         of results, scdigitizer.out, and a plot of the curves of statical
%         and dynamical stability, where the latter means the curve of areas
%         under the former curve. The two output arrays are obtained
%         if the function is called with two output arguments, as shown
%         above. Instead of SSCURVE and AREAS the user may write any other
%         names. The first array contains the real-word GZ values, in m, given
%         at 2.5-degree intervals. The second array contains the values of the
%         area under the curve, in m.rad, at the same angle values. The output
%         file displays the heel angles, the GZ and area values in
%         tabular form.
%          Written by Adrian Biran as complementary software to the book
%          ‘Ship Hydrostatics and Stability’, revised reprint 2005,
%          Butterworth Heinemann. Copyright 2006 by Adrian B. Biran.

clf

if nargin < 1
    [ filename, pathname ] = uigetfile({'*.jpg'; '*.gif'; '*.bmp'; '*.tif' },...
    'Select a graphic input file')
end

A = imread(filename); him = image(A); axis off, grid
hold on

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALIBRATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
hm = msgbox('To calibrate pick up three non-collinear points');
waitfor(hm)
```

```
X                     = ones(3, 3);     % Allocate space for real-world, homogeneous
                                       % coordinates of calibrating points
S                     = zeros(3, 2);    % Allocate space for screen coordinates of
                                       % calibrating points;
% begin arguments of input dialogue
prompt                = {'Enter actual angle value, deg',...
                         'Enter actual lever value, m'};
numlines              = 2; defaultanswer = {'', ''};
 options.Resize       = 'on';
 options.WindowsStyle = 'normal';
 options.Interpreter  = 'tex';
% end arguments of input dialogue

for k = 1:3                             % begin calibrating loop
    [ xs, ys ] = ginput(1);
    S(k, 1) = xs;
    S(k, 2) = ys;
    name        = [ 'Point ' num2str(k) ];

    Actual      = inputdlg(prompt, name, numlines, defaultanswer, options);
    X(k, 1)     = str2double(Actual{1});
    X(k, 2)     = str2double(Actual{2});
end                                     % end calibrating loop
%%%%%%%%%%%%%%% CALCULATE THE MATRIX OF TRANSFORMATION COEFFICIENTS %%%%%%%%%%%%

C = X\S;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DIGITIZING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
hm = msgbox('Start digitizing from zero to maximum heel');
waitfor(hm)
DIG    = ones(2, 1);                     % initialize array of coordinates of the
                                         % digitized points
l      = 1;                              % initialize counter of digitized points
button = 'Yes';                          % initialize loop sentinel

while strcmp(button, 'Yes')
    [ xd, yd ] = ginput(1);
    plot(xd, yd, 'r.')                   % mark digitized point
    DIG(1, l)  = xd;
    DIG(2, l)  = yd;
    l          = l+ 1;
    qstring    = [ 'Digitize point ' num2str(l) ];
    button = questdlg(qstring, 'Digitizing');
end
hold off

%%%%%%%%%%%%%% CALCULATE REAL-WORLD COORDINATES OF DIGITIZED POINTS %%%%%%%%%%%%%%
```

```
[ m, n ] = size(DIG);
REAL     = zeros(2, n);                  % allocate space for real-world
                                         % coordinates of digitized points
A2       = C(1:2,:)'; C2       = C(3, :)'*ones(1, n); B2       =
DIG - C2;
REAL     = A2\B2;                        % returns real-world coordinates of
                                         % digitized points
phi      = REAL(1, :);                   % real-word heel-angle values, degrees
phi(1)   = 0;
GZ       = REAL(2, :);                   % real-world righting-lever values
GZ(1)    = 0;
phir     = pi*phi/180;                   % real-world heel-angle values, radians
%%%%%%%%%%%%%%%%%%%%%%%%%%%% POST-PROCESSING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
phii     = 0: 2.5: max(phi);             % axis for spline interpolation
GZi      = spline(phi, GZ, phii);        % interpolated values
%%%%%%%%%%%%%%%%%%%%%%%%%% AREA UNDER THE GZ CURVE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
l   = length(GZi); GZ1 = GZi(1: (l-1)); S1  = [ 0 cumsum(GZ1) ];
GZ2 = GZi(2: l); S2  = [ 0 cumsum(GZ2) ]; S   =
(pi*2.5/(2*180))*(S1 + S2)
%%%%%%%%%%%%%%%%%%%%%%%%%%%% TANGENT IN ORIGIN %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Choose mode of entering GM
LS = [ {'Numerical value of GM known'},...
       {'GM to be picked up on graph'},...
       {'GM unknown'} ];
[ Selection, ok ] = listdlg('Liststring', LS, 'SelectionMode',
'Single'); switch Selection
    case 1
        % begin arguments of input dialogue
        prompt               = {'Enter GM value in m'};
        numlines             = 1;
        defaultanswer        = {''};
        options.Resize       = 'on';
        options.WindowsStyle = 'normal';
        options.Interpreter  = 'tex';
        % end arguments of input dialogue
        GMstring = inputdlg(prompt, name, numlines, defaultanswer, options);
        GM = str2double(GMstring);
    case 2
        [ xd, yd ] = ginput(1)
        Bgm       = [ xd; yd ] - C(3, :)';
        GMphi    = A2\Bgm;                    % returns real-world coordinates
        GM       = GMphi(2);
    case 3
        GM = 0;
end
```

```
%%%%%%%%%%%%%%%%%% PLOT CURVES OF STATICAL AND DYNAMICAL STABILITY %%%%%%%%%%%%%%%%%
figure(2) [ Ax, H1, H2 ] = plotyy(phii, GZi, phii, S);

 if GM > 0
     hold on
     axis([ 0 1.1*max(phii) 0 1.1*GM ])
     plot([ 0 180/pi ], [ 0 GM ]);
     t = [ 'GM = ' num2str(GM) ' m' ];
     text(185/pi, GM, t)
     hold off
 end

 grid
 title([ 'Curves of statical and dynamical stability for file ' filename ])
 xlabel('Heel angle, deg')
 set(get(Ax(1), 'Ylabel'),'String','Lever arms, m')
 set(get(Ax(2),
'Ylabel'),'String','Area under curve, m \cdot rad')
 SSCURVE = GZi;
 AREAS   = S;

 fid   = fopen('scdigitizer.out', 'w');
 fprintf(fid, 'Curves of statical and dynamical stability\n');
 fprintf(fid, 'calculated with the function SCDIGITIZER\n');
 fprintf(fid, 'for file  %s \n', filename);
 fprintf(fid, '  Heel angle     GZ        Area\n');
 fprintf(fid, ' Degrees      m         m.rad\n');
 for k = 1:l
    fprintf(fid, '%10.1f %10.2f %10.3f \n', phii(k), GZi(k), S(k));
 end
 fclose(fid)
```

10.4 Listing 4 — A general digitizer-integrator

```
function     [ REAL, PROPERTIES ] = digint(filename)

%DIGINT  Digitizes the coordinates of points in an imported image,
%        calculates distances between points and geometric properties of
%        plane, polygonal figures.
%        NOTE
%        If a figure has an axis of symmetry, that axis is also a
%        principal axis. Small errors of calibration and/or digitization
%        can yield wrong principal axes. Therefore, for symmetrical
%        figures use the function wldigitizer instead of this function.
%        For curves of statical stability use the function scdigitizer,
%        and for body plans the function bpdigitizer.
%        INPUT
%        The image can be presented in a format admitted by the IMREAD
%        function, for example jpeg. The filename and filetype of the image
%        can be supplied, between quotes, as an argument of the function. Then,
%        the function can be called as
%                    [ REAL, PROPERTIES ] = digitizer('filename.filetype')
%        If the function is called without an input argument the user will be prompted
%        to choose the file type and the file name from a dialogue box.
%        CALIBRATION AND DIGITIZING
%        To calibrate the image the user has to pick up three noncollinear points
%        of the image and input their real-world coordinates. This will enable the function
%        to find the origin of coordinates, the axes of coordinates and the scale of
%        the image. In the next phase the user can digitize as many
%        points as necessary. The process is guided by dialogue boxes.
%        The points must be followed in a counterclockwise sense, that
%        is so as to have the polygonal area to the left of the cursor.
%        Digitizing in the opposite direction leads to wrong results.
%        POSTPROCESSING
%        In the third phase the user is prompted to choose a postprocessing of the
%        acquired data, or to exit the programme. The possibilities of post processing are:
%            - draw the axes of coordinates;
%            - find the real-world coordinates of a point picked up by
%              the user;
%            - find the real-world distance between two points picked up
%              by the user;
%            - calculate the geometric properties of the polygonal
%            figure defined by the digitized points.
%        The calculation of the first and the second moments of the polygonal figure is
%        based on Green's theorem (see, for example, Turkowski, K., `Computing 2D polygon
%        moments using Green's theorem', Apple Computer, Inc.).
%        OUTPUT
%        The results of calculations are displayed on-line in dialogue
%        boxes. In addition, the function outputs two arrays of data:
%            - the 2-by-n array REAL that contains the real-world coordinates
%              of the digitized points in the format
%                    x1  x2 ... xn
%                    y1  y2 ... yn
%            - the 1-by-12 array PROPERTIES that contains the geometric
%              properties of the polygonal figure in the order
%                       1) polygonal area
%                       2) x-coordinate of centroid
%                       3) y-coordinate of centroid
%                       4) second moment about barycentric y-axis
%                       5) second moment about barycentric y-axis
%                       6) product of inertia about barycentric axes
%                       7) radius of gyration about barycentric y-axis
%                       8) radius of gyration about barycentric x-axis
%                       9) maximum principal moment of inertia
%                      10) minimum principal moment of inertia
%                      11) maximum radius of gyration about pricipal axes
%                      12) minimum radius of gyration about pricipal axes
%        These, and additional data are also printed on a file called `digint.out'.
%        The digitized polygon and the principal axes are plotted in real-world coordinates
%        in a window called Figure 2. To uncover the underlying input figure drag Figure 2
%        aside.
%        This function calls the function POINT. Therefore, the file of
%        this function must be included in the same directory as this
%        function.
%        Written by Adrian Biran as complementary software to the book
%        `Ship Hydrostatics and Stability', revised reprint 2005,
%        Butterworth Heinemann. Copyright 2006 by Adrian B. Biran.

           THIS PART IS IDENTICAL TO THE DIGITIZING PART OF SCDIGITIZER

%%%%%%%%%%%%%%%%%%%%%%%%%%% POST-PROCESSING %%%%%%%%%%%%%%%%%%%%%%%%%%%
            A0   = 0;                % initialize polygonal area
            Per  = 0;                % initialize perimeter
            My   = 0;                % initialize moment about given y-axis
            Mx   = 0;                % initialize moment about given x-axis
            xC   = 0;                % initialize x-coordinate of centroid
            yC   = 0;                % initialize y-coordinate of centroid
            Iy   = 0;                % initialize second moment about given y-axis
            Ix   = 0;                % initialize second moment about given x-axis
            Ixy  = 0;                % initialize product of inertia about given axes
            Iy0  = 0;                % initialize second moment about barycentric y-axis
            Ix0  = 0;                % initialize second moment about barycentric y-axis
            Ixy0 = 0;                % initialize product of inertia about barycentric axes
            ky0  = 0;                % initialize radius of gyration about barycentric y-axis
            kx0  = 0;                % initialize radius of gyration about barycentric x-axis
            Imax = 0;                % initialize maximum principal moment of inertia
            Imin = 0;                % initialize minimum principal moment of inertia
            kmax = 0;                % initialize maximum radius of gyration about pricipal axes
            kmin = 0;                % initialize minimum radius of gyration about pricipal axes

flag = 'Continue'; while strcmp(flag, 'Continue')
    S  = [ {'Show axes'}, {'Coordinates of one point'},...
         {'Distance between 2 points'}, {'Polygonal area'},...
         {'Geometric properties'}, {'Exit'} ];
    [ Selection, ok ] = listdlg('Liststring', S, 'SelectionMode', 'single');
switch Selection
   case 1      % Show axes
       Xmax  = max(max(REAL(1, :)), max(X(:, 1)));
       Ymax  = max(max(REAL(2, :)), max(X(:, 2)));
       Ereal = [ Xmax 0 1; 0 Ymax 1; 0 0 1 ];     % describes real-world axes
       Escr  = Ereal*C ;                           % describes screen axes
       plot([ Escr(3, 1) Escr(1, 1)], [ Escr(3, 2) Escr(1, 2) ], 'k-')
       plot([ Escr(3, 1) Escr(2, 1)], [ Escr(3, 2) Escr(2, 2) ], 'k-')
   case 2      % Coordinates of one point
       [ xs, ys ] = ginput(1);       % screen coordinates of digitized point
       B2         = [ xs; ys ] - C(3, :)';
       R          = A2\B2;           % real-world coordinates of digitized point
       t          = [ 'x = ' num2str(R(1)) ', y = ' num2str(R(2)) ];
       uiwait(msgbox(t, 'Coordinates', 'none', 'modal'));
   case 3      % Distance between 2 points
       [ x1, y1 ] = ginput(1);       % screen coordinates of first point
       B2         = [ x1; y1 ] - C(3, :)';
       R1         = A2\B2;           % real-world coordinates of first point
       [ x2, y2 ] = ginput(1);       % screen coordinates of second point
       B2         = [ x2; y2 ] - C(3, :)';
       R2         = A2\B2;           % real-world coordinates of second point
       R3         = R2 - R1;
       d          = sqrt(R3'*R3);    % distance
       t          = [ 'distance = ' num2str(d) ];
       uiwait(msgbox(t, 'Distance','none', 'modal'));
   case 4     % Polygonal area
       A0 = polyarea(REAL(1, :), REAL(2, :));
       t  = [ 'Polygone area = ' num2str(A0) ];
       uiwait(msgbox(t, 'Area','none', 'modal'));
   case 5     % Geometric properties
       A0   = polyarea(REAL(1, :), REAL(2, :)); % calculate polygonal area
       [ m, n ] = size(REAL);
       for k = 2:n     % Begin loop over straight-line segments
           Per = Per + norm(REAL(:, k) - REAL(:, (k-1)));
           dMy = (REAL(1,k)^2 + REAL(1,k)*REAL(1,(k-1)) + REAL(1,(k-1))^2)*(REAL(2,k) - REAL(2,(k-1)));
           My  = My + dMy;
           dMx = (REAL(2,k)^2 + REAL(2,k)*REAL(2,(k-1)) + REAL(2,(k-1))^2)*(REAL(1,k) - REAL(1,(k-1)));
           Mx  = Mx - dMx;
           dIy = REAL(1, (k-1))^3 + REAL(1, (k-1))^2*REAL(1, k) + REAL(1, (k-1))*REAL(1, k)^2 + REAL(1,k)^3;
           Iy  = Iy + dIy*(REAL(2,k) - REAL(2,(k-1)));
           dIx = REAL(2, (k-1))^3 + REAL(2, (k-1))^2*REAL(2, k) + REAL(2, (k-1))*REAL(2, k)^2 + REAL(2,k)^3;
           Ix  = Ix - dIx*(REAL(1,k) - REAL(1,(k-1)));
           dxy = 2*REAL(1,k)*REAL(1,(k-1))*(REAL(2,(k-1)) + REAL(2,k));
           dxy = dxy + REAL(1,(k-1))^2*(3*REAL(2,(k-1)) + REAL(2,k));
           dxy = dxy + REAL(1,k)^2*(REAL(2,(k-1)) + 3*REAL(2,k));
           Ixy = Ixy + dxy*(REAL(2,k) - REAL(2,(k-1)));
       end             % End loop over straight-line segments
       % Calculate moments about given axes
       My   = My/6;
       xC   = My/A0;
       Mx   = Mx/6;
       yC   = Mx/A0;
       Iyy  = Iy/12;
       Ixx  = Ix/12;
       if (isnan(Iyy)) | (isnan(Ixx))
            warnstr = 'Digitizing error';
            dlgname  = 'Moment of inertia not a number';
            msgbox(warnstr, dlgname, 'warn', 'modal');
       end
       if (Iyy <= 0) | (Ixx <= 0)
           warnstr = 'Digitizing sense probably wrong';
           dlgname  = 'Moment of inertia <= 0';
           msgbox(warnstr, dlgname, 'warn', 'modal');
       end
       Ixy  = Ixy/24;
       % Calculate moments about barycentric axes
       Iy0  = Iyy - xC^2*A0;
       Ix0  = Ixx - yC^2*A0;
       Ixy0 = Ixy - xC*yC*A0;
       % Radii of gyration about barycentric axes
       ky0  = sqrt(Iy0/A0);                % about y axis
       kx0  = sqrt(Ix0/A0);                % about x axis
       % Calculate angle of principal axes
       tgt  = -2*Ixy0/(Ix0 - Iy0);
       th   = atand(tgt)/2;                % angle in degrees
       tr   = atan(tgt)/2;                 % angle in radians
       % Calculate principal moments of inertia
       I1   = (Iy0 + Ix0)/2;
       I2   = ((Ix0 - Iy0)/2)^2;
       Imax = I1 + sqrt(I2 + Ixy0^2);
       Imin = I1 - sqrt(I2 + Ixy0^2);
       % Radii of gyration about principal axes
       kmax = sqrt(Imax/A0);               % maximum
       kmin = sqrt(Imin/A0);               % minimum
       % Plot digitized figure and principal axes
       x    = REAL(1, :);
       y    = REAL(2, :);
       % Centroid
       figure(2)
       hp = area(x, y); grid
       set(hp, 'FaceColor', 'y', 'EdgeColor', 'k', 'LineWidth', 1.5)
       axis equal
       alpha(0.15)
       xmin = min(x);
       xmax = max(x);
       ymin = min(y);
       ymax = max(y);
       hold on
       point([ xC; yC ], (xmax - xmin)/100)
       % axis x
       yxmin = yC - (xC - xmin)*tan(tr);
       yxmax = yC + (xmax - xC)*tan(tr);
       plot([ xmin xmax ], [ yxmin yxmax ], 'g-')
       % axis y
       xymin = xC + (yC - ymin)*tan(tr);
       xymax = xC - (ymax - yC)*tan(tr);
       plot([ xymin xymax ], [ ymin ymax ], 'r-')
       hold off        % End plot of digitized figure and principal axes
       % Calculate tensorial invariants
       Inv11 = Ix0 + Iy0;         % Trace before rotation
       Inv12 = Imax + Imin;       % Trace after rotation
       Inv21 = Ix0*Iy0 - Ixy0^2;  % Determinant before rotation
       Inv22 = Imax*Imin;         % Determinant after rotation
       % print results
       fid = fopen('digitizer.out', 'w');
       fprintf(fid, 'Properties of geometric figure\n');
       fprintf(fid, 'calculated with the function DIGITIZER\n')
       fprintf(fid, 'Area, ......................................... %14.4f \n', A0);
       fprintf(fid, 'Perimeter ..................................... %14.4f \n', Per);
       fprintf(fid, 'First moment about the y axis .................. %14.4f \n', My);
       fprintf(fid, 'First moment about the x axis .................. %14.4f \n', Mx);
       fprintf(fid, 'x-coordinate of centroid ....................... %14.4f \n', xC);
       fprintf(fid, 'y-coordinate of centroid ....................... %14.4f \n', yC);
       fprintf(fid, 'Moment of inertia about given y-axis ........... %14.4f \n', Iyy);
       fprintf(fid, 'Moment of inertia about given x-axis ........... %14.4f \n', Ixx);
       fprintf(fid, 'Product of inertia about given axes ............ %14.4f \n', Ixy);
       fprintf(fid, 'Moment of inertia about barycentric y-axis ...... %14.4f \n', Iy0);
       fprintf(fid, 'Moment of inertia about barycentric x-axis ...... %14.4f \n', Ix0);
       fprintf(fid, 'Product of inertia about barycentric axes ....... %14.4f \n', Ixy0);
       fprintf(fid, 'Radius of gyration about barycentric y-axis ..... %14.4f \n', ky0);
       fprintf(fid, 'Radius of gyration about barycentric x-axis ..... %14.4f \n', kx0);
       fprintf(fid, 'Angle of rotation of principal axes ............ %14.4f deg \n', th);
       fprintf(fid, 'Maximum principal moment ....................... %14.4f \n', Imax);
       fprintf(fid, 'Minimum principal moment ....................... %14.4f \n', Imin);
       fprintf(fid, 'Maximum radius of gyration about principal axis . %14.4f \n', kmax);
       fprintf(fid, 'Minimum radius of gyration about principal axis . %14.4f \n', kmin);
       fprintf(fid, 'Check tensor invariants\n')
       fprintf(fid, 'Trace before rotation .......................... %14.4f \n', Inv11);
       fprintf(fid, 'Trace after rotation ........................... %14.4f \n', Inv12);
       fprintf(fid, 'Determinant before rotation .................... %14.4f \n', Inv21);
       fprintf(fid, 'Determinant after rotation ..................... %14.4f \n', Inv22);
fclose(fid)
```

```
         case 6     % Exit
            flag = 'Quit';
    end
end
% Fill output array of geometric properties
PROPERTIES = [ A0 xC yC Iy0 Ix0 Ixy0 ky0 kx0 Imax Imin kmax kmin ];
hold off
```

The function digint calls the function point. Therefore, the following function must stay in the same folder as digint.

```
     function        point(C, r)

%POINT  plots solid circles marking points
%       C   coordinates of centre
%       r   radius of circle

t = 0: pi/30: 2*pi;
x = C(1) + r*cos(t); y = C(2) + r*sin(t);
patch(x, y, [ 0 0 0 ])
```