Line Glade, s194354

DTU Compute
Department of Mathematics and Computer Science

Math Apps

A Guide to Creating Math Apps

Maple Math Apps
A Guide to Creating Maple Math Apps

Math Apps guide
Fall 2022

By
Line Glade

Contents

1 Introduction

MathApps is a built-in functionality in Maple which allows for creation of math questions with different types of grading. These worksheets can then be exported to Möbius and made available for use by students. It can be a great repetition tool to make sure a certain concept is well-understood, as well as a way to get statistics on homework assignments with less manual grading.

MathApps is a sort of continuation of programming in Maple, meaning that all functionality of Maple is available when creating MathApps.

A Maple Math App consists of a Maple Worksheet with embedded components and the startup code. The worksheet itself is the front-end of the Math App and should contain all relevant information and text for the question.

To create a MathApp one must first create a worksheet in Maple. The startup code editor can then be opened using ”Ctrl+Shift+E” or through ”Edit” → ”Startup Code”.

Using the Components Palette in Maple, it is possible to insert space for plots, text areas, buttons, sliders and more, creating communication between the back-end and the front-end. These components make it possible to change the front-end through the startup code.

In the following chapters we will first introduce the components palette and how to use embedded components in Maple. Then the creation of the Math App worksheets is discussed and the startup code is described. When describing the worksheets and startup codes we will use examples as a baseline, from which we can explain. Some of these examples are created as templates, meant to serve as a way to get started on making Math Apps without having to start from scratch.

Maple offers a free service called the MapleCloud, where one can upload and access Math Apps, [1]. This service is free and makes it possible for users to solve Math Apps without having Maple: it simply requires that the creator of the Math Apps own Maple.

The templates mentioned in the guide have been uploaded to Github and to MapleCloud.

2 Components

2.1 The Components Palette

Through the Components Palette it is easy to insert components into the worksheet, as seen in Figure 2.1.

Figure 2.1: The components palette

For each component there are different options that can be changed directly in the worksheet or in the startup code. Maple names the components according to their type and the number of the component starting from zero. If the "Editable" box is enabled, the value of the component can be changed in the worksheet itself, otherwise it can not be changed by a user attempting to solve the math question. In this case the component can only be accessed through the startup code.

The most simple component is the Text Area component. Text Area components are used to display text or values for the question, which are set in the startup code, or to get input from a user.

When using the components, one should use a naming convention. A way to make the coding easier is to name the components according to their function. For example, we could name the Text Area component meant to accept input from the user "Input" and the information needed to answer a question could be presented in Math Container components named "Given1" and "Given2" etc.

Each component has several properties which can be set directly in the worksheet through the Context Panel (the panel on the right-hand side of the worksheet, which opens when a component is clicked). The context panel for a Text Area component can be seen in Figure 2.2.

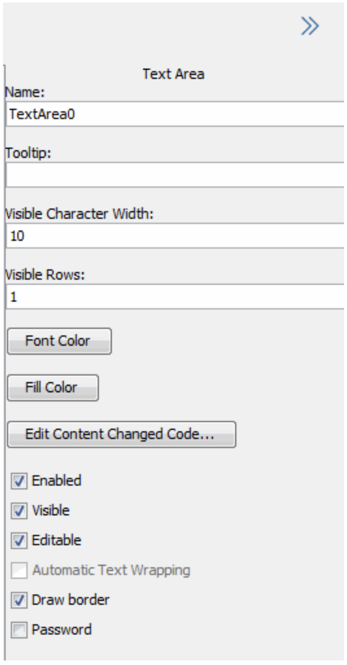


Figure 2.2: Properties of a Text Area component, image taken from [2].

If a component is "Enabled" the component can be interacted with. If "Enabled" is set to false, the component is dimmed and cannot be changed. "Visible" determines whether the component is displayed, and "Editable" determines if a component can only be changed through the startup code, or if the user can change the value directly in the worksheet. Further information on what the different properties mean can be found at Maple's help page: https://www.maplesoft.com/support/help/Maple/view.aspx?path=EmbeddedComponents.

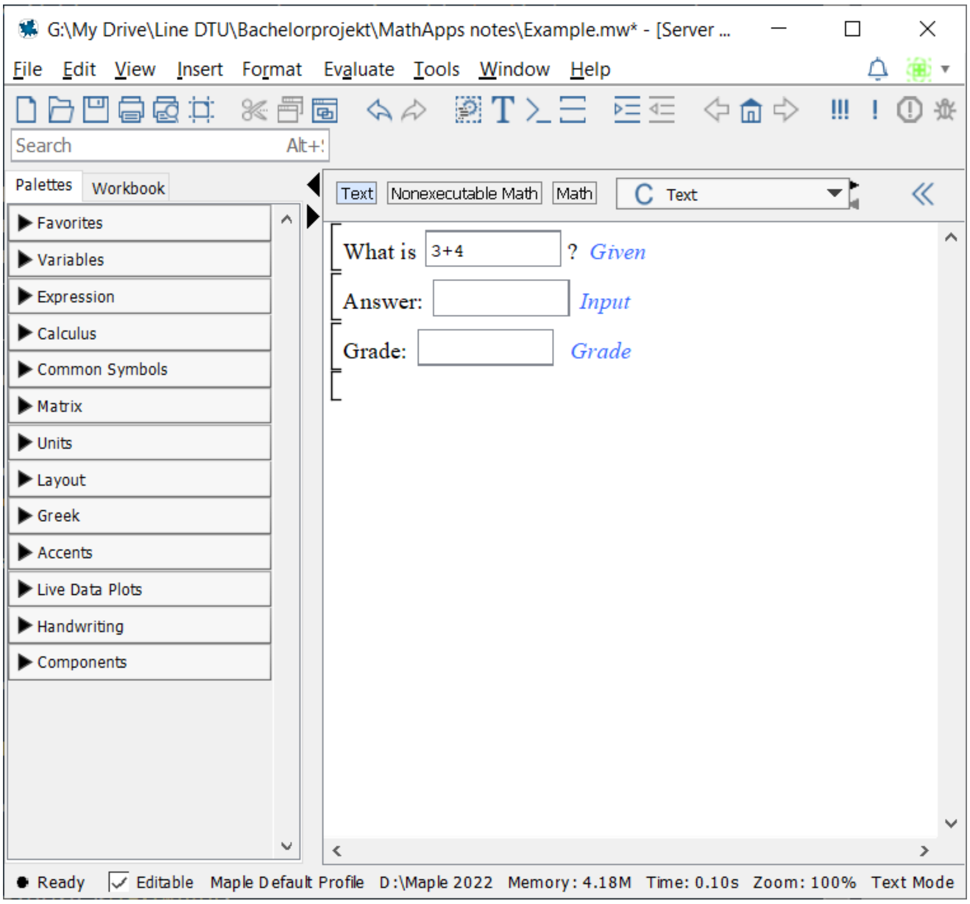
A full list of all available components can be seen in Table 2.1.

Button	Check Box	Combo Box	Data Table	Dial
Label	List Box	Math Expression	Meter	Microphone
Plot	Radio Button	Rotary Gauge	Shortcut	Slider
Speaker	Text Area	Toggle Button	Video	Volume Gauge

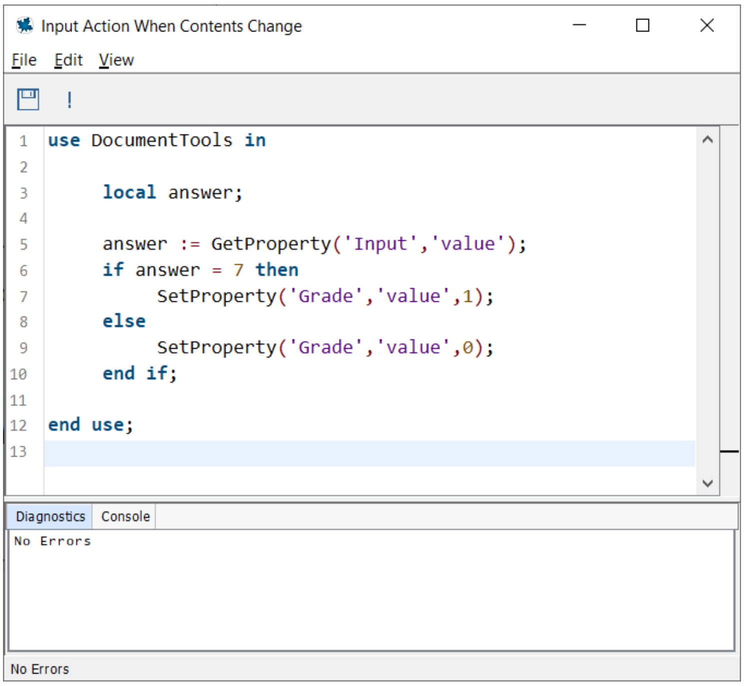
Table 2.1: Available components in Maple 2022, [3].

2.2 Edit interaction code

As seen in Figure 2.2 for Text Areas we can Edit Content Changed Code. Say we have a static problem of "3+4" we can write code to determine if the correct answer is entered into the answer box. This simple Math App is shown in Figure 2.3. Here, we have written in blue the names of the Text Area components. The "Given" Text Area has been changed manually. The next section is about the worksheet itself.



(a) A Maple worksheet of a simple Math App



(b) The Edit Content Changed Code area for the "Answer" Text Area.

Figure 2.3: An example of a very simple Math App with grading done using only the components.

3 The Math Apps worksheet

We start with a worksheet as one often uses in Maple.

3.1 Shortcuts

First we will mention some practical keyboard shortcuts for those who might not be used to using Maple's Worksheets. These are for Windows users, but similar shortcuts exist for Mac, often just replacing the "Ctrl"-button with the "Cmd"-button.

The most frequently used shortcuts when creating the worksheet part of a Math App are as shown in Table 3.1.

Shortcut	Functionality
Ctrl + J	Inserts an execution group after the current line
Ctrl + K	Inserts an execution group before the current line
Ctrl + T	Changes the mode to be text instead of math mode. Also removes the Maple prompt (Shown as ">")
F9	Toggles display of the execution group boundaries, giving the worksheet a more polished look
F5	Switches between text and math mode
_ _	Creates subscript
^	Creates superscript

Table 3.1: Some practical shortcuts for the Math App worksheet.

Additional shortcuts can be found at Maple's online help pages: Windows, UNIX and Mac.

3.2 Examples

What makes a good worksheet for a Math App depends on the type of question. It is also a good idea to experiment with different layouts to get an idea of what one prefers. In the following subsections, we will present different examples of Math App worksheets, ranging from simple to more complex. If the purpose of the Math App is to be uploaded, e.g. to Möbius for online access, it is best to separate an assignment into exercises made in different Math Apps. This provides more freedom and robustness.

3.2.1 Simple static Math App

In Figure 3.1 is shown a Math App worksheet before any input is entered. Only the answer box here is set to be editable. The layout and the text outside of the boxes is written without any connection to the startup code. As previously, the names of the components are written in blue next to the components.

Once input has been entered the MathApp is as shown in Figure 3.2.

The startup code for the Math App is shown in Figure 3.4 and can also be found in the appendix. In the next section we will describe the startup code. Finally, the code for when

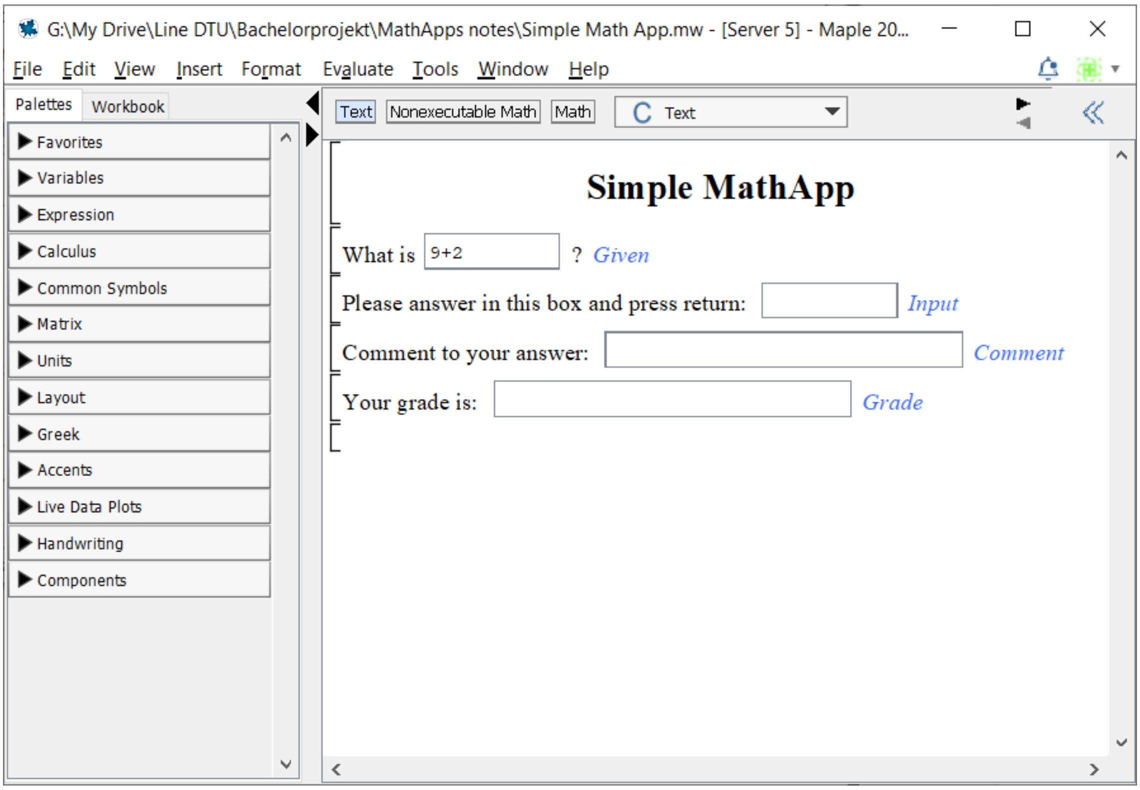


Figure 3.1: The worksheet of a simple static MathApp before user input

the content of the ”Input” Text Area is changed can be seen in Figure 3.3. In comparison with Figure 2.3b the code is shorter, since we can simply call a procedure defined in the startup code.

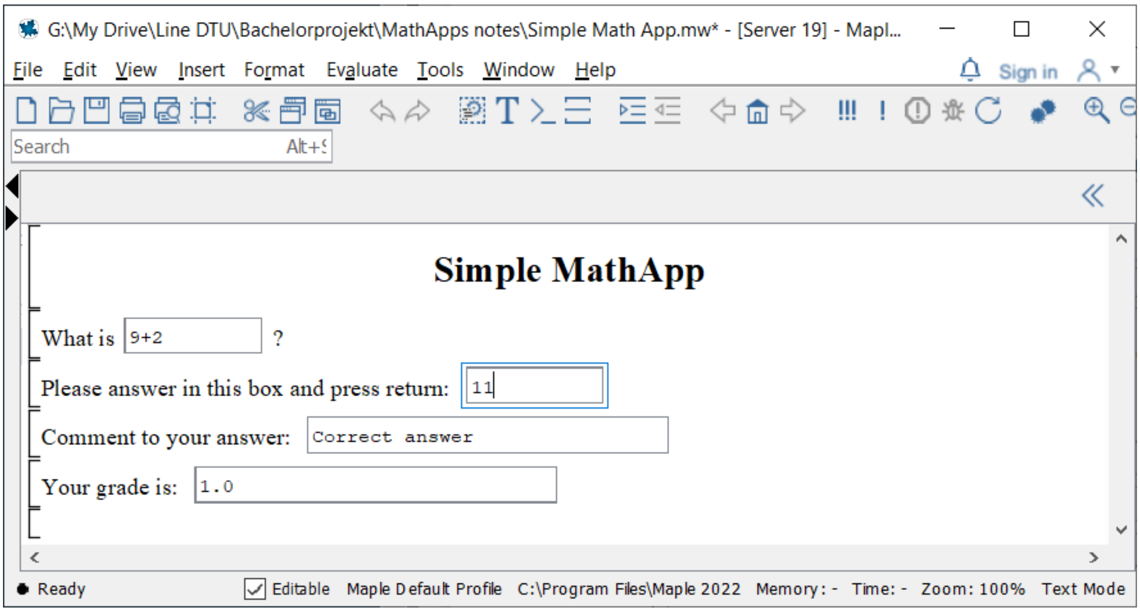


Figure 3.2: The worksheet after user input is entered

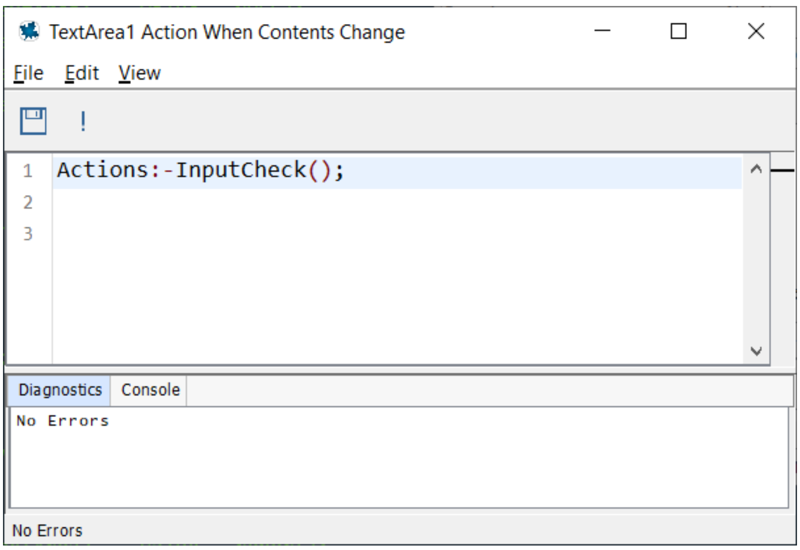


Figure 3.3: The "Content Changed Code..." for grading the MathApp when input is entered

Startup Code For: Simple Math App

File Edit View

```
Actions := module()
     export Initialize, InputCheck, Grade;
     local a, ans, n1, n2; # define local and global values
     global grade, report;

     # Initialization
     Initialize:= proc()
          n1 := 9; n2 := 2;
          ans := n1+n2;
          DocumentTools:-SetProperty('Given', 'value', "9+2");
          DocumentTools:-SetProperty('Input', 'value', NULL);
          DocumentTools:-SetProperty('Comment', 'value', "");
          DocumentTools:-SetProperty('Grade', 'value', "");
     end proc; # Initialize procedure end

     ## Input check
     InputCheck:= proc()
          a:= parse(DocumentTools:-GetProperty('Input', 'value'));

          if evalb(a = ans) then
               grade := 1.0;
               report:= "Correct answer";
          else
               grade:= 0.0;
               report:= "Not correct answer";
          end if;

          DocumentTools:-SetProperty('Comment', 'value', report);
          DocumentTools:-SetProperty('Grade', 'value', grade);
     end proc: # Input check procedure end

     Grade:= proc()
          if whattype(evalf(grade)) = float then
               return grade;
          else
               return 0;
          end if;
     end proc: # Grade procedure end

end module: # Actions module end

Actions:-Initialize():
```

Diagnostics Console

No Errors

Figure 3.4: The Startup code for the MathApp

3.2.2 A simple non-static Math App: Template 1

By hiding the execution group boundaries and changing the font in the worksheet, it is possible to achieve a much cleaner look. We have also put the contents into a table with a blue background color for the headline. The following Math App (Figure 3.5) is similar to the previous Math App, but now with randomized variables and layout changes. We have also added some blue and red text. These are meant to serve as guides to inform the user of what the components are named and how to access the startup code.

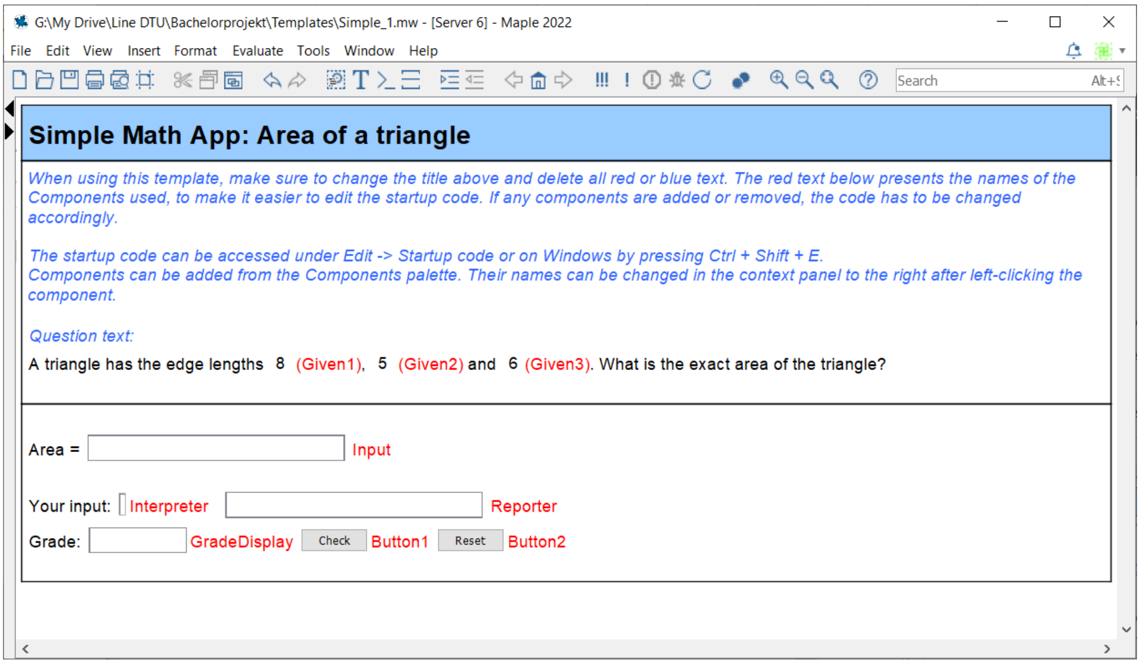
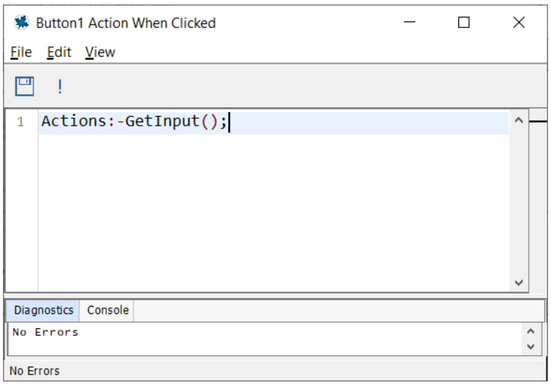


Figure 3.5: The Worksheet for Template 1

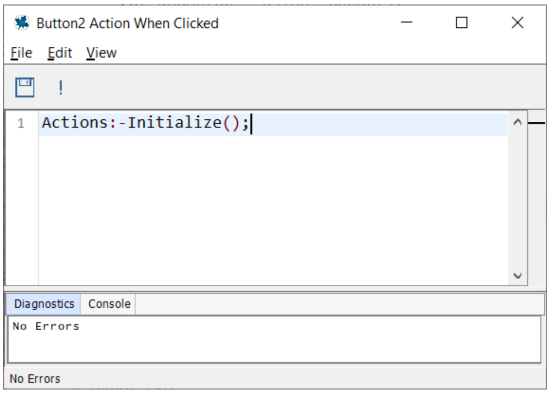
In the Math App presented in Figure 3.5, we are asked to determine the area of a triangle given the lengths of the three edges.

Three Math Containers are used to display the edge lengths of the triangle since these are randomly generated and thus change values each time the Math App is run. For the input, we use a Text Area named "Input". Once some input has been entered, the procedure GetInput in the Startup Code will be executed. This outputs a variable containing the input as interpreted as math by Maple, and a string containing comment to the input type. The math is presented in the "Interpreter" Math Container. If the input entered has the correct type (in this case a decimal number, integer or a fraction) the comment will be "", meaning that there was no comment. The procedure also outputs a grade, which is displayed in the "GradeDisplay" Text Area. Finally, we have two buttons. The code for what happens when you click on them can be seen in Figure 3.6. The "Check" button does the same as pressing enter after typing in an answer in the "Input" box: It executes the GetInput procedure. The "Reset" button calls the Initialize procedure, which resets the grade and clears all input-dependent components: Input, Interpreter, Reporter and GradeDisplay, and then calls the SetProblem procedure to generate variables for the new instance of the problem.

The Content Changed Code for the "Input" text area component is similar to the code for when "Button1" is clicked: the GetInput() procedure is called.



(a) Action when clicked code for the "Check" button.



(b) Action when clicked code for the "Reset" button.

Figure 3.6

4 The Startup Code

The startup code is the back-end of the Math App. It is written in the built-in Startup Code Editor. In the startup code, we typically write multiple procedures, which work similar to functions in other programming languages. This means that they have names and can have input and output. Inside each procedure, we can declare variables local or global to the procedure, which can then be accessed elsewhere in the startup code. If a variable is declared local to the Actions module, and then changed in one of the procedures, the variable will remain changed. In Maple, language procedures are defined as $\mathrm{FuncName} :=$ `proc`(input) and closed by `end proc`:. As such, output is defined directly in the procedure by the "return" command. In the templates we have created, most procedures do not directly return any values.

4.1 The Built-in Maple Startup Code Editor

The startup code editor has built-in live syntax checking which updates automatically when changes are made to it. There is also some automatic indentation. However, the indentation done by the editor is lacking, and it thus requires a bit of extra care to make sure the code is somewhat easily readable. The editor is seen in figure 4.1.

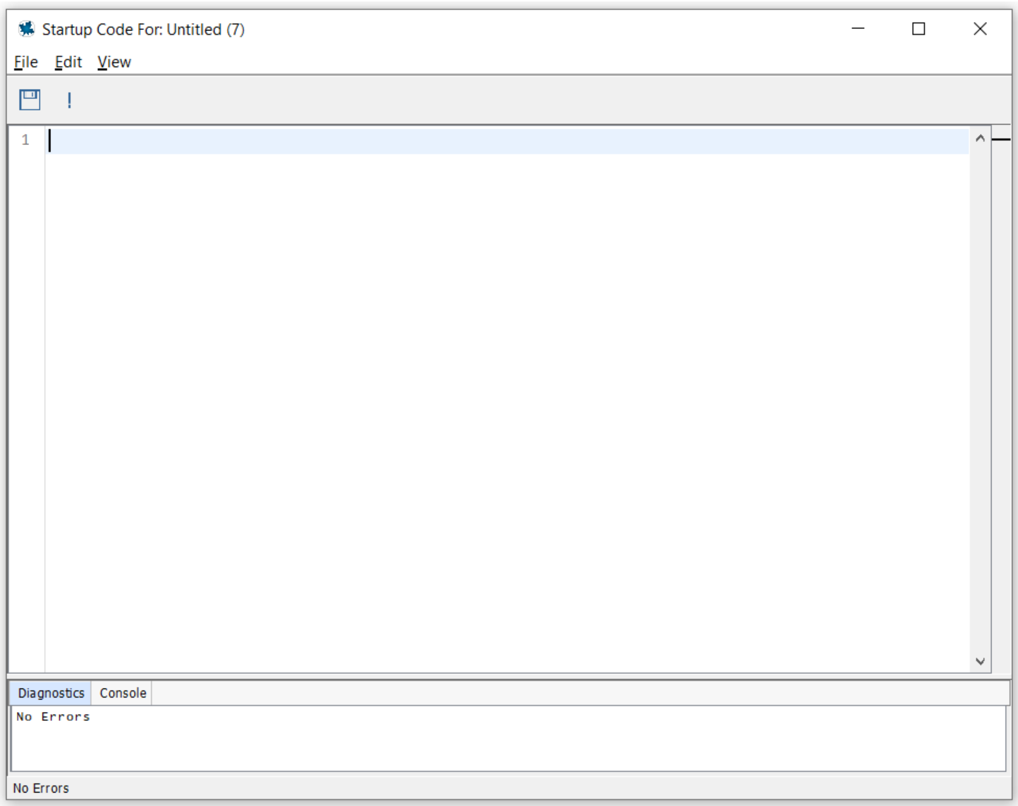


Figure 4.1: The Startup Code editor as it appears when you first open it

Errors in the code will be shown in the Diagnostics tab while Console can be used for debugging etc. If an error prevents the startup code from being executed, an error message will pop up in a new Maple window. The error message will usually display the name of the procedure in which the error was encountered.

The editor will highlight keywords by coloring them in certain colors. Comments (created by "#") are written in a mossy green, and keywords such as "module", "local", "export", "proc", "if", "return" etc. are made bold and blue. Keywords such as "integer", "range", "complex" etc are written in a bold burgundy color, while strings are written in a purple color. Symbols such as "+", "=", "[" etc. are also written in burgundy. It is not currently possible to change the color scheme in the editor. It can also suggest text while you are writing, however, it is does not seem to be properly predictive.

When editing the startup code inside the startup code editor, pressing "Ctrl + S" saves the changes to the code and "Ctrl + E" executes the startup code. The editor does not save automatically therefore it is important to save frequently when editing the startup code.

4.2 Startup code

The very simplest Math App can be written directly on the first line of the startup code. If we want any sort of communication between the front-end and the back-end, however, we need to place the main bulk of the code inside an "Actions module". This main bulk consists of the procedures that make up the Math App. If we want the Math App to work with the Möbius platform, there needs to be a Grade procedure that is exported. This is used by Möbius to extract the grade from the Math App. These procedures are made in the same manner as when using procedures in a regular Maple worksheet.

We have had the best success with structuring the Startup Code according to the following list:

- Packages to include (e.g. plottools, LinearAlgebra etc.)
- Macros
- Actions module
  - Declaring export, local and global variables
  - Procedure for setting up the problem
  - Procedure for initialization
  - Procedure for checking the input
  - Procedure for grading
- Calling the initialization action

For randomized exercises, we start the startup code with *restart* to ensure that the seed for randomization is different every time.

Using the simple Math App template shown in Figure 3.5, we will now go through the startup code in sections.

4.2.1 Preamble

First we type "restart;" and then we include packages and macros. For the triangle question we include the packages RandomTools, combinat, ListTools, LinearAlgebra, Plots and plottools. The inclusion of the DocumentTools package is implicit and does not need to be declared. We define macros for setting a property of a component, retrieving a property of a component and for generating random objects using the RandomTools package.

This becomes the following:

```
restart;

############### INCLUDE RELEVANT PACKAGES ###############
with(RandomTools): with(combinat): with(ListTools):
with(LinearAlgebra): with(plots): with(plottools):

############### Creating practical macros ###############
# i.e. "shortcuts" for commands we will use frequently
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):
```

Listing 4.1: Preamble for the simple Math App template

The **DocumentTools[SetProperty]** command takes the following input: id of the component to change a property of, the attribute we want to change, the value we want to change it to and "refreshopt". The "refreshopt" option can be true or false and specifies whether the document should be updated immediately. By default, the document will not be updated immediately if the command is called by an embedded component. This is the command we use to display variables generated randomly for a question. It is also using this command we can display plots using the Plot component.

The **DocumentTools[GetProperty]** command queries a specific component for the value of a property. Often, it is used to retrieve user input.

The **RandomTools[Generate]** command generates a particular random object that is determined by the expression input into the function.

The macros are set using the Maple command $\mathrm{macro}(\mathrm{e1}, \mathrm{e2}, \ldots, \mathrm{en})$ where $e[1], e[2], ..., e[n]$ are zero or more equations. For every occurrence of the macro abbreviation in the startup code, the command will replace it with the full expression when the code is run.

4.2.2 Actions module

The Actions module contains the bulk of the startup code. The first lines in the Actions module should define the procedures to be exported and declare local and global variables.

For the Simple Math App template, the Actions module is written as follows:

```
Actions := module()
    export Initialize, GetInput, SetProblem, Grade, InputCheck;
    # Exported procedures can be accessed outside of the startup code,
    #  e.g the Grade procedure is used by Möbius to retrieve the grade.
    local SOL;
    # stores the solution for the question, but is not accessible outside of
       the Actions module
     global Grad, grad, report, intepretation;

    # Procedures....
    SetProblem := proc()
        # ...
    end proc:

    InputCheck := proc()
        # ...
    end proc:

```

```
    Initialize := proc()
        # ...
    end proc:

    GetInput := proc()
        # ...
    end proc:

    Grade := proc()
        # ...
    end proc:

end module:
```

Listing 4.2: Startup code for the Actions module in the template for a Simple Math App

As can be seen in the above startup code section, we export an Initialize procedure, a GetInput procedure, a SetProblem procedure, a Grade procedure and an InputCheck procedure. We will now go through each of these.

The SetProblem procedure

For the triangle problem, we need three randomly generated integers for the edge lengths. The first thing we do is therefore to call $\text{randomize}()$. This function generates a new random seed. We also want to make sure that the area of the triangle is not complex. Before progressing, we calculate the area of the generated triangle using Heron's formula using additional variables "s" and "ans". Once this has been confirmed, we will use the SetProperty command to display the edge lengths in the components "Given1", "Given2" and "Given3". Lastly, the area is stored in the "SOL" variable.

```
SetProblem := proc()

    randomize();
    Grad := 0;

    ####################################################
    ###  CHANGE THE FOLLOWING TO FIT YOUR QUESTION    ###
    ####################################################

    # add or remove variables according to the question
    local a, b, c, s, ans;

    # Generating the 3 sides randomly
    a := RG(integer(range = 1..10));
    b := RG(integer(range = 1..10));
    c := RG(integer(range = 1..10));

    # Calculating the area
    s := (1/2)*(a+b+c);
    ans := eval( sqrt(s*(s-a)*(s-b)*(s-c)) );

    # Avoiding non-existing triangles:
    if evalb( whattype(evalf(ans)) = complex(extended_numeric)) then
        Actions:-SetProblem();
        #If we get a complex area, we call the SetProblem procedure again.
        # However, make sure that the problem is solvable to avoid infinite
           recursion.
        return;
    end if;

```

```
30      # Displaying the side lengths to the user using the DocumentTools command:
            DocumentTools:-SetProperty, defined by macro as "SP".
31      # SetProperty changes a property of a component to the wanted value:
32      # SP('Component name', 'property to change (often value)', value to change
            to);
33      SP('Given1','value',a);             ###### COMPONENTS
34      SP('Given2','value',b);             ###### COMPONENTS
35      SP('Given3','value',c);             ###### COMPONENTS
36
37      # Storing the solution as a list in a variable SOL,
38      #   declared local to the Actions module.
39      SOL := [ans];
40
41  end proc:
```

In the SetProblem procedure we generate the integers using the macro we defined previously. The RandomTools[Generate] command takes the type of number to randomly generate as input. In this case we want an integer in the interval between 1 and 10.

To check if the computed area actually corresponds to an existing traingle, we use an if statement. Here, we see if the type of the evaluated answer is equal to a complex number. We force the comparison to be evaluated as a boolean using $\mathrm{evalb}()$. If it is true that the answer type is complex, we want to generate new numbers. Calling ”return” means that we should eventually come back to the ”original” call to the SetProblem procedure. This part of the procedure can be ignored if the question variables are generated in a way that we always know that the triangle exists, i.e. if the square root cannot be complex.

On line 33 to 35 we change the values of the three components ”Given1”, ”Given2” and ”Given3” to be the three edge lengths. Note that the ”refreshopt” option is not in use here. Finally the solution is stored as a list inside the variabel SOL, which is local to the Actions module.

Since the variable is local to the Actions module, it is not accessible outside of the startup code, and therefore it cannot be accessed directly in the Maple worksheet.

The InputCheck procedure

This is the most advanced procedure in the startup code. This was written as a collaboration between Steen Markvorsen and Line Glade and the current version is able to handle many types of input. In this version, the user can get points for being close to the correct answer. It also accepts an answer which is very close to the correct answer, as this might be due to round-off error either by Maple or by the user. We have tried to highlight the areas in the code that are to be changed if more harsh grading is desired.

If you are not familiar with Maple programming, we recommend that you not spend too much time trying to understand this procedure.

We input the code below:

```
#### Checking if the input is of the correct type
InputCheck:= proc(raw::string, sol::list)
    local unpack, num, numsol, listnum, listnumsol,
    userflat, solflat, soldiff, solref, GO1, GO2, names,
    differ, i, solflatfct, ran, q, w;
    global  grad, report, interpretation;

    GO1:= 1;

```

```
    try
        parse(raw);
    catch :
        GO1:= 0;
    end try;

    if evalb(GO1=1) then
        report:= "";
        unpack:= [eval(parse(raw))];
        num:= nops(Flatten(unpack, 1));
        numsol:= nops(Flatten([sol], 1));
        listnum:= nops(Flatten(unpack));
        listnumsol:= nops(Flatten([sol]));

        if not(num=numsol and listnum=listnumsol and indets(evalf(unpack), name) = indets(evalf(sol), name)) then
            GO2:= 0;
            report:= "Wrong type or missing input";
            interpretation:= "";
        else
            GO2:= 1; report:= "";
            if (num=3 and listnum=9) or (num=2 and listnum=4) then
                interpretation:= Matrix(Flatten(unpack, 1));
            else
                interpretation:= Flatten(unpack, 1);
            end if;

            if (num=1 and listnum=1) then
                interpretation:= unpack[1];
            end if;
        end if;
    else
        report:= "Syntax error"; interpretation:= "";
    end if;

    if evalb(GO1=1 and GO2=1) then
        userflat:= evalf(Flatten(unpack));
        solflat:= evalf(Flatten(sol));
        names:= indets(evalf(unpack), name);

        if nops(names) > 0 then
            differ:= unapply((solflat - userflat), seq(names[i], i=1...nops(names)));
            solflatfct:= unapply(solflat, seq(names[i], i=1...nops(names)));
            ran:= rand(-10...10);

            # Here we determine the maximum difference in value when evaluating the functions in 20 randomly generated points.
            soldiff:= max(seq(evalf[3](Norm(Vector(differ(seq(ran(), w=1...nops(names)))),2)), q=1...20));
            solref:= max(seq(evalf[3](Norm(Vector(solflatfct(seq(ran(), w=1...nops(names)))),2)), q=1...20));

        else
            if solflat=[] and userflat=[] then
                soldiff:= 0.0;
                solref:= 1.0;
            else
                soldiff:= evalf[3](abs((solflat - userflat)[1]));
                solref:= evalf[3](abs(solflat[1]));
            end if
        end if;
```

```
67
68         ###################################################################
69         ##### CHANGE THE FOLLOWING IF DIFFERENT GRADING IS WANTED #########
70         ###################################################################
71
72         if soldiff <= 0.001*solref then
73             grad := 1.0;
74         elif soldiff <= 0.1*solref then
75             grad:= 0.5;
76         else
77             grad:= 0.0;
78         end if;
79
80         ###################################################################
81         ######################### END EDIT ################################
82         ###################################################################
83
84     else
85         grad:= 0.0
86     end if;
87
88 end proc: # end InputCheck( userInput, SOL)
```

Listing 4.3: The InputCheck procedure for the simple Math App template

Since the code is rather long, we will go through it in smaller sections. `InputCheck := proc( raw::string, sol::list)` means that the procedure takes two input: a string called "raw" and a list called "sol". "raw" here refers to the raw input taken directly from the user input and "sol" is short for solution. The objective of this function is to compare the input string to the solution list.

Various local variables are declared for later use. The variables "grad", "report" and "interpretation" are double declared, as they were already declared in the Actions module. This ensures that the changes we make to the variables also change the global variables.

We define "GO1" to be 1. This is a control variable to be able to see if Maple was able to successfully parse the user-input. We try to parse the user-input, and if unsuccessful, the variable GO1 is changed to 0.

To be able to grade the answer, we have to be able to parse it, meaning that Maple can reformat it in a way that it can understand. Thus we proceed if GO1 is still equal to 1.

On line 17 to 22, the user-input is put into a list, and the number of operands (e.g. number of elements in a list, number of elements in a matrix or number of terms in an equation or function) is determined. Two lists are also created containing the operands of the user-input and the operands of the solution.

On line 24, we try to determine if the number of operands is the same for the user-input and the solution. This is done in order to also handle vector- and matrix-functions of less than three dimensions. If this is not the case we declare "GO2" (another control variable) to be 0, and define "report" to be "Wrong type or missing input". The interpretation cannot be shown and is therefore defined as an empty string. If the solution and the user-input have the same operands GO2 is set to be 1 and the report is set to be an empty string, as the input type is deemed to be correct. Next step is to define the interpretation variable such that the user-input can be presented in the "Interpreter" Math Container component in the worksheet. Line 30 to 38 formats the interpretation in different ways, depending on the formatting of the raw user-input and the solution.

Line 40-42 is executed if GO1 was not equal to 1, meaning that Maple was not able to parse the user-input. In this case the report is defined as ”Syntax error” and the interpretation variable is set to an empty string.

Next up is the grading. If both GO1 and GO2 are 1, the input is deemed to be the correct type. ”userflat” and ”solflat” are variables containing the input and the solution, respectively, as one-dimensional lists. The command ”indets” is used to determine if the input is a function.
On line 49-57 is the case when the input and solution are functions. As written in the comment on line 54, the maximum difference in value is determined when comparing the two functions over 20 randomly generated points ranging from -10 to 10. If the functions are not defined at this interval, some change might be needed. However, since we have a comparison using the Maple command `max()`, it seems to be the case that if both functions are undefined at a point, the difference will be zero. This would however require some changes to the InputCheck procedure, as we would need to make sure that the sequence of points we test the functions at are the same. A simple workaround would be to add a third input to the InputCheck function containing the interval in which the functions are defined. During the testing of this InputCheck procedure, we have not run into any issues regarding this check. However, it should be noted that the checks were done primarily in the context of differential geometry.

If the user-input and the solution are not functions, we can compare them more directly. We define ”soldiff” to be the absolute value of the difference between the user-input and the solution, and ”solref” as the absolute value of the solution.

The actual grading is then done on lines 72-78. A full point is given if the difference between the solution and the user-input is less than .1% of ”solref”, and a half point is given if the user-input is within 10% of ”solref”. If the difference is larger than 10% of ”solref”, zero points is awarded for the question.

Line 85 sets the grade to 0 if the input has a syntax error or is of the wrong type.

The main ”output” of this procedure is the changes to ”interpretation”, ”report” and ”grade”, which can then be displayed and stored in appropriate values inside the procedure from which the InputCheck procedure was called.

The Initialize procedure

This procedure is the first to be executed. This is the procedure where the initial grade is set to zero, input areas are set to be empty and lastly the SetProblem procedure is called. The code is pretty simple and should be self-explanatory:

```
##### Initialization
Initialize := proc()
    # Setting the grade to zero initially
    Grad := 0;

    # clearing the boxes
    SP('Input','value',NULL);                      ###### COMPONENTS
    SP('GradeDisplay', 'value', NULL);             ###### COMPONENTS
    SP('Reporter', 'value', NULL);                 ###### COMPONENTS
    SP('Interpreter', 'value', NULL);              ###### COMPONENTS

    # setting up the problem according to the previous procedure
    Actions:-SetProblem();

```

```
end proc: # end Initialize()
```

Listing 4.4: The intitialization procedure for the simple Math App template

The GetInput procedure

This procedure retrieves the user-input and calls the InputCheck function. The grade is then stored in a variable "Grad", which was declared globally for the Actions module.

```
GetInput := proc()
    local userInput;

    # Getting the input from the user
    userInput := GP('Input','value');                  ###### COMPONENTS
    # Checking the input with the known solution
    Actions:-InputCheck(userInput, SOL);

    # Defining the grade
    Grad := grad;
    # Displaying the grade
    SP('GradeDisplay','value', Grad);                  ###### COMPONENTS

    # Displaying the interpretation if possible
    if (report = "") then
        SP('Interpreter','value', interpretation); ###### COMPONENTS
    else
        SP('Interpreter','value',NULL);                ###### COMPONENTS
    end if;

    # Displaying the report, i.e. comments on the input type.
    SP('Reporter','value',report);                     ###### COMPONENTS

end proc: # end GetInput()
```

Listing 4.5: The GetInput procedure for the simple Math App template

The Grade procedure

This procedure can contain more, but we found that the following works well when the Math App is uploaded to Möbius.

```
########################################
# GRADE OUT
########################################
Grade := proc();
    return Grad;
end proc: # end Grade()
```

Listing 4.6: The Grade procedure for the simple Math App template

4.2.3 Executing the startup code

The final command in the startup code is a call to the Initialize procedure. The full code without breaks can be found in the appendix.

5 Templates

In this chapter we will go through the other templates as we did with the previous template. To avoid repeating ourselves, we will skip parts that are unchanged compared to the first template.

5.1 Math App Template 2: Multiple subquestions

This Math App is similar to the previous Math App, but now we have five subquestions each with their own answer boxes. The worksheet can be seen in Figure 5.1.

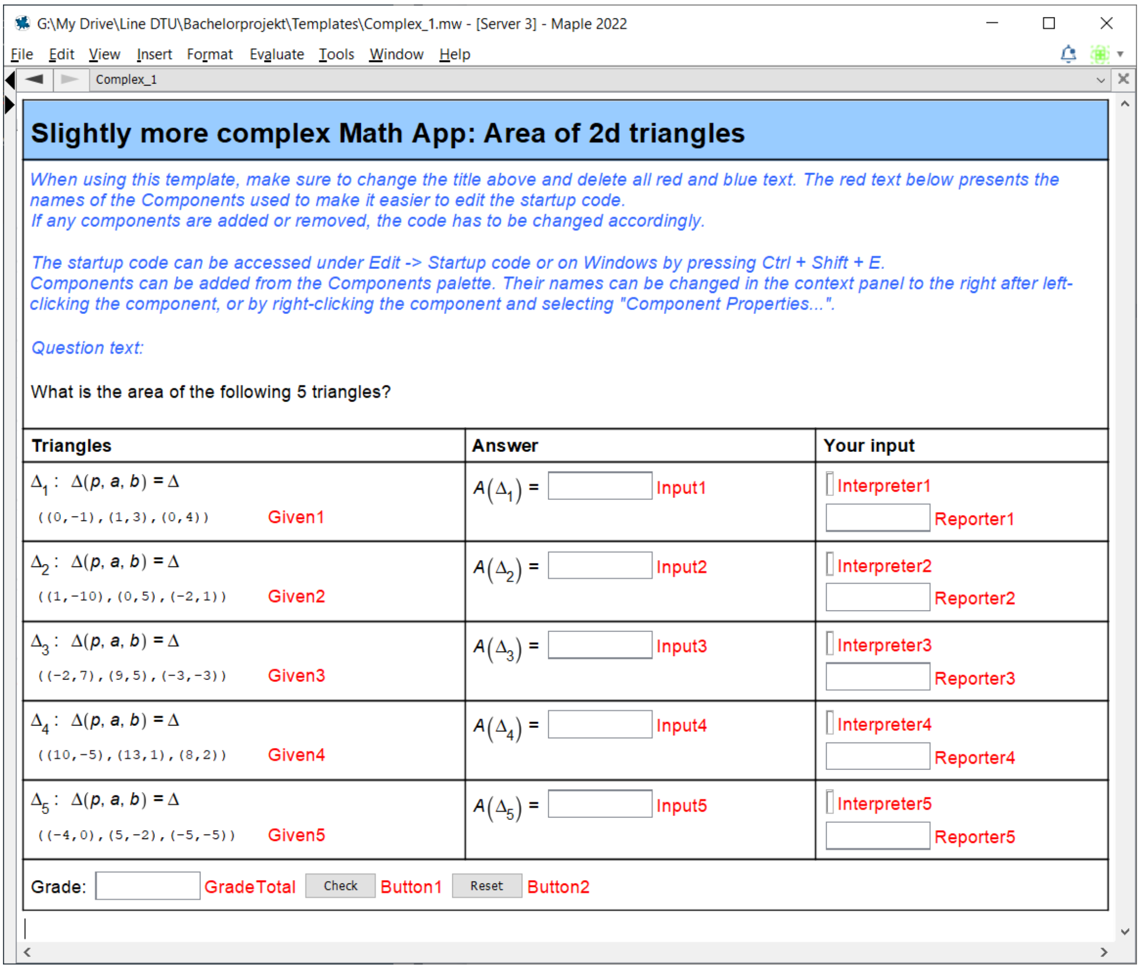


Figure 5.1: The worksheet for Math App Template 2

In this exercise we again are asked to determine the area of triangles, but this time we are given the triangles by two edge-vectors a, b from the starting point p.

5.1.1 The Startup Code

The preamble is identical to the preamble for the first template:

```
restart;

########## INCLUDE RELEVANT PACKAGES ############
with(RandomTools): with(combinat): with(ListTools):
```

```
with(LinearAlgebra):with(plots): with(plottools):

######### Creating practical macros ###########
# i.e. "shortcuts" for commands we will use
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()

    # ...

end module;

# Initializing the Math App
Actions:-Initialize()
```

Listing 5.1: Preamble for Template 2

The only change in the preamble of the Actions module is the addition of a global variable ”GradTotal”. The total grade is now given based on the sum of each of the subquestions (each weighted $1/5$).

```
Actions := module()
    export Initialize, GetInput, SetProblem, Grade, InputCheck;
    local SOL;
    global Grad, GradTotal, grad, report, intepretation;

    #### Setting up the problem
    SetProblem := proc()
        # ...
    end proc; # end SetProblem

    #### Checking if the input is of the correct type
    InputCheck:= proc(raw::string, sol::list)
        # ...
    end proc;

    ##### Initialization
    Initialize := proc()
        # ...
    end proc;

    ##### GetInput
    GetInput := proc( n ) # Retrives the input for one box at a time
        # ...
    end proc;

    #### Grading
    Grade := proc();
        # ...
    end proc: # end Grade()

 end module;
```

Listing 5.2: Actions module for Template 2

The InputCheck procedure and the Grade procedure are identical to the procedures in the first template, with the only exception being that the Grade procedure returns "GradTotal" instead of "Grad".

The SetProblem procedure

```
#### Setting up the problem
  SetProblem := proc()
    randomize();

    local ans, T, i, j, F, p, a, b;

    # Making random triangles
    # - note it is very unlikely for the same triangle to appear
    for i from 1 to 5 do
      p := [RG(integer(range = -5.....10)), RG(integer(range = -10.....10))];
      a := [RG(integer(range = -10.....15)), RG(integer(range = -5.....5))];
      b := [RG(integer(range = -8.....12)), RG(integer(range = -10.....5))];

      T[i] := [p,a,b];

      # Calculating the answers
      SOL[i] := [(1/2)*abs(Determinant(<T[i][2][1],T[i][2][2]| T[i][3][1], T[i
         ][3][2]>))];
    end do;

    # Writing the formulas for the triangles:
    for i from 1 to 5 do
      F := "(";
      for j from 1 to 3 do
        F := cat(F,"(", T[i][j][1], ",", T[i][j][2], ")");
        if not evalb(j = 3) then
          F := cat(F, ",");
        end if;
      end do;
      F := cat(F, ")");
      SP(cat('Given',i),'value',F);
    end do;

  end proc; # end SetProblem
```

Listing 5.3: SetProblem procedure for Template 2

In the SetProblem procedure we use a for loop to generate all five triangles. Inside the for loop we store the triangles in a variable T and the solutions in a variable SOL. We want to display the triangles on the form (p, a, b). Using the Text Area component we can display $T[i]$ as $[p, a, b]$. We want it written with parentheses. To do this we use the Maple command `cat` which can concatenate strings. For each of the five triangles we define a variable F, which contains the string. To create F we concatenate the variables with appropriate placing of commas and parentheses. Finally the string is displayed in the appropriate component.

The Initialize procedure

```
##### Initialization
Initialize := proc()
    local i;
    # Setting the grade to zero initially
    GradTotal := 0;
```

```

    # clearing the boxes
    for i from 1 to 5 do
        Grad[i] := 0;
        SOL[i] := 0;
        SP(cat('Input',i),'value',NULL);
        SP(cat('Reporter',i), 'value', NULL);
        SP(cat('Interpreter',i), 'value', NULL);
    end do;

    SP('GradeTotal','value',NULL); # clearing the grade text area

    # setting up the problem by calling the SetProblem procedure
    Actions:-SetProblem();

end proc: # end Initialize()
```

Listing 5.4: Initialize procedure for Template 2

The Initialize procedure is very similar to the procedure used to initialize Template 1. Here we have used indexing and a for loop to clear all the relevant components.

The GetInput procedure

```
GetInput := proc( n ) # Retrives the input for one box at a time
    local userInput;

    # Getting the input from the user
    userInput := GP(cat('Input',n),'value');
    # Checking the input with the known solution
    Actions:-InputCheck(userInput, SOL[n]);

    # Defining the grade
    Grad[n] := grad;

    # Updating the grade:
    GradTotal := evalf[3]((Grad[1] + Grad[2] + Grad[3] + Grad[4] + Grad[5])/5)
        ;

    SP('GradeTotal','value',GradTotal);

    # Displaying the interpretation if possible
    SP(cat('Interpreter',n),'value',interpretation);

    # Displaying the report, i.e. comments on the input type.
    SP(cat('Reporter',n),'value',report);

end proc: # end GetInput()
```

Listing 5.5: GetInput procedure for Template 2

The GetInput procedure is called with the appropriate indexing from each of the Input Components in the worksheet. This means that the ”Edit Content Changed Code” for Input1 is `Actions:-GetInput(1)`, etc. The advantage of this is that the Reporter components for the other subquestions is kept empty, instead of showing ”Wrong type or missing input”. The Grad variable is created as a vector containing the grade for each of the subquestions. This was initialized to be a zero-vector which means that the total grade always can be computed. The new total grade will be updated every time input is entered.

5.2 Math App Template 3: Checkboxes

In this Math App, the input is now whether or not a checkbox is selected. That a checkbox is selected is equivalent to the value of the component being "true". Similar to the previous template, we again have five triangles on the form (p, a, b). The worksheet can be seen in Figure 5.2.

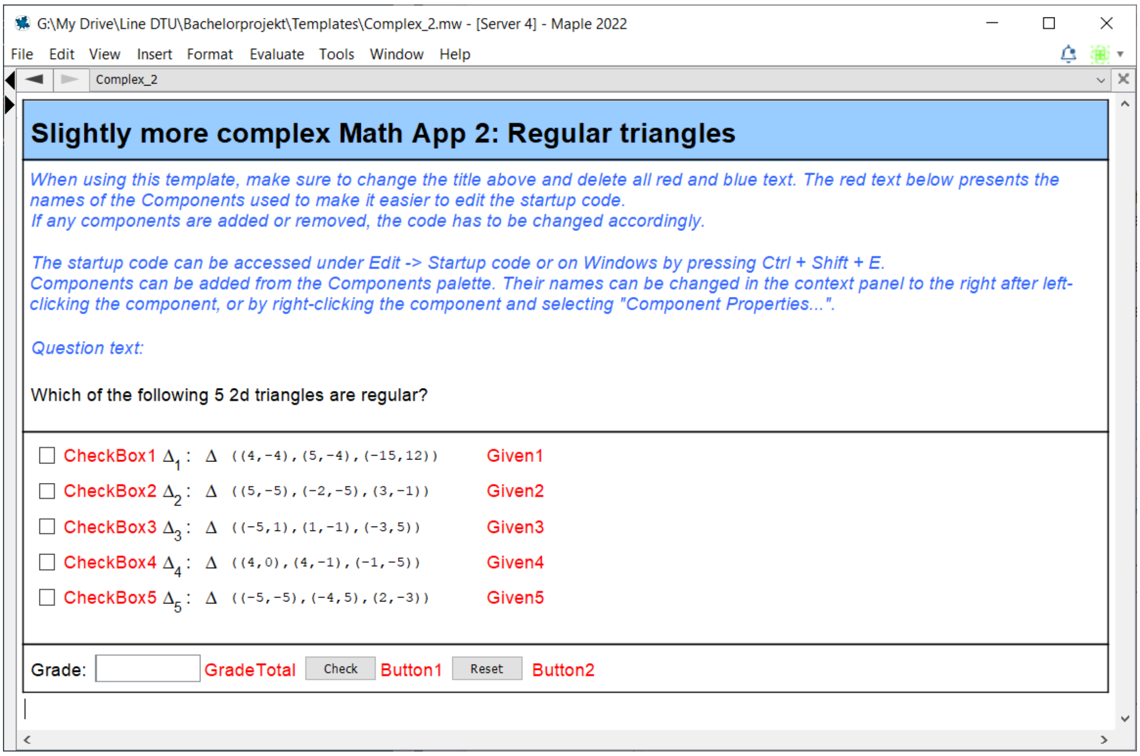


Figure 5.2: The worksheet for Math App Template 3

5.2.1 The startup code

Again the preamble is similar to the previous templates. The main difference between this template and the previous ones is that the InputCheck procedure has been removed. When we get input through checkboxes, there is no need to parse the input. This means that there is no risk of encountering errors when retrieving the input. The grading procedure is identical to the previous grading procedures and will therefore not be discussed.

```
restart;

########## INCLUDE RELEVANT PACKAGES ############
with(RandomTools): with(combinat): with(ListTools):with(LinearAlgebra):with(
    plots): with(plottools):

######### Creating practical macros ###########
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()
  export Initialize, GetInput, SetProblem, Grade;
```

```
  local SOL;
  global Grad;

  #### Setting up the problem
  SetProblem := proc()
    # ...
  end proc; # end SetProblem

  ##### Initialization
  Initialize := proc()
    # ...
  end proc; # end Initialize()

        #### Retrieving the input and grading it
  GetInput := proc( )
    # ...
  end proc; # end GetInput()

        #### Grading
  Grade := proc();
      return Grad;
  end proc; # end Grade()

end module;

Actions:-Initialize();
```

Listing 5.6: Preamble and Actions module for Template 3

Removing the InputCheck procedure makes the startup code shorter and somewhat more simple.

The SetProblem procedure

The SetProblem procedure is quite similar to the previous template. The triangles are generated and displayed using the same method as before, with the exception that we this time want to be sure that at least one of the generated triangles are non-regular.

```
#### Setting up the problem
SetProblem := proc()
    # add or remove variables according to the question
    local ans, T, p, a, b, r, i, j, F, ar;

    randomize();

    ###  Making random triangles
        # Most of the triangles will be regular
        # We want to make sure at least one triangle is non-regular
    r := RG(integer(range=1..5)); # the triangle we will force to be non-regular

    for i from 1 to 5 do
        p := [RG(integer(range = -5...5)), RG(integer(range = -5...5))];
        a := [RG(integer(range = -5...5)), RG(integer(range = -5...5))];
        if evalb(i = r) then
            b := RG(integer(range=-10...-1))*a;
        else
            b := [RG(integer(range = -3.....5)), RG(integer(range = -5.....5))];
        end if;

        T[i] := [p,a,b]; # storing the vectors of the triangle
```

```

        # Calculating the areas
        ar[i] := evalf((1/2)*abs(Determinant(<T[i][2][1],T[i][2][2]| T[i
            ][3][1], T[i][3][2]>)));
        if not evalb(evalf(ar[i]) = 0) then
            ans[i] := true;
        else
            ans[i] := false;
        end if;
    end do;

    ##### Writing the triangles on the wanted form into the respective "Given"
         TextAreas
    for i from 1 to 5 do
        F := "(";
        for j from 1 to 3 do
            F := cat(F,"(", T[i][j][1], ",", T[i][j][2], ")");
            if not evalb(j = 3) then
                F := cat(F, ",");
            end if;
        end do;
        F := cat(F, ")");
        SP(cat('Given',i),'value',F);
    end do;

    # Storing the solution
    SOL := [ans[1], ans[2], ans[3], ans[4], ans[5]];

end proc: # end SetProblem
```

Listing 5.7: SetProblem procedure for Template 3

In order to make sure at least one triangle is non-regular, we generate a random number r, denoting the triangle to force to be non-regular. We then generate the point p and the vector a as usual. Before generating the vector b, we check if the for loop variable i is equal to r. If this is the case, we set the vector b to be equal to the a vector multiplied by some negative scalar. This creates a triangle with an area of zero, thus making it non-regular. If i is not equal to r, the b-vector is generated as usual. Due to the vectors being generated randomly there is always a chance that the resulting triangle will be degenerate.
The triangles are again stored in a variable T and the areas are computed. Instead of storing the areas in the solution variable, we store a boolean stating whether or not they are regular. Using the same method as before, we display the triangles in the appropriate components on the desired form. The solution is stored in the variable SOL.

The Initialize procedure
To initialize the Math App, we unselect all checkboxes, clear the grade area, set the initial grade to zero and then call the SetProblem procedure:

```
##### Initialization
Initialize := proc()
    local i;
    # Setting the grade to zero initially
    Grad := 0;

    # Clearing the checkboxes
    for i from 1 to 5 do
        SP(cat('CheckBox',i),'value', false);
    end do;
```

```

    # Clearing the grade area
    SP('GradeTotal','value',NULL);

    # setting up the problem according to the previous procedure
    Actions:-SetProblem();

end proc: # end Initialize()
```

Listing 5.8: Initialize procedure for Template 3

The GetInput procedure

In order to retrieve and grade the input, we define two counters, `u_inc` and `u_cor` to count the number of incorrect and correct choices, respectively. These are used for partial grading.

```
GetInput := proc( )
    local userInput, u_inc, u_cor, grad, i, sol;

    grad := 0: u_inc := 0: u_cor := 0;
    # We use u_inc and u_cor to denote the number of correct and incorrect
       choices
    # this is used for partial grading

    for i from 1 to 5 do
        userInput := parse(GP(cat('CheckBox',i), 'value')); # checking if a
            box is checked
        sol := SOL[i];
        if evalb(userInput = sol) then
            u_cor := u_cor + 1;
        else
            u_inc := u_inc + 1;
        end if;
    end do;

    # Determining the grade
    grad := evalf[3]((u_cor)/ (u_cor + u_inc));

    Grad := grad;
    SP('GradeTotal','value',Grad);

end proc: # end GetInput()
```

Listing 5.9: GetInput procedure for Template 3

Instead of retrieving all of the input at once, we go through them one at a time, evaluating whether a check box has been marked correctly and incrementing the relevant counter. The grade is then computed as the number of correct answers divided by the total number of answers. Finally, the total grade is displayed.

One thing to note here, is that a user can be awarded points for selecting no checkboxes at all. This is due to the fact that we consider an unchecked checkbox a conscious choice, which can therefore be correct or incorrect.

5.3 Math App Template 4: Plot

This Math App is an attempt at a reverse engineered question where instead of having the user provide properties of a defined, given object, we want the user to define an object with a given property. The fact that the user can quickly check the area in the Math App makes this question somewhat simple to solve by guessing, but might encourage a different way to think of triangles. The worksheet for this Math App can be seen in Figure 5.3.

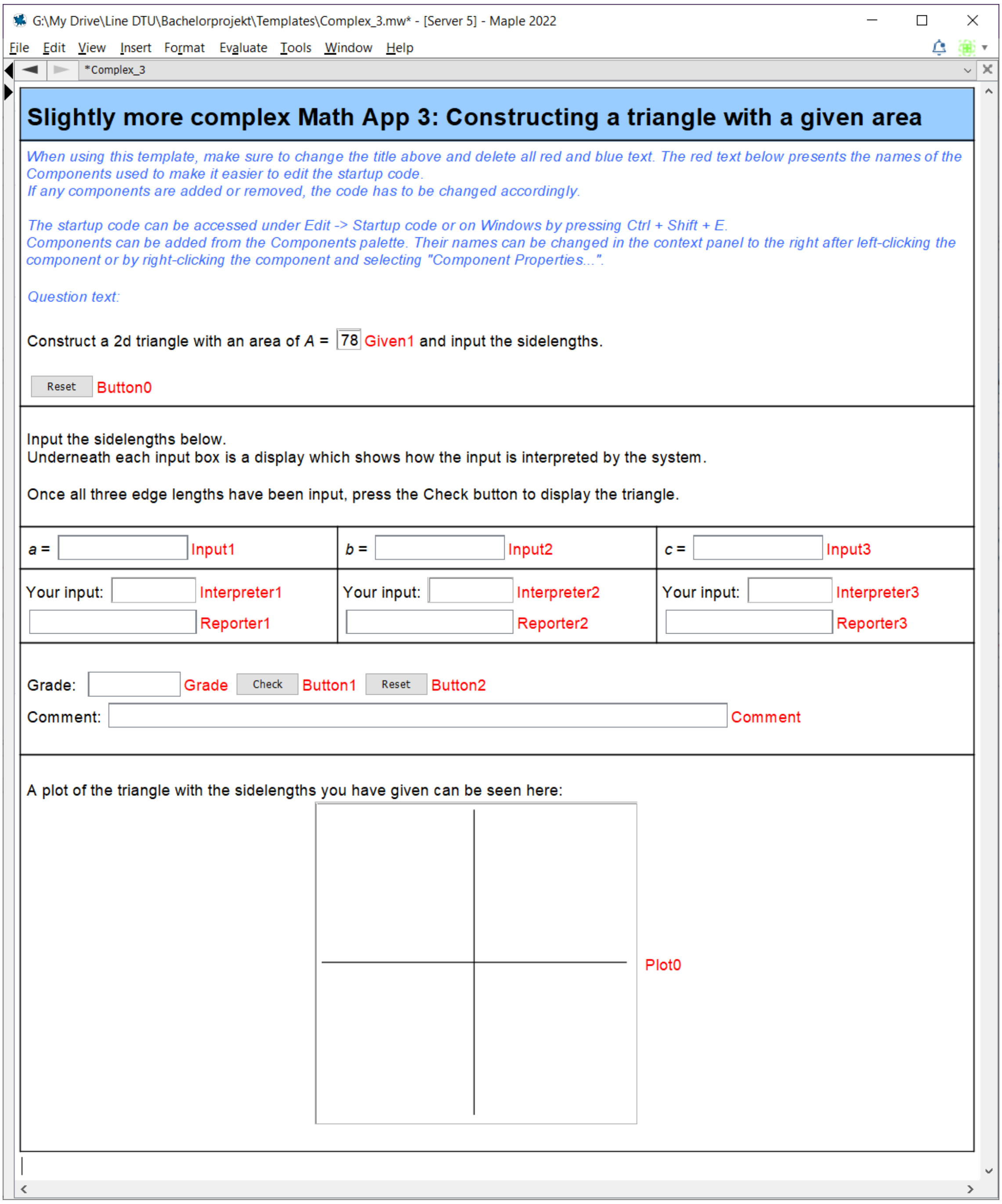


Figure 5.3: The worksheet for Math App Template 4

In this Math App, we have added additional commentary on the input. The user is asked to input the edge-lengths of a triangle with a specific area. The triangle with the edge-lengths as input is then shown in the Plot component.

5.3.1 The startup code

Again, not much is changed in the preamble. There are changes in the exported procedures: We have added a Plot procedure and a ShowInput procedure. We have also made some changes to the InputCheck procedure. Instead of taking a string and a list containing the solution as input, the procedure now only takes a string as input. The grading procedure is unchanged.

```
restart;

########## INCLUDE RELEVANT PACKAGES ############
with(RandomTools): with(plots): with(plottools):

######### Creating practical macros ###########
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()
  export Initialize, Plot, GetInput, ShowInput, SetProblem, Grade, InputCheck;
  local triang;
  global Grad, SOL, report, interpreter, a, b, c, GO;

  ##### Initialization
  Initialize := proc()
    # ...
  end proc: # end Initialize()

  #### Setting up the problem
  SetProblem := proc()
    # ...
  end proc: # end SetProblem

  #### Checking if the input is of the correct type
  InputCheck:= proc( raw::string )
    # ...
  end proc: # end InputCheck

  #### Creating the plot
  Plot := proc()
    # ...
  end proc:

  #### Retrieving the input
  GetInput := proc()
    # ...
  end proc: # end GetInput()


  #### Grading
  Grade := proc();
      return Grad;
  end proc: # end Grade()

end module:

Actions:-Initialize();
```

Listing 5.10: Preamble and Actions module for Template 4

The Initialize procedure

To initialize the Math App, we first set the grade variable to zero as well as a control parameter, "GO". All three input areas for edge lengths of the triangle are cleared. Similarly the "Grade" Text Area component, "Comment" Text Area component and "Plot0" Plot component are cleared. Finally, the Initialize procedure calls the SetProblem procedure and the Plot procedure.

```
##### Initialization
Initialize := proc()
    local i;
    # Setting the grade to zero initially
    Grad := 0.0;
    GO := 0;

    # clearing the boxes
    for i from 1 to 3 do
        SP(cat('Input',i),'value',NULL);
        SP(cat('Interpreter',i),'value',NULL);
        SP(cat('Reporter',i),'value',NULL);
    end do;

    SP('Grade','value',NULL);
    SP('Comment','value',NULL);
    SP('Plot0','value',NULL);

    # setting up the problem according to the previous procedure
    Actions:-SetProblem();
    Actions:-Plot();

end proc: # end Initialize()
```

Listing 5.11: Initialize procedure for Template 4

The SetProblem procedure

The SetProblem procedure for this Math App is very simple, since we only have to generate a random integer for the desired area and display this in the relevant component.

```
#### Setting up the problem
SetProblem := proc()
    randomize();

    # Generating a random area
    SOL := RG(integer(range = 1..100));
    SP('Given1','value', SOL);

end proc: # end SetProblem
```

Listing 5.12: SetProblem procedure for Template 4

The InputCheck procedure

This version of the InputCheck procedure is shorter than usual since the evaluation of the type of the solution has been removed. Instead, we use the fact that we know that the input should be a float or a decimal number.

```
#### Checking if the input is of the correct type
InputCheck:= proc( raw::string )
    local GO1, userinput;
    global report, interpreter;

    GO1:= 1;

    if (raw = "") then
        GO1 := 0;
        report := "Missing input";
        interpreter := "";
    else

```

```
        try
            parse(raw);
        catch :
            G01:= 0;
        end try;

        if evalb(G01 = 1) then
            userinput := parse(raw);

            # Checking if the input can be evaluated to a float
            if evalb( whattype(evalf(userinput)) = float) then
                report := "";
                interpreter := userinput;
            else
                report := "Wrong type of input";
                interpreter := "";
            end if;
        else
            report := "Syntax error";
            interpreter := "";
        end if;
    end if;

end proc: # end InputCheck
```

Listing 5.13: InputCheck procedure for Template 4

The Plot procedure

The creation of the plot is somewhat complex, as we use scaling to make the longest edge equal to one and function as the constitute the baseline of the triangle. This makes the placement of the triangle very simple, as we then have two of the three vertices without any computations. These two vertices are (0,0) and (1,0).

```
#### Creating the plot
Plot := proc()

    local LL, L, Sgam, Cgam, Sbeta, Cbeta, triangFill, a,b,c, s, Area;
    global G0;

    if evalb(G0 = 1) then
        a := evalf(parse(GP('Input1','value')));
        b := evalf(parse(GP('Input2','value')));
        c := evalf(parse(GP('Input3','value')));

        s := (1/2)*(a+b+c);
        Area := simplify(sqrt(s*(s-a)*(s-b)*(s-c)));

        # To make the plot we scale the triangle such that the longest edge
        #   becomes the baseline with a length of 1.
        LL:= sort([a,b,c], `>`); # Sorting the edge lengths
        L:= [1, evalf(LL[2]/LL[1]), evalf(LL[3]/LL[1])]; # Scaled edge lengths

        Sgam:= 2*Area/(LL[2]*LL[1]);
        Cgam:= (LL[1]^2 + LL[2]^2 - LL[3]^2)/(2*LL[1]*LL[2]);
        Sbeta:= 2*Area/(LL[3]*LL[1]);
        Cbeta:= (LL[1]^2 + LL[3]^2 - LL[2]^2)/(2*LL[1]*LL[3]);
        triang:= (
            line([0,0], [1,0], thickness=3, color=blue),
            line([0,0], L[2]*[Cgam, Sgam], thickness=3, color=blue),
            line([1,0], [1,0]+L[3]*[-Cbeta, Sbeta], thickness=3, color=blue),
            disk([0,0], 0.05, color=red),
            disk([1,0], 0.05, color=red),
```

```
            disk([1,0]+L[3]*[-Cbeta, Sbeta], 0.05, color=red)
        );
        triangFill:= polygon([[0,0], [1,0], [1,0]+L[3]*[-Cbeta, Sbeta]], color
            =cyan);
        SP('Plot0', 'value', plots:-display(triang, triangFill, axes=none,
            scaling=constrained));
    else
        # Draw empty plot
        SP('Plot0','value', plots:-display(plots:-pointplot([]), tickmarks
            =[0,0]));
    end if;

end proc:
```

Listing 5.14: Plot procedure for Template 4

We make use of some properties of a triangle to determine the third vertex. In order to plot the triangle we define the object ”triang” which is given by disks marking the three vertices and the lines connecting the vertices. The triangle is then colored in and displayed in the plot using the SetProperty command.

Since the Plot procedure can be called if the three edge lengths are not defined (correctly or at all), we have the initial check of the control parameter GO. If GO is equal to one, the edges make up a non-degenerate triangle and this triangle can be plotted. If GO is equal to zero, the edge lengths have not been defined correctly and therefore an empty plot is displayed.

The GetInput procedure

First we wish to retrieve the three edge lengths and check that they are of the correct type. Then we wish to determine if all edge lengths are positive, if the resulting triangle is non-degenerate and then if the area of the triangle is equal to the desired area. The result of these checks is given by the control parameter GO, which is set to zero if the edge lengths do not make up a non-degenerate triangle.

```
GetInput := proc()
    local aRaw, bRaw, cRaw, s, area, soldiff, solref;
    GO := 0;
    # Since we have multiple correct answers we cannot compare the input
        directly to a solution.
    # Instead we compute the area of the triangle with the edge lengths as
        input, and compare this to the wanted area.

    # Getting the input from the user
    aRaw := GP('Input1','value');
    bRaw := GP('Input2','value');
    cRaw := GP('Input3','value');

    Actions:-InputCheck( aRaw);
    SP('Reporter1','value',report);
    SP('Interpreter1','value',interpreter);
    if (report = "") then
        a := parse(aRaw);
        GO := 1;
    else
        GO := 0;
    end if;

    Actions:-InputCheck( bRaw);
    SP('Reporter2','value',report);
```

```
    SP('Interpreter2','value',interpreter);
    if (report = "") then
        b := parse(bRaw);
        GO := 1;
    else
        GO := 0;
    end if;

    Actions:-InputCheck( cRaw);
    SP('Reporter3','value',report);
    SP('Interpreter3','value',interpreter);
    if (report = "") then
        c := parse(cRaw);
        GO := 1;
    else
        GO := 0;
    end if;

    # If GO = 1 then all three edge lengths are floats.
    # However, we have some additional tests such as a, b and c being positive
    # and s > area.
    if (GO = 1) then
        if evalb( evalf( min(a,b,c)) <= 0) then
            Grad := 0.0;
            SP('Comment','value',"All edge lengths must be positive");
            GO := 0;
        else
            s := (a+b+c)/2;

            if ( evalf(s) > evalf(max(a,b,c)) ) then
                area := eval(sqrt(s*(s-a)*(s-b)*(s-c)));
                SP('Comment','value',cat("The area of the triangle is ", evalf
                    [5](area)));

                # comparing the area to the desired value
                soldiff := abs(evalf(area)-evalf(SOL));
                solref := evalf(SOL);

                # Partial points if the area is close to the desired value
                if evalb( soldiff < 0.01*solref ) then
                    Grad := 1;
                elif evalb( soldiff < 0.1*solref) then
                    Grad := 0.5;
                else
                    Grad := 0.0;
                end if;
            else
                Grad := 0.0;
                SP('Comment','value',"The input edge lengths does not make a
                    non-degenerate triangle");
                GO := 0;
            end if;
        end if;
    else
        Grad := 0.0;
        SP('Comment','value',"Please insert real numbers.");
    end if;

    # Displaying the grade
    SP('Grade','value',Grad);

    # Drawing the plot
```

```
84     Actions:-Plot();
85
86 end proc: # end GetInput()
```

Listing 5.15: GetInput procedure for Template 4

The control parameter GO is initially set to zero, however, this does create some redundancy in the code, since in some cases the parameter will again be defined as zero. On lines 7-40 we retrieve the input and check that all three edge lengths are of the correct type by calling the InputCheck procedure for each of them.

If the control parameter is equal to one, all three edge lengths (a, b and c) have been successfully defined. We can now determine if they actually make up a non-degenerate triangle. The first check is if the smallest of the edge lengths is equal to or less than zero. If this is not the case, we can proceed to compute the variable "s" as half of the sum of the three edge lengths. This comes from Heron's formula for computing the area of a triangle from its edge lengths. If s is larger than the largest edge length, we can evaluate the area. The area of the given triangle is then displayed in the "Comment" Text Area component.

The area of the given triangle is then compared to the desired area and the grade is defined. Finally, the grade is displayed and the Plot procedure is called.

In this worksheet, the Edit Content Changed Code is identical for the three Input Text Areas: The GetInput procedure is called. For the "Check" button, Button1, the click code calls the GetInput procedure alongside the Plot procedure. The Reset button calls the Initialize procedure, and thus resets the state of the worksheet.

Bibliography

[1] Maplesoft. *MapleCloud*. URL: https://www.maplesoft.com/products/maple/features/maplecloud.aspx.

[2] Maplesoft. *Text Area Component*. URL: https://www.maplesoft.com/support/help/maple/view.aspx?path=TextAreaComponent.

[3] Maplesoft. *Embedded Components*. URL: https://www.maplesoft.com/support/help/Maple/view.aspx?path=EmbeddedComponents&cid=22.

A Appendix

A.1 Templates

In the code for the templates we have attempted to highlight areas of the code that has to be changed in order to use the template to create a new problem.

A.1.1 Template 1

Worksheet

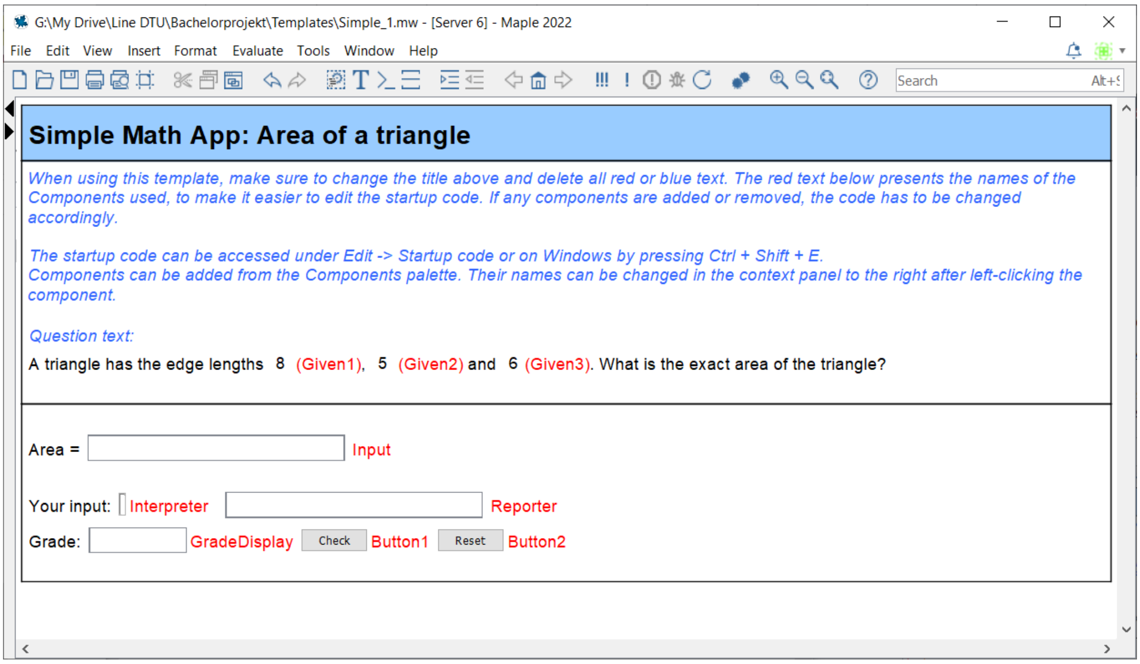


Figure A.1: The Worksheet for Template 1

Startup code

```
###############################################################
### Maple Math App Template: Simple 1
### - Line Glade, DTU 2022
###   version 2, 07.11.22: Added more comments
###
##############################################################
###
### In the code look for lines similar to:
###
###       #####################################################
###   ###  TEXT HERE, SHORT DESCRIPTION OF CHANGE       ###
###   #####################################################
###
### If any components are added or removed, pay attention to
### all markings of components, like "###### COMPONENTS".
### Remember to remove or add the relevant components in the worksheet.
###
###############################################################
restart;

########## INCLUDE RELEVANT PACKAGES ############
```

```
with(RandomTools): # Necessary, do not remove
with(combinat): with(ListTools):with(LinearAlgebra):with(plots): with(
    plottools):

######### Creating practical macros ###########
# i.e. "shortcuts" for commands we will use
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()
  export Initialize, GetInput, SetProblem, Grade, InputCheck;
  # Exported procedures can be accessed outside of the startup code,
  #  e.g the Grade procedure is used by Möbius to retrieve the grade.
  local SOL; # stores the solution for this module, not available to the
      students
  global Grad, grad, report, interpretation;
  # Global variables are available to the entire document

  #### Setting up the problem
  SetProblem := proc()
    randomize();
    Grad := 0;

    #####################################################
    ###  CHANGE THE FOLLOWING TO FIT YOUR QUESTION    ###
    #####################################################

    # add or remove variables according to the question
    local a, b, c, s, ans;

    # Generating the 3 sides randomly
    a := RG(integer(range = 1..10));
    b := RG(integer(range = 1..10));
    c := RG(integer(range = 1..10));

    # Calculating the area
    s := (1/2)*(a+b+c);
    ans := eval( sqrt(s*(s-a)*(s-b)*(s-c)) );

    # Avoiding non-existing triangles:
    if evalb( whattype(evalf(ans)) = complex(extended_numeric)) then
      Actions:-SetProblem();
      # If we get a complex area, we call the SetProblem procedure again.
      # However, make sure that the problem is solvable to avoid infinite
          recursion.
      return;
    end if;

    # Displaying the side lengths to the user using the Document Tools command
        : DocumentTools:-SetProperty, defined by macro as "SP".
    # SetProperty changes a property of a component to the wanted value:
    # SP('Component name', 'property to change (often value)', value to change
         to);
    SP('Given1','value',a);                         ###### COMPONENTS
    SP('Given2','value',b);            ###### COMPONENTS
    SP('Given3','value',c);            ###### COMPONENTS

    # Storing the solution as a list in a variable SOL,
```

```
  #   declared local to the Actions module.
  SOL := [ans];

  ############### END EDIT ############################
  #### IF NEEDED, CHANGE THE GRADING IN InputCheck ####
  #####################################################
end proc: # end SetProblem

#### Checking if the input is of the correct type
InputCheck:= proc(raw::string, sol::list)
  local unpack, num, numsol, listnum, listnumsol,
  userflat, solflat, soldiff, solref, GO1, GO2, names,
  differ, i, solflatfct, ran, q, w;
  global  grad, report, interpretation;

  GO1:= 1;

  try
    parse(raw);
  catch :
    GO1:= 0;
  end try;

  if evalb(GO1=1) then
    report:= "";
    unpack:= [eval(parse(raw))];
    num:= nops(Flatten(unpack, 1));
    numsol:= nops(Flatten([sol], 1));
    listnum:= nops(Flatten(unpack));
    listnumsol:= nops(Flatten([sol]));

    if not(num=numsol and listnum=listnumsol and indets(evalf(unpack), name)
        = indets(evalf(sol), name)) then
      GO2:= 0;
      report:= "Wrong type or missing input";
      interpretation:= "";
    else
      GO2:= 1; report:= "";
      if (num=3 and listnum=9) or (num=2 and listnum=4) then
        interpretation:= Matrix(Flatten(unpack, 1));
      else
        interpretation:= Flatten(unpack, 1);
      end if;

      if (num=1 and listnum=1) then
        interpretation:= unpack[1];
      end if;
    end if;
  else
    report:= "Syntax error"; interpretation:= "";
  end if;

  if evalb(GO1=1 and GO2=1) then
    userflat:= evalf(Flatten(unpack));
    solflat:= evalf(Flatten(sol));
    names:= indets(evalf(unpack), name);

    if nops(names) > 0 then
      differ:= unapply((solflat - userflat), seq(names[i], i=1...nops(names)
        ));
      solflatfct:= unapply(solflat, seq(names[i], i=1...nops(names)));
      ran:= rand(-10...10);
```

```


        # Here we determine the maximum difference in value when evaluating
           the functions in 20 randomly generated points.
        soldiff:= max(seq(evalf[3](Norm(Vector(differ(seq(ran(), w=1...nops(
           names)))),2)), q=1...20));
        solref:= max(seq(evalf[3](Norm(Vector(solflatfct(seq(ran(), w=1...nops
           (names)))),2)), q=1...20));

      else
                     if solflat=[] and userflat=[] then
                       soldiff:= 0.0;
                       solref:= 1.0;
                     else
                       soldiff:= evalf[3](abs((solflat - userflat)[1]));
                       solref:= evalf[3](abs(solflat[1]));
                     end if
      end if;


      ###################################################################
      ##### CHANGE THE FOLLOWING IF DIFFERENT GRADING IS WANTED #########
      ###################################################################

      if soldiff <= 0.001*solref then
        grad := 1.0;
      elif soldiff <= 0.1*solref then
        grad:= 0.5;
      else
        grad:= 0.0;
      end if;

      ###################################################################
      ######################### END EDIT ################################
      ###################################################################
    else
      grad:= 0.0
    end if;

  end proc: # end InputCheck( userInput, SOL)

  ##### Initialization
  Initialize := proc()
    # Setting the grade to zero initially
    Grad := 0;

      # clearing the boxes
    SP('Input','value',NULL);             ###### COMPONENTS
    SP('GradeDisplay', 'value', NULL);        ###### COMPONENTS
    SP('Reporter', 'value', NULL);        ###### COMPONENTS
    SP('Interpreter', 'value', NULL);       ###### COMPONENTS

    # setting up the problem according to the previous procedure
    Actions:-SetProblem();

  end proc: # end Initialize()

  GetInput := proc()
    local userInput;

    # Getting the input from the user
    userInput := GP('Input','value');         ###### COMPONENTS
```

```
    # Checking the input with the known solution
    Actions:-InputCheck(userInput, SOL);

    # Defining the grade
    Grad := grad;
    # Displaying the grade
    SP('GradeDisplay','value', Grad);       ###### COMPONENTS

    # Displaying the interpretation if possible
    if (report = "") then
      SP('Interpreter','value', interpretation);  ###### COMPONENTS
    else
      SP('Interpreter','value',NULL);     ###### COMPONENTS
    end if;

    # Displaying the report, i.e. comments on the input type.
    SP('Reporter','value',report);        ###### COMPONENTS

  end proc; # end GetInput()

  ########################################
  # GRADE OUT
  ########################################
  Grade := proc()

      return Grad;

  end proc; # end Grade()

end module:

Actions:-Initialize();
```

A.1.2 Template 2

Worksheet

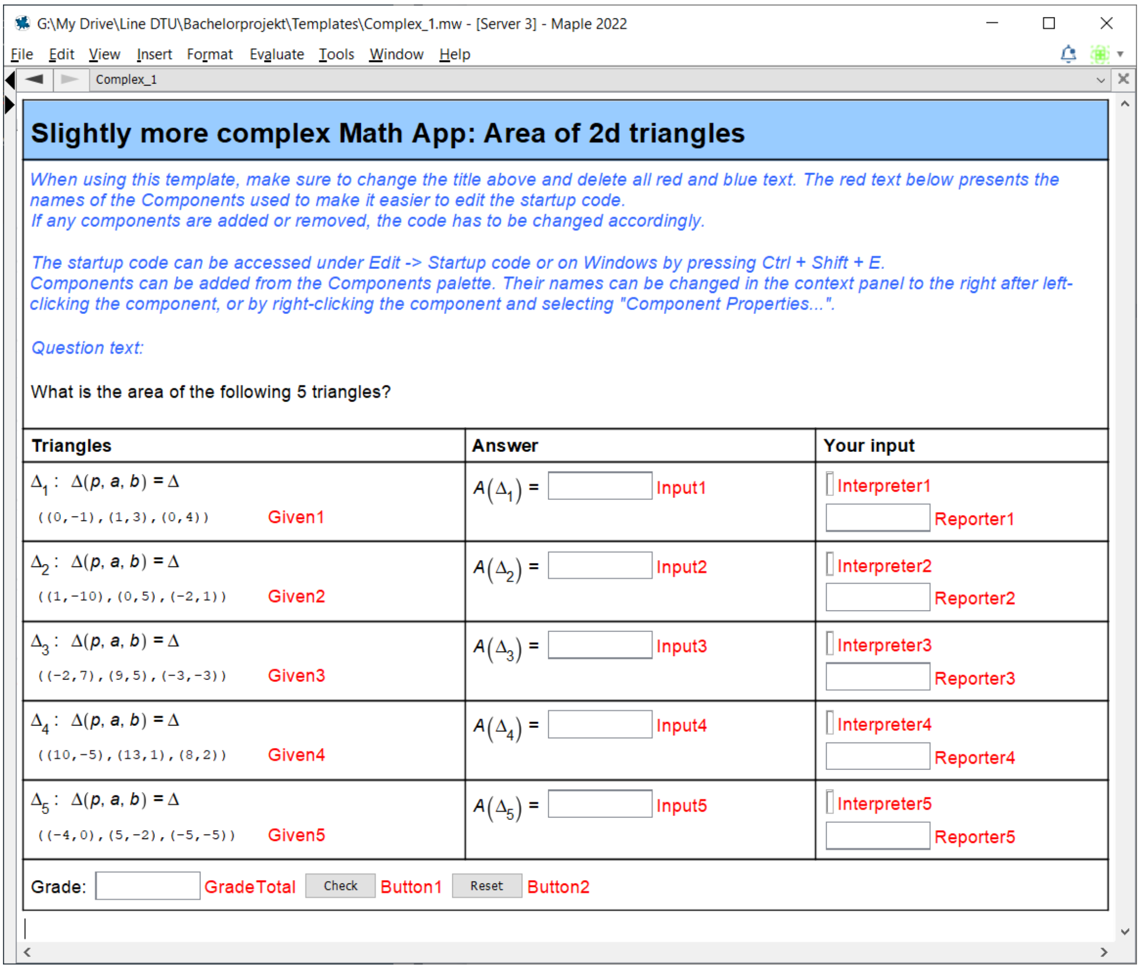


Figure A.2: The worksheet for Math App template 2

Startup code

```
#################################################################
### Maple Math App Template: Complex 1
### - Line Glade, DTU 2022
################################################################
###
### In the code look for lines similar to:
###
###    #####################################################
###    ###  TEXT HERE, SHORT DESCRIPTION OF CHANGE       ###
###    #####################################################
###
### If any components are added or removed, pay attention to
### all markings of components, like "###### COMPONENTS".
### Remember to remove or add the relevant components in the worksheet.
###
#################################################################
restart;

########## INCLUDE RELEVANT PACKAGES ############
with(RandomTools): with(combinat): with(ListTools):
with(LinearAlgebra):with(plots): with(plottools):
```

```
########## Creating practical macros ###########
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()
  export Initialize, GetInput, SetProblem, Grade, ShowInput, InputCheck;
  local SOL;
  global Grad, GradTotal, grad, report, intepretation;

  #### Setting up the problem
  SetProblem := proc()
    randomize();

    ####################################################
    ###  CHANGE THE FOLLOWING TO FIT YOUR QUESTION    ###
    ####################################################

    # add or remove variables according to the question
    local ans, T, i, j, F, p, a, b;

    # Making random triangles
    # - note it is very unlikely for the same triangle to appear
    for i from 1 to 5 do
      p := [RG(integer(range = -5.....10)), RG(integer(range = -10.....10))];
      a := [RG(integer(range = -10.....15)), RG(integer(range = -5.....5))];
      b := [RG(integer(range = -8.....12)), RG(integer(range = -10.....5))];

      T[i] := [p,a,b];

      # Calculating the answers
      SOL[i] := [(1/2)*abs(Determinant(<T[i][2][1],T[i][2][2]| T[i][3][1], T[i
          ][3][2]>))];
    end do;

    # Writing the formulas for the triangles:
    for i from 1 to 5 do
      F := "(";
      for j from 1 to 3 do
        F := cat(F,"(", T[i][j][1], ",", T[i][j][2], ")");
        if not evalb(j = 3) then
          F := cat(F, ",");
        end if;
      end do;
      F := cat(F, ")");
      SP(cat('Given',i),'value',F);
    end do;


    ############### END EDIT ###########################
    #### IF NEEDED, CHANGE THE GRADING IN InputCheck ####
    ####################################################
  end proc: # end SetProblem

  #### Checking if the input is of the correct type
  InputCheck:= proc(raw::string, sol::list)
    local unpack, num, numsol, listnum, listnumsol, user,
    userflat, solflat, soldiff, solref, GO1, GO2, names,
```

```
  differ, i, solflatfct, ran, q, w;
  global  grad, report, interpretation;

  GO1:= 1;

  try
    parse(raw);
  catch :
    GO1:= 0;
  end try;

  if evalb(GO1=1) then
    report:= "";
    unpack:= [eval(parse(raw))];
    num:= nops(Flatten(unpack, 1));
    numsol:= nops(Flatten([sol], 1));
    listnum:= nops(Flatten(unpack));
    listnumsol:= nops(Flatten([sol]));

    if not(num=numsol and listnum=listnumsol and indets(evalf(unpack), name) = indets(evalf(sol), name)) then
      GO2:= 0;
      report:= "Wrong type or missing input";
      interpretation:= "";
    else
      GO2:= 1; report:= "";
      if (num=3 and listnum=9) or (num=2 and listnum=4) then
        interpretation:= Matrix(Flatten(unpack, 1));
      else
        interpretation:= Flatten(unpack, 1);
      end if;

      if (num=1 and listnum=1) then
        interpretation:= unpack[1];
      end if;
    end if;
  else
    report:= "Syntax error"; interpretation:= "";
  end if;

  if evalb(GO1=1 and GO2=1) then
    user:= evalf(Flatten(unpack, 1));
    userflat:= evalf(Flatten(unpack));
    solflat:= evalf(Flatten(sol));
    names:= indets(evalf(unpack), name);

    if nops(names) > 0 then
      differ:= unapply((solflat - userflat), seq(names[i], i=1...nops(names)));
      solflatfct:= unapply(solflat, seq(names[i], i=1...nops(names)));
      ran:= rand(-10...10);


      # Here we determine the maximum difference in value when evaluating the functions in 20 randomly generated points.
      soldiff:= max(seq(evalf[3](Norm(Vector(differ(seq(ran(), w=1...nops(names)))),2)), q=1...20));
      solref:= max(seq(evalf[3](Norm(Vector(solflatfct(seq(ran(), w=1...nops(names)))),2)), q=1...20));

    else
                if solflat=[] and userflat=[] then
```

```
                        soldiff:= 0.0;
                        solref:= 1.0;
                      else
                        soldiff:= evalf[3](abs((solflat - userflat)[1]));
                        solref:= evalf[3](abs(solflat[1]));
                      end if
        end if;


        ###################################################################
        ##### CHANGE THE FOLLOWING IF DIFFERENT GRADING IS WANTED #########
        ###################################################################

        if soldiff <= 0.001*solref then
          grad := 1.0;
        elif soldiff <= 0.1*solref then
          grad:= 0.5;
        else
          grad:= 0.0;
        end if;
      else
        grad:= 0.0
      end if;

  end proc: # end InputCheck

  ##### Initialization
  Initialize := proc()
    local i;
    # Setting the grade to zero initially
    GradTotal := 0;

    # clearing the boxes
    for i from 1 to 5 do
      Grad[i] := 0;
      SOL[i] := 0;
      SP(cat('Input',i),'value',NULL);       ###### COMPONENTS
      SP(cat('Reporter',i), 'value', NULL);    ###### COMPONENTS
      SP(cat('Interpreter',i), 'value', NULL);  ###### COMPONENTS
    end do;

    SP('GradeTotal','value',NULL); # clearing the grade text area

    # setting up the problem according to the previous procedure
    Actions:-SetProblem();

  end proc: # end Initialize()

  GetInput := proc( n ) # Retrives the input for one box at a time
    local userInput;

    # Getting the input from the user
    userInput := GP(cat('Input',n),'value');       ###### COMPONENTS
    # Checking the input with the known solution
    Actions:-InputCheck(userInput, SOL[n]);

    # Defining the grade
    Grad[n] := grad;

    # Updating the grade:
    GradTotal := evalf[3]((Grad[1] + Grad[2] + Grad[3] + Grad[4] + Grad[5])/5)
        ;
```

```
        SP('GradeTotal','value',GradTotal);

        # Displaying the interpretation if possible
        if (report = "") then
          SP(cat('Interpreter',n),'value',interpretation);  ###### COMPONENTS
        else
          SP(cat('Interpreter',n),'value',NULL);      ###### COMPONENTS
        end if;

        # Displaying the report, i.e. comments on the input type.
        SP(cat('Reporter',n),'value',report);       ###### COMPONENTS

    end proc: # end GetInput()

    ########################################
    # GRADE OUT
    ########################################
    Grade := proc();

        return GradTotal;

    end proc: # end Grade()

end module:

Actions:-Initialize();
```

A.1.3 Template 3

Worksheet

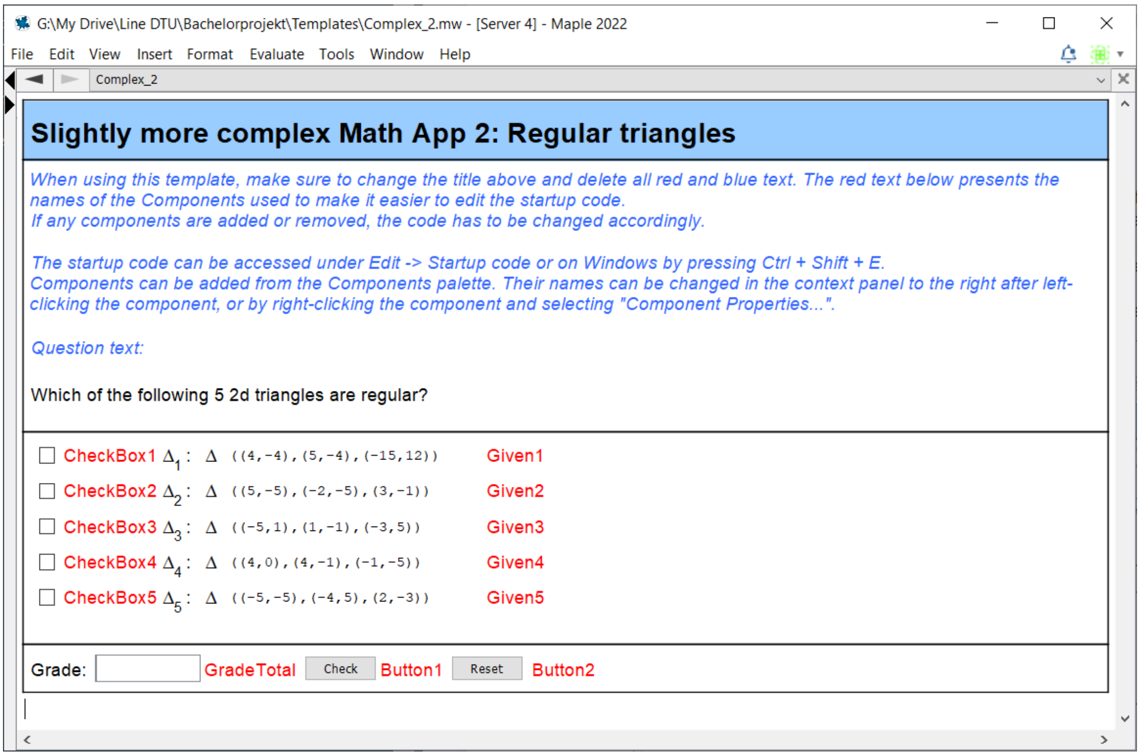


Figure A.3: The worksheet for Math App template 3

Startup code

```
################################################################
### Maple Math App Template: Complex 2
### - Line Glade, DTU 2022
###############################################################
###
### In the code look for lines similar to:
###
###    #####################################################
###    ###  TEXT HERE, SHORT DESCRIPTION OF CHANGE       ###
###    #####################################################
###
### If any components are added or removed, pay attention to
### all markings of components, like "###### COMPONENTS".
### Remember to remove or add the relevant components in the worksheet.
###
### Note that we for this Math App do not use the InputCheck() procedure,
### since we have check boxes instead of text areas for user input.
################################################################
restart;

########## INCLUDE RELEVANT PACKAGES ############
with(RandomTools): with(combinat): with(ListTools):with(LinearAlgebra):with(plots): with(plottools):

######### Creating practical macros ###########
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()
  export Initialize, GetInput, SetProblem, Grade;
  local SOL;
  global Grad;

  #### Setting up the problem
  SetProblem := proc()
    # add or remove variables according to the question
    local ans, T, p, a, b, r, i, j, F, ar;

    randomize();

    #####################################################
    ###  CHANGE THE FOLLOWING TO FIT YOUR QUESTION    ###
    #####################################################

    ###  Making random triangles
      # Most of the triangles will be regular
      # We want to make sure at least one triangle is non-regular
      r := RG(integer(range=1..5)); # the triangle we will force to be non-regular

      for i from 1 to 5 do
        p := [RG(integer(range = -5...5)), RG(integer(range = -5...5))];
        a := [RG(integer(range = -5...5)), RG(integer(range = -5...5))];
        if evalb(i = r) then
          b := RG(integer(range=-10...-1))*a;
        else
          b := [RG(integer(range = -3.....5)), RG(integer(range = -5.....5))];
```

```
      end if;

      T[i] := [p,a,b]; # storing the vectors of the triangle

      # Calculating the areas
      ar[i] := evalf((1/2)*abs(Determinant(<T[i][2][1],T[i][2][2]| T[i
        ][3][1], T[i][3][2]>)));
      if not evalb(evalf(ar[i]) = 0) then
        ans[i] := true;
      else
        ans[i] := false;
      end if;
    end do;

  ##### Writing the triangles on the wanted form into the respective "Given"
     TextAreas
    for i from 1 to 5 do
      F := "(";
      for j from 1 to 3 do
        F := cat(F,"(", T[i][j][1], ",", T[i][j][2], ")");
        if not evalb(j = 3) then
          F := cat(F, ",");
        end if;
      end do;
      F := cat(F, ")");
      SP(cat('Given',i),'value',F);
    end do;

  # Storing the solution
  SOL := [ans[1], ans[2], ans[3], ans[4], ans[5]];
  ############### END EDIT ###########################
end proc: # end SetProblem


##### Initialization
Initialize := proc()
  local i;
  # Setting the grade to zero initially
  Grad := 0;

  # Clearing the checkboxes
  for i from 1 to 5 do
    SP(cat('CheckBox',i),'value', false);
  end do;

  # Clearing the grade area
  SP('GradeTotal','value',NULL);

  # setting up the problem according to the previous procedure
  Actions:-SetProblem();

end proc: # end Initialize()

GetInput := proc( )
  local userInput, u_inc, u_cor, grad, i, sol;

  grad := 0: u_inc := 0: u_cor := 0;
  # We use u_inc and u_cor to denote the number of correct and incorrect
     choices
  # this is used for partial grading

  for i from 1 to 5 do
```

```
      userInput := parse(GP(cat('CheckBox',i), 'value')); # checking if a box
          is checked
      sol := SOL[i];
      if evalb(userInput = sol) then
        u_cor := u_cor + 1;
      else
        u_inc := u_inc + 1;
      end if;
    end do;

    # Determining the grade
    grad := evalf[3]((u_cor)/ (u_cor + u_inc));

    Grad := grad;
    SP('GradeTotal','value',Grad);

  end proc: # end GetInput()

  ########################################
  # GRADE OUT
  ########################################
  Grade := proc();

      return Grad;

  end proc: # end Grade()

end module:

Actions:-Initialize();
```

A.1.4 Template 4

Worksheet

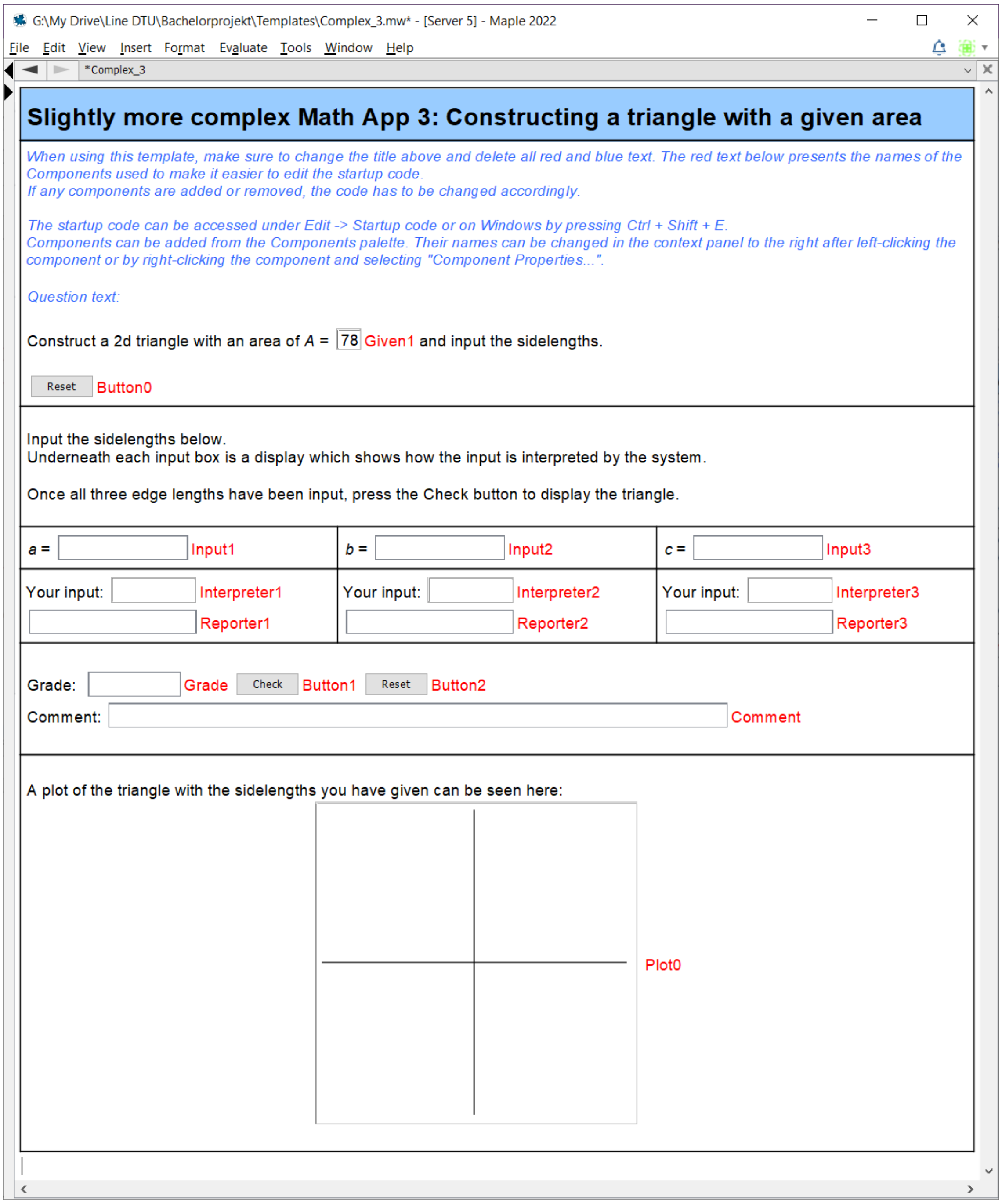


Figure A.4: The worksheet for Math App template 4

Startup code

```
###############################################################
### Maple Math App Template: Complex 3
### - Line Glade, DTU 2022
##############################################################
###
### In the code look for lines similar to:
###
```

```
###   ####################################################
###   ###  TEXT HERE, SHORT DESCRIPTION OF CHANGE       ###
###   ####################################################
###
### If any components are added or removed, pay attention to
### all markings of components, like "###### COMPONENTS".
### Remember to remove or add the relevant components in the worksheet.
###
### Note that the nature of this Math App is also slightly different,
### since we have multiple correct answers.
### The InputCheck procedure has been changed to now only check if
### the user input evaluates to a float.
##############################################################
restart;

########## INCLUDE RELEVANT PACKAGES ############
with(RandomTools): with(plots): with(plottools):

######### Creating practical macros ###########
macro(
SP = DocumentTools:-SetProperty,
GP = DocumentTools:-GetProperty,
RG = RandomTools:-Generate
):

#### Beginning the actions module ####
Actions := module()
  export Initialize, Plot, GetInput, ShowInput, SetProblem, Grade, InputCheck;
  local triang;
  global Grad, SOL, report, interpreter, a, b, c, G0;

  ##### Initialization
  Initialize := proc()
    local i;
    # Setting the grade to zero initially
    Grad := 0.0;
    G0 := 0;

      # clearing the boxes
    for i from 1 to 3 do
      SP(cat('Input',i),'value',NULL);          ###### COMPONENTS
      SP(cat('Interpreter',i),'value',NULL);       ###### COMPONENTS
      SP(cat('Reporter',i),'value',NULL);       ###### COMPONENTS
    end do;

    SP('Grade','value',NULL);            ###### COMPONENTS
    SP('Comment','value',NULL);            ###### COMPONENTS
      SP('Plot0','value',NULL);            ###### COMPONENTS

    # setting up the problem according to the previous procedure
    Actions:-SetProblem();
    Actions:-Plot();

  end proc: # end Initialize()

  #### Setting up the problem
  SetProblem := proc()
    randomize();

    #####################################################
    ###  CHANGE THE FOLLOWING TO FIT YOUR QUESTION    ###
    #####################################################
```

```

  # Generating a random area
  SOL := RG(integer(range = 1..100));
  SP('Given1','value', SOL);

  ############### END EDIT ############################
  #### FURTHER CHANGES IN PLOT PROCEDURE            ####
  ####################################################

end proc: # end SetProblem

#### Checking if the input is of the correct type
InputCheck:= proc( raw::string )
  local GO1, userinput;
  global report, interpreter;

  GO1:= 1;

  if (raw = "") then
    GO1 := 0;
    report := "Missing input";
    interpreter := "";
  else

    try
      parse(raw);
    catch :
      GO1:= 0;
    end try;

    if evalb(GO1 = 1) then
      userinput := parse(raw);

      # Checking if the input can be evaluated to a float
      if evalb( whattype(evalf(userinput)) = float) then
        report := "";
        interpreter := userinput;
      else
        report := "Wrong type of input";
        interpreter := "";
      end if;
    else
      report := "Syntax error";
      interpreter := "";
    end if;
  end if;

end proc: # end InputCheck

########################################
# Creating the plot
########################################
Plot := proc()

  ####################################################
  ###  CHANGE THE FOLLOWING TO FIT YOUR QUESTION    ###
  ####################################################

  local LL, L, Sgam, Cgam, Sbeta, Cbeta, triangFill, a,b,c, s, Area;
  global GO;

  if evalb(GO = 1) then
```

```
    a := evalf(parse(GP('Input1','value')));
    b := evalf(parse(GP('Input2','value')));
    c := evalf(parse(GP('Input3','value')));

    s := (1/2)*(a+b+c);
    Area := simplify(sqrt(s*(s-a)*(s-b)*(s-c)));

    # To make the plot we scale the triangle such that the longest edge
    #   becomes the baseline with a length of 1.
    LL:= sort([a,b,c], `>`); # Sorting the edge lengths
    L:= [1, evalf(LL[2]/LL[1]), evalf(LL[3]/LL[1])]; # Scaled edge lengths

    Sgam:= 2*Area/(LL[2]*LL[1]);
    Cgam:= (LL[1]^2 + LL[2]^2 - LL[3]^2)/(2*LL[1]*LL[2]);
    Sbeta:= 2*Area/(LL[3]*LL[1]);
    Cbeta:= (LL[1]^2 + LL[3]^2 - LL[2]^2)/(2*LL[1]*LL[3]);
    triang:= (
      line([0,0], [1,0], thickness=3, color=blue),
      line([0,0], L[2]*[Cgam, Sgam], thickness=3, color=blue),
      line([1,0], [1,0]+L[3]*[-Cbeta, Sbeta], thickness=3, color=blue),
      disk([0,0], 0.05, color=red),
      disk([1,0], 0.05, color=red),
      disk([1,0]+L[3]*[-Cbeta, Sbeta], 0.05, color=red)
    );
    triangFill:= polygon([[0,0], [1,0], [1,0]+L[3]*[-Cbeta, Sbeta]], color=cyan);
    SP('Plot0', 'value', plots:-display(triang, triangFill, axes=none, scaling=constrained));
    #SP('Plot0','value',plots:-display(plot(x^2)));
  else
    # Draw empty plot
    SP('Plot0','value', plots:-display(plots:-pointplot([]), tickmarks=[0,0]));
  end if;

  #####################################################
  #################### END EDIT #######################
  #####################################################

end proc:

GetInput := proc()
  local aRaw, bRaw, cRaw, s, area, soldiff, solref;
  GO := 0;
  # Since we have multiple correct answers we cannot compare the input directly to a solution.
  # Instead we compute the area of the triangle with the edge lengths as input, and compare this to the wanted area.

  # Getting the input from the user
  aRaw := GP('Input1','value');         ###### COMPONENTS
  bRaw := GP('Input2','value');         ###### COMPONENTS
  cRaw := GP('Input3','value');         ###### COMPONENTS

  Actions:-InputCheck( aRaw);
  SP('Reporter1','value',report);
  SP('Interpreter1','value',interpreter);
  if (report = "") then
    a := parse(aRaw);
    GO := 1;
  else
    GO := 0;
```

```
  end if;

  Actions:-InputCheck( bRaw);
  SP('Reporter2','value',report);
  SP('Interpreter2','value',interpreter);
  if (report = "") then
    b := parse(bRaw);
    GO := 1;
  else
    GO := 0;
  end if;

  Actions:-InputCheck( cRaw);
  SP('Reporter3','value',report);
  SP('Interpreter3','value',interpreter);
  if (report = "") then
    c := parse(cRaw);
    GO := 1;
  else
    GO := 0;
  end if;

  # If GO = 1 then all three edge lengths are floats.
  # However, we have some additional tests such as a, b and c being positive
  # and s > area.
  if (GO = 1) then
    if evalb( evalf( min(a,b,c)) <= 0) then
      Grad := 0.0;
      SP('Comment','value',"All edge lengths must be positive");
      GO := 0;
    else
      s := (a+b+c)/2;

      if ( evalf(s) > evalf(max(a,b,c)) ) then
        area := eval(sqrt(s*(s-a)*(s-b)*(s-c)));
        SP('Comment','value',cat("The area of the triangle is ", evalf[5](area)));

        # comparing the area to the desired value
        soldiff := abs(evalf(area)-evalf(SOL));
        solref := evalf(SOL);

        # Partial points if the area is close to the desired value
        if evalb( soldiff < 0.01*solref ) then
          Grad := 1;
        elif evalb( soldiff < 0.1*solref) then
          Grad := 0.5;
        else
          Grad := 0.0;
        end if;
      else
        Grad := 0.0;
        SP('Comment','value',"The input edge lengths does not make a non-degenerate triangle");
        GO := 0;
      end if;
    end if;
  else
    Grad := 0.0;
    SP('Comment','value',"Please insert real numbers.");
  end if;

```

```
    # Displaying the grade
    SP('Grade','value',Grad);

    Actions:-Plot();

  end proc: # end GetInput()


  ########################################
  # GRADE OUT
  ########################################
  Grade := proc();

      return Grad;

  end proc: # end Grade()

end module:

Actions:-Initialize();
```