Scientific Calculator Plus Manual and M F P Programming Language Tutorials

For Scientific Calculator Plus v. 2.1.0.92 or later

Scientific Calculator Plus Manual & MFP Programming Language Tutorials

CYZSoft
NSW, Australia
Tel: +61-4-25388821

Contents

h
h

Preface

Scientific Calculator Plus is a powerful tool to do mathematical analysis. Besides the functionality supported by most of other calculator apps, it includes a lot of novel capabilities like printed math expression recognition, solving mathematical equation(s), calculus and (3D) chart plotting. In particular, the most important feature is its programmability. In essence, Scientific Calculator Plus is a wrapper of a simple but capable programming language named MFP. This language has similar keywords and syntax to Basic, while it possesses built-in supports to complex number, array (matrix), string, file operations, time and date. More appealing, each function of MFP can be compiled into an Android app and the created APK file can be published in Google Play or any other Android app distribution web-site.

As the developer of Scientific Calculator Plus, I was always asked why I created a calculator instead of a more popular web based app. Indeed, an Android user may be dazzled by the huge list of calculator apps available in Google Play. On the other hand, most of the calculators are quite similar to each other. They basically are copies of hardware-based calculators sold in the market with some capabilities to plot graphs and/or solve math equations. But unfortunately many needs of engineers, science and technology students and advanced Android users are not addressed at all.

Advanced requirements come from various scenarios where only small and portable devices can be selected to analyze and solve complex mathematical problems. In general, these problems include a number of inputs and the logic cannot be simplified to a single math expression such that programming is inevitable. Several approaches can be selected to tackle them. First is using a hardware based programmable calculator, e.g. TI or Casio, or their Android emulator(s). However, these calculators are never cheap and notoriously hard to operate. Moreover, so far no easy solutions have been provided to share programs inside a distributed work team. Users have to upload and download source files and store them in the right place which is a major obstacle to beginners.

Second way is developing apps specifically for each of the problem. Although occasionally some companies and individuals do invest resources to do this work, considering the programming knowledge, skill and experience requirements and the vast number of to-solve problems, expense prohibits any wide application of this solution.

The last solution resorts to a scripting language, e.g. Python, Matlab and Pearl. This approach is viable if a laptop is available. Otherwise, it is as impractical as using a TI emulator since finding a script file in Android (or any other mobile OS) file system is almost an impossible mission to most of mobile users.

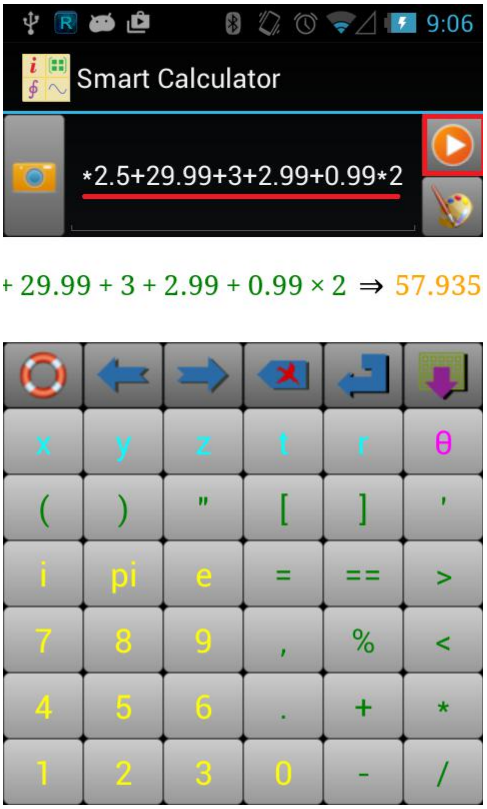
I am a quantitative developer and my every day duty involves intensive mathematical analysis, graph plotting and programming. Sometimes out of hours, I am required to deliver calculation results or update some numeric figures. Without a computer nearby, a multi-functional calculator is in desperate need. This calculator has to be programmable, and support 3D graphing, and every result has to be stored and reproduced later on, whether in a computer or in some other devices. As no appropriate software or hardware found after searching the market, I finally decided to create a calculator app by myself.

The initial step of my long development journey was selecting a platform. As a free and open-source embedded/mobile operating system, Android can be and is being adapted to every industry as well as people's life. From medical equipment to wireless sensor, and from office automation to remote control, this little green robot emerges everywhere and demonstrates enormous influence. In about five years, we will see a great chance that Android turns into the industry standard of embedded operating system.

Another benefit to pick Android comes from its JAVA compatibility. Android is a JAVA-like virtual machine on top of Linux. Although the byte codes are distinct, Android APIs are extended from JAVA 6 with no variations. This gives rise to the idea that program once, run everywhere. This feature is particularly significant to me because any result or chart generated by a calculator should be reproducible in a computer. In fact, Scientific Calculator Plus for Android includes a JAVA based component which can execute any MFP program in any PC with JAVA installed.

The first release of Scientific Calculator Plus which supported matrix, complex number, 2D graphing and programming was introduced in the Google Market in 2012. Since then, I contribute all my spare time to add fresh and exciting functionality and performance improvements in every upgrading. Version 1.0.4 was able to perform matrix division; version 1.1 included Scientific Calculator Plus for JAVA; version 1.2 allowed to define user specified input pad; version 1.3.1 can locate roots of polynomials; version 1.4 was capable of solving math equations; version 1.5 supported 3D graphing; version 1.6.2 strengthened support to integration; version 1.6.3 included a set of file operation functions; version 1.6.4 started to recognize printed math formulas; version 1.6.6 was able to build independent Android app from an MFP function; version 1.7 introduced citingspace and started to support derivative calculation; from version 1.7.1 MFP is able to start from command line or self-executable with a shebang declaration, like Python or Perl; in version 1.7.2 developer can build cross-platform games using MFP; version 1.8.0 started to supported parallel computing; and since version 2.0.0 object oriented programming is supported. Because of my efforts, this application evolves so rapidly that now it is able to cope with any hardware based programmable calculators and their emulators.

Scientific Calculator Plus can assist user in the following aspects. First of all, this app includes a component called Smart Calculator. Its interface is close to a traditional calculator, such that user can use it to perform simple calculation in shopping, trading, money management, etc. For example, in a supermart, chicken breast is priced at $7.99, honey is $29.99 a bottle, bread is $3 a loaf, 2 kg potato bag is $2.99 and the price of carrot is $0.99/kg. In order to calculate the total expense to purchase 2.5 kg chicken, one bottle of honey, one loaf of bread, one bag of potato and 2 kg carrots, user simply inputs 7.99*2.5+29.99+3+2.99+0.99*2, and taps the start button (orange circle with a white triangle inside), then the final result 57.935 will come out, as shown in the following chart.



Smart Calculator

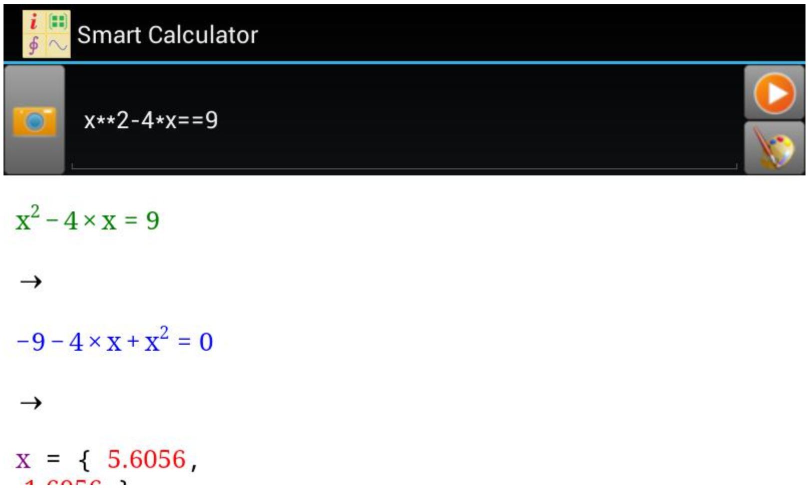
*2.5+29.99+3+2.99+0.99*2

+ 29.99 + 3 + 2.99 + 0.99 × 2 ⇒ 57.935

Figure 0.1: Use Smart Calculator (a component of Scientific Calculator Plus) to do basic calculations in shopping.

Second, Scientific Calculator Plus is a good tool for mathematics education. Scientific Calculator Plus can solve mathematical equations. As demonstrated in Figure 0.2: , user inputs x**2-4*x==9 in Smart Calculator and taps the start button, then the roots for equation will be listed. Note that in Scientific Calculator Plus, power operator is ** instead of ^, and equality is == instead of =.

In order to arouse students' interest in math study, Scientific Calculator Plus provides some funny examples. For instance, after the app is started, user can tap Chart Plotter, select Plot polar chart, and then click View button, heart, flower and circle, which are plotted curves of three mathematical equations, will be shown.



```
x**2-4*x==9
```

$x^2 - 4 \times x = 9$

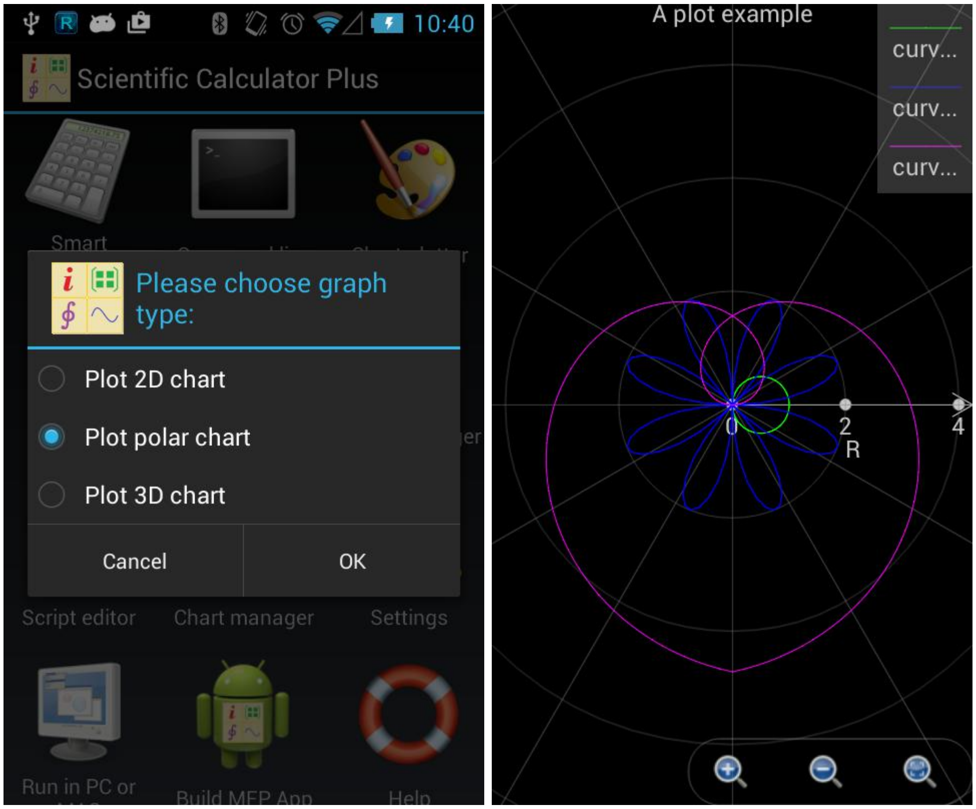
→

$-9 - 4 \times x + x^2 = 0$

→

```
x = { 5.6056,
-1.6056 }
```

Figure 0.2: Scientific Calculator Plus can solve equations.



Please choose graph type:

○ Plot 2D chart

● Plot polar chart

○ Plot 3D chart

Cancel OK

Figure 0.3: Use Scientific Calculator Plus to draw funny curves.

Third, Scientific Calculator Plus can be used in financial engineering. The foundation of this field is maths. As pointed out before, work of every participant in this industry involves substantial calculation. Furthermore, analysts are generally required to create charts to present calculation results in a straightforward way. A good example is volatility surface. Although traditional PC software applications, e.g. Matlab and Excel, can draw 3D charts, their charts are static. Rotating, shifting and zooming the surface are necessary if user wants to see the whole volatility surface from different perspectives, or wants to analyze the details of volatility smile. Scientific Calculator Plus offers these capabilities not only in Android mobiles, but in any computers with JAVA installed, as such it is a qualified candidate to undertake any financial analysis task.

The following two charts generated in a PC and an Android device respectively demonstrate the same volatility surface seen at different observation points.

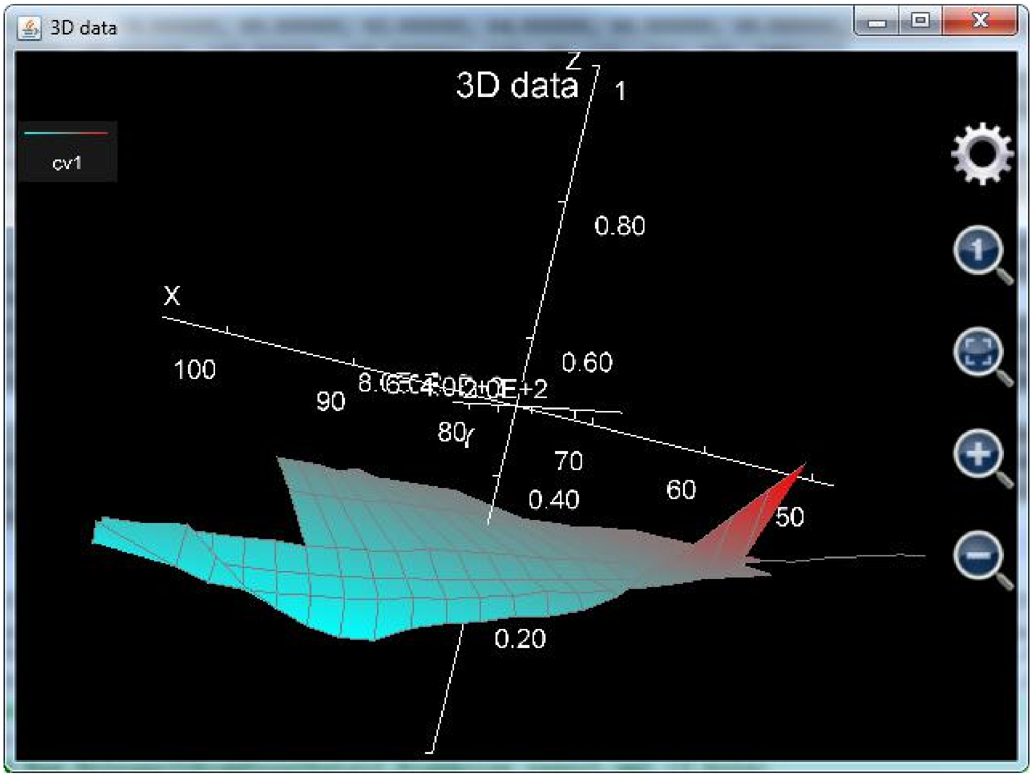


Figure 0.4: Scientific Calculator Plus plots volatility surface in a PC.

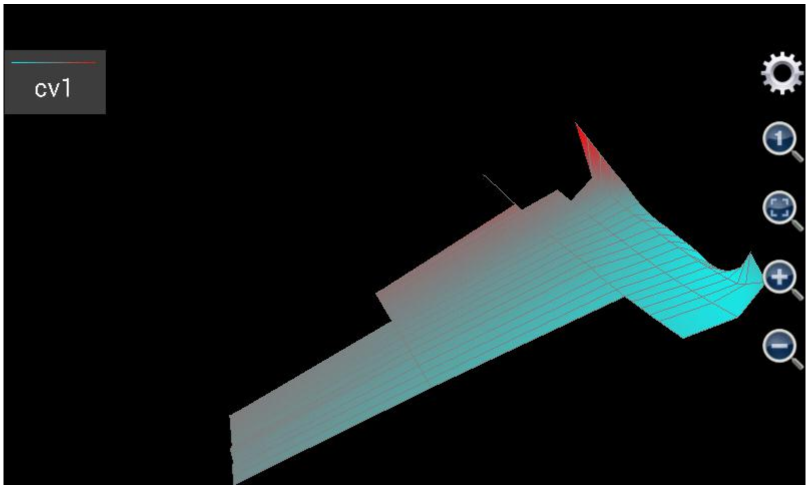
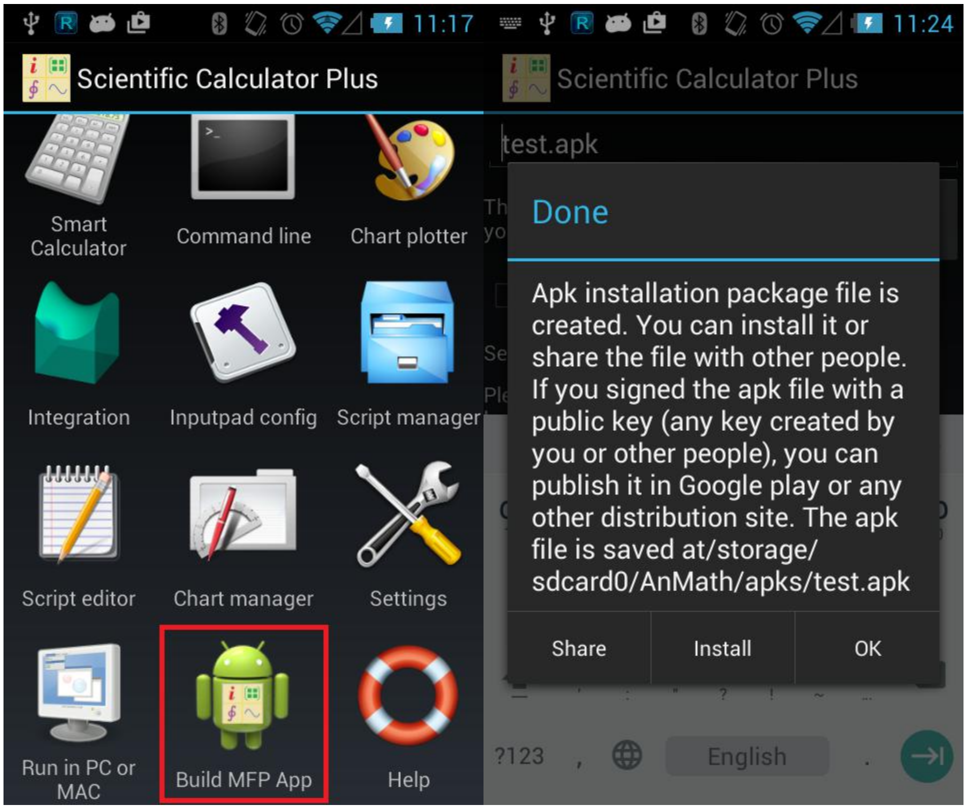


Figure 0.5: Scientific Calculator Plus plots volatility surface in an Android phone from a different perspective (from Figure 0.4). The coordinate axes are hidden.

Fourth, Scientific Calculator Plus is the best choice for on-site calculation in industry. During construction surveying, equipment commissioning and performance test, small computing scripts are usually in need to perform some analysis too complicated to do manually, but not big enough to particularly build a software application for it. In an outdoor environment lack of power source, computer, even a laptop, is far too heavy and inflexible. Rather, people prefer a light, palm size and durable device. Therefore, taking advantage of the programming capability of Scientific Calculator Plus is the most feasible decision. Further, as a bonus, Scientific Calculator Plus is able to generate Android app from any MFP function. As such, engineer can easily pack his/her programs and share with colleagues and/or any other potential users all over the world via email, Bluetooth, NFC, Google Play or even Facebook. As a result, duplicate development work is avoided and convenience is propagated.



Apk installation package file is created. You can install it or share the file with other people. If you signed the apk file with a public key (any key created by you or other people), you can publish it in Google play or any other distribution site. The apk file is saved at/storage/sdcard0/AnMath/apks/test.apk

Figure 0.6: Scientific Calculator Plus can generate Android app from any MFP function and allows user to install and share the app.

Fifth, Scientific Calculator Plus allows users with programming background to develop games. Nowadays, games, whether running in PC or in mobile, are generally implemented by compilable languages like C/C++, C# or JAVA. These languages are fast and are needed by large-scale video games which have high realtime requirements. However, these languages need specific SDK and IDE, and generally are not cross-platform (or have to be recompiled to transplant). Some simple games can be developed by html or JAVA script. Their cross-platform capability comes from Internet browser. However, they are very slow and difficult to access local resources so that their use is limited. Scripting languages, on the other hand, are best candidates for games without high real-time requirements but not that simple.

Many scripting languages, e.g. Python and Lua, can develop games. However, they are not cross-platform. For example, Python based PyGame module can run in PC, but not in Android. And LuaJAVA can call both Android and traditional JAVA's APIs. However, the code for Android is different from for traditional JAVA so that LuaJAVA's game cannot run in both platforms without modification.

Program once, run everywhere is important. In the next ten years, not everyone will have a PC, but everyone will have at least one Android device. However, Android is designed for touch screen so that infeasible for programming and debugging. If a language allows developers to program and debug in PC and run in any Android device (not Android emulator), productivity will be significantly enhanced.

Scientific Calculator Plus has made a great leap forward toward this target. From version 1.7.2, MFP includes a 2D game engine which unifies the interfaces in Android and in traditional JAVA. Developer can build a game script in PC, then run it everywhere. Developer can even package the game script into an Android APK and share the APK in google play with all over the world. For example, the following GemGem game was implemented using a single MFP script and it runs in both PC and Android.

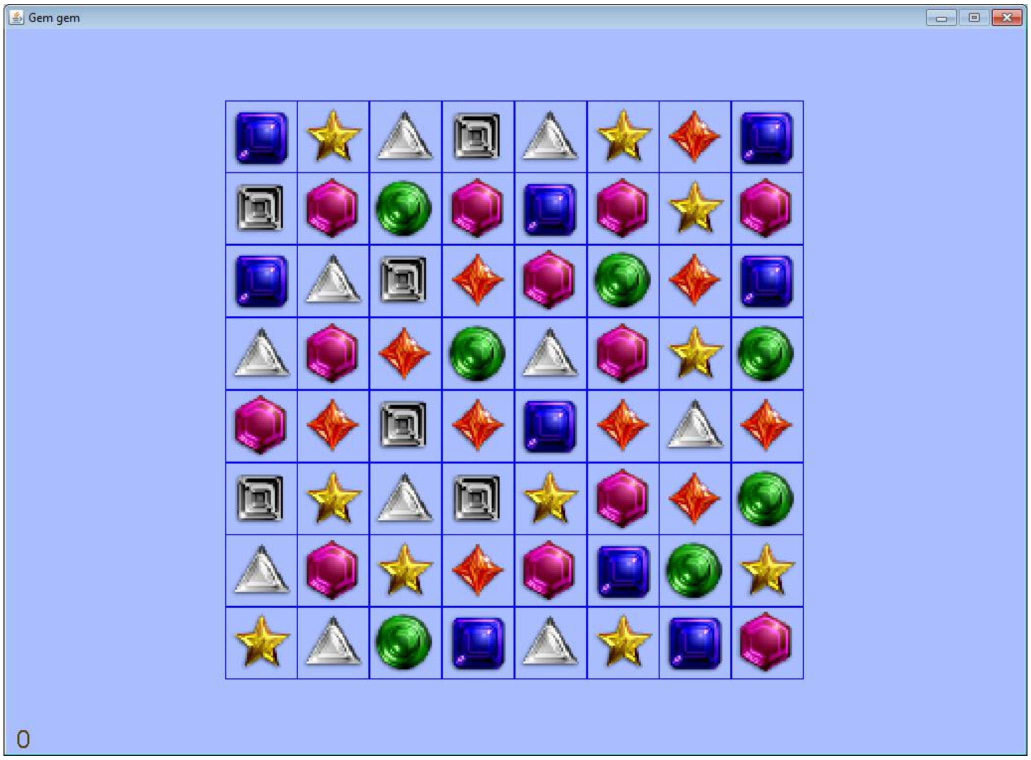


Figure 0.7: The GemGem game running in a PC.

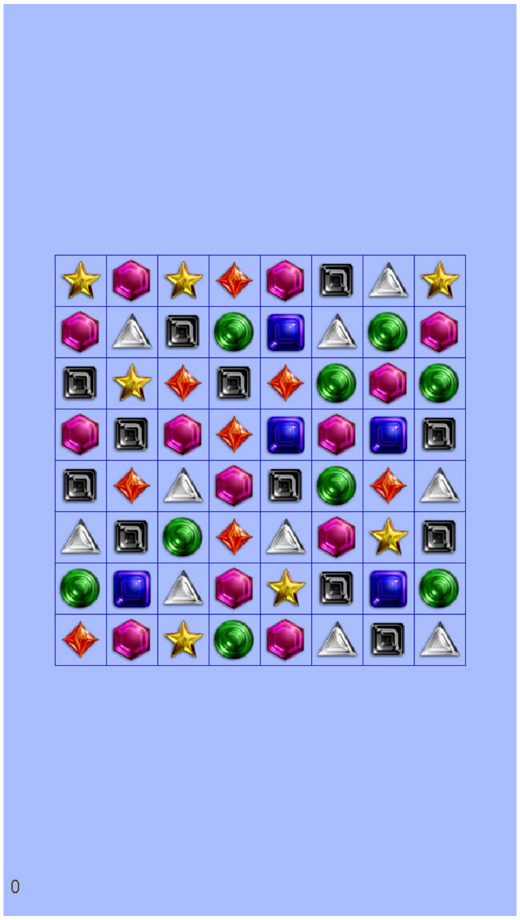


Figure 0.8: The GemGem game running in an Android mobile.

Now, besides mathematical calculation and game programming, Scientific Calculator Plus has been able to support drawing 2D/polar/3D graphs, text based input/output, file reading and writing, and time and date inquiry. But this is far from enough. Step by step, I will add APIs to access WWW, email and message, control camera, USB and Bluetooth, and create Graphic User Interface (GUI). My destination is to turn this app into a comprehensive and powerful scripting language (i.e. MFP) IDE, and a platform to rapidly build up any Android applications.

My target seems too massive for an individual developer. However, since the first release, so many functions and features, which were really difficult to realize, have been achieved gradually in this software. Moreover, everyday emails from users all over the world transfer their greetings and ask for new functionality. Users' encouragement also urges me to never give up. I will pursue my object, steadily and indefatigably. I believe that energy and persistence will conquer all things.

Chapter 1 Tutorial for Beginners

Scientific Calculator Plus is designed and developed for everyone who needs to do calculation. Nonetheless, most of the potential users are not programmers. However, they can still enjoy the facilitating functions offered by this app.

Section 1 Installation and Starting

Scientific Calculator Plus can be downloaded from Google Play, Samsung Apps and Amazon Store. User needs to log in and search "Scientific Calculator Plus", then download the apk file to install. Note that the size of apk is not small. For example, version 1.7.x is more than 50M bytes. If the Internet connection is not fast, user may wait quite a while for the finish of downloading.

After the software is setup and started, the main panel will be shown as in the following chart.

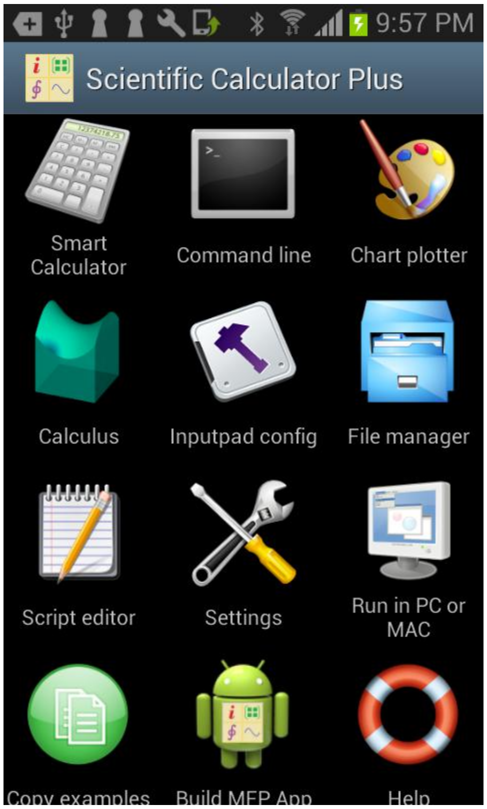


Figure 1.1: Main panel of Scientific Calculator Plus.

There are 12 modules listed in the main panel. The Smart Calculator module is specifically developed for users without any programming knowledge. This module is able to carry out all the mathematical calculations implemented in this software, plot 2D/3D/polar curves, recognize printed math expressions and record user's all historical activities, including both input and output (i.e. calculation result or plotted chart). Note that Smart Calculator needs no input of variable variation range to draw curves. Rather, it dynamically recalculates the plotted points aligning with any adjustment of the chart triggered by user. As such the input is simple (only a math expression) and user is able to see the whole picture of the curve. However, because of the recalculation, chart adjustment sometimes is very sluggish.

Command Line is a handy tool for engineers, researchers and science & technology students. Although not a must, basic programming knowledge is a reward to users of this component. It works like Matlab, DOS window or Linux terminal, i.e. user inputs a command and presses Enter, and then output will be printed below. Each command is either a mathematical expression or an MFP function call. The called MFP function can be either a built-in or a user-defined one.

Task of Chart Plotter is to draw 2D/polar/3D graphs for math expressions. Different from Smart Calculator, user needs to specify the variation range of each parameter before plotting. This characteristic ensures all the points are determined before generation of the graph. In other words, when user zooms or shifts the chart, no point needs to recalculate so that operation is very smooth and swift and user experience is improved.

Calculus tool calculates derivative expression and values, and indefinite and definite integrals. Second and third order derivative, and double even triple definite integrals are also supported.

Inputpad Config is designed to accelerate input. Instead of typing a function name letter by letter, with this tool user can define a button for the whole function name and place the button in the input pad of Smart Calculator or Command Line. Hence, multiple key typings are replaced by a single tap.

File Manager manages user-defined MFP source codes, user-created charts and user-built MFP Apps. User can open an MFP source file to edit in Script Editor, review a generated chart or install an MFP App package by long-clicking the file icon.

Settings are a collection of app-level configs e.g. scientific notation for the output, historical record length etc.

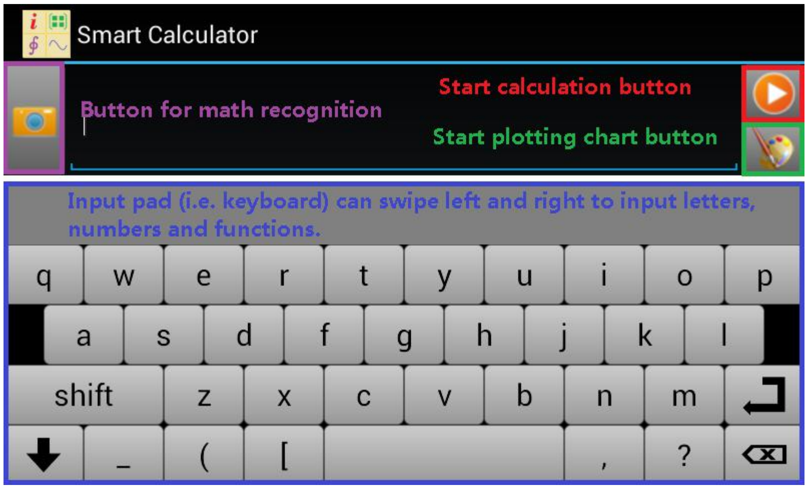
Run in PC or MAC is a guideline to set up Scientific Calculator Plus for JAVA. Details will be listed in the following sections.

Build MFP App is used in creating an Android app from an MFP function, whether it is a software built-in function or a user defined function. The generated APK file can then be installed or published.

Help provides HTML and PDF manuals, as well as sample codes. User is allowed to copy the sample codes (.mfps files) into scripts folder to read and run.

Section 2 Usage of Smart Calculator

Smart Calculator, which can be used by anyone with or without programming background, is the most important component of Scientific Calculator Plus. Its interface, which is similar to a traditional calculator, is shown in the following chart.



Button for math recognition

Start calculation button

Start plotting chart button

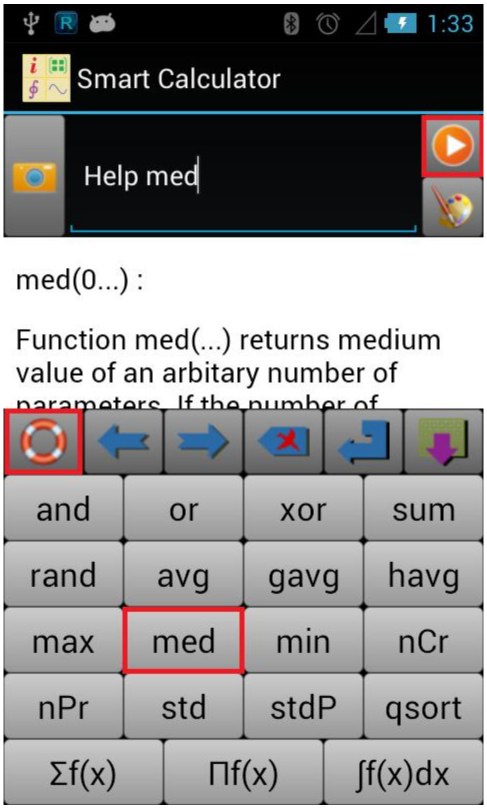
Input pad (i.e. keyboard) can swipe left and right to input letters, numbers and functions.

Figure 1.2: Interface of Smart Calculator.

2.1 Input

Input pad of Smart Calculator provides number & operator mode, letter mode and function mode. Letter mode includes a function dictionary so that input pad can match input characters to potential function names. User swipes input pad left and right to select one of the input modes.

If not sure about the usage of a function, user can click the help ring button (in the number and function name input mode) or the "?" button (in the letter input mode), and feed in function name, then tap the start calculation button to extract on-line help.



Help med

med(0...) :

Function med(...) returns medium value of an arbitary number of parameters. If the number of

Figure 1.3: Obtain function's on-line help.

If user prefers Android system keyboard rather than Smart Calculator's input pad, or has to input some characters unavailable in the input pad, he\she simply needs to tap the system menu key (which is located under the screen in most of legacy devices with a mark like ≡, or the right of the name bar on top of the screen with a mark like ⋮) and select "Enable Soft Input", system keyboard will pop up. Then user can tap system menu key again and select "Hide Soft Input" to go back to Smart Calculator's input pad. See Figure 1.4:

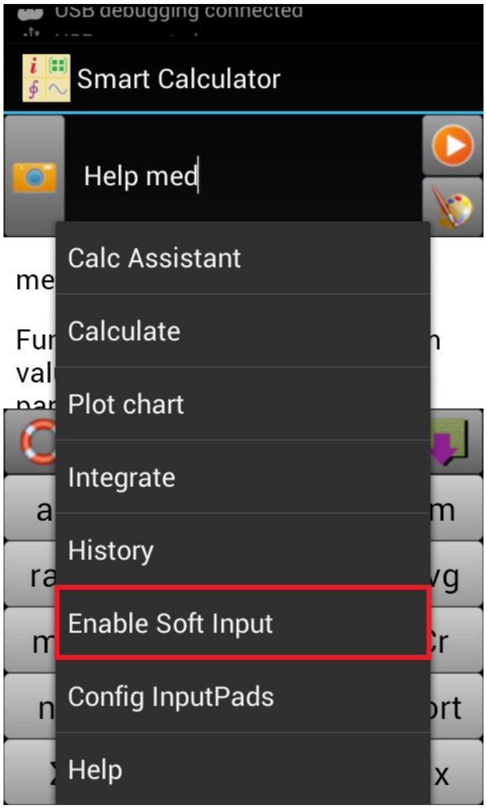


Figure 1.4: Select system keyboard or inputpad.

2.2 How to Calculate

Calculation is carried out by inputting an expression (e.g. 3 + log(4.1 / avg(1,5,-3)) or 4*x**2 + x == 3), or a group of expressions (one expression fits in one line) like

y1*3+4*y2-3*y3==7

y2/2-3*y3+y1==9

y3/3-6*y1+y2==2.4

, then tapping start button.

There are a few things user need keep in mind. First, in Scientific Calculator Plus equality is "==" while "=" is assignment. For example, to calculate x value where x + 3 is 5, user should input x + 3 == 5 instead of x + 3 = 5. But user can still assign a value to a variable, e.g. x = 7.

Second, in Scientific Calculator Plus power is "**" not "^". For example, to calculate x value where square x is 7, user should input x ** 2 == 7 instead of x ^ 2 == 7.

Third, in Smart Calculator user is able to input at most six expressions at once. Smart Calculator looks on all the expressions as an expression group to solve. The procedure is similar to *solve* block in MFP language. However, compared to PC, in the absence of powerful CPU and large memory, performance of Android devices limits iteration times to look up all possible solutions. In light of this, Smart Calculator only goes through the expressions once while *solve* block scans the expressions twice. Thus some expressions that can be handled by *solve* block may not be solved by Smart Calculator. For example, if user inputs

x**2 + 2*x == y

y + 1 == 2

, Smart Calculator can only get y value which is -1 but cannot obtain x since solving x needs y's result but Smart Calculator tries to solve x before y. Contrarily, if writing a script using an in-line *solve* block enclosing the two expressions (with the same order), and then running the script in Command Line or JAVA based Scientific Calculator Plus, user can get both x and y's values because MFP language goes back to the first expression after y is solved. Hence, when inputting expressions, user may try to place independent expressions on top before the expressions relying on other variables' values to solve.

2.3 How to Plot Graphs

User is able to input expression(s) to plot 2D, polar or 3D charts. Accepted input can be either equation, e.g. y**2 == sin(x) * x or t1 + t2 == t3, or assignment (=) with a single variable on the left side like a = b + 3. If an expression is neither equation nor assign expression, Scientific Calculator Plus will assume the value of the expression equals to another single variable. For example, if user inputs 2 * x + 5, Scientific Calculator Plus will automatically convert it to 2 * x + 5 == f_x where f_x is another single variable.

User can type at most 4 expressions in one input to plot a chart. If the total number of variables in all the expressions is 3, a 3D chart will be plotted. If the total number of variables in all the expressions is 2, and none of the variables is α, β, γ or θ, a 2D chart will be plotted. If the total number of variables in all the expressions is 2, and at least one of the variables is α, β, γ or θ, a polar chart will be plotted. For instance,

y=x+2z

z==sin(x)*y

x/abs(tan(x) + 1) == y

4==x

will be plotted as a 3D chart while

x + abs(x) - 3

4 + y == x

is a 2D chart because x + abs(x) - 3 will be automatically converted to x + abs(x) - 3 == y so that in total only two variables. And

log(r)

θ

will be plotted as a polar chart with θ value being the angle.

When a chart is shown, user may notice several operation icons in the bottom or on the right side of the chart for chart configuration, zooming in, zooming out, adjusting x, y (and z) ratio to 1:1(:1), and stretching plotted curves (surfaces) to fill the chart respectively. If user taps chart configuration (the gear icon), a dialog box will be popped up where user can adjust plotting range, set point number and enable/disable singular point detection. More than the operation buttons, user is able to use gestures, i.e. drag to move the curves and pinch to zoom the chart.

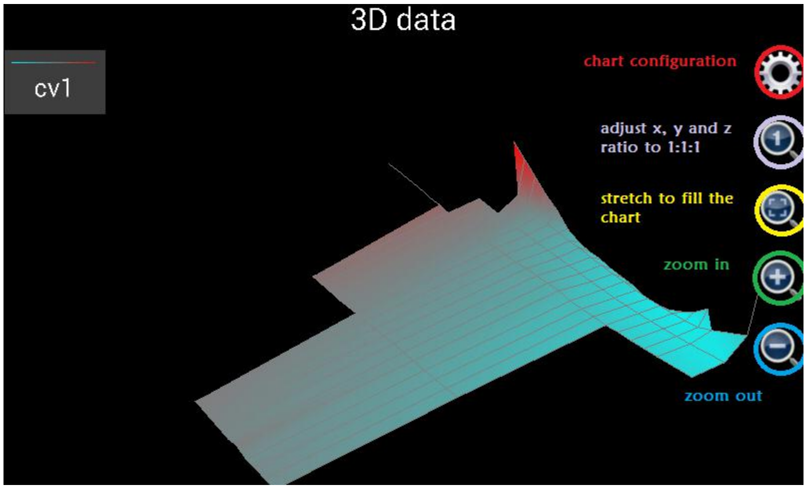


Figure 1.5: Generated chart.

Please note that the plot functionality in Smart Calculator is different from the independent Chart Plotter program. In Smart Calculator, expressions not data values are plotted. As such, if user changes plotting range by zooming or sliding the chart, Smart Calculator would automatically recalculate the expression values for the new plotting range. Comparatively, data range has been determined before chart is drawn by an independent Chart Plotter so that zooming or sliding will not change data range.

Also, if input expression is an implicit function, Smart Calculator tries to solve it first and then plot the root expression(s). If user plots a 2-variable implicit function, at most 4 root expressions will be drawn. If plot a 3-variable implicit function, Smart Calculator may calculate roots for each of the variables and then plot at most 2 roots for every variable. In other words, a single 3-variable implicit function may imply at most 6 to-be-plotted expressions. The whole solving-plotting procedure in this case will take quite long time.

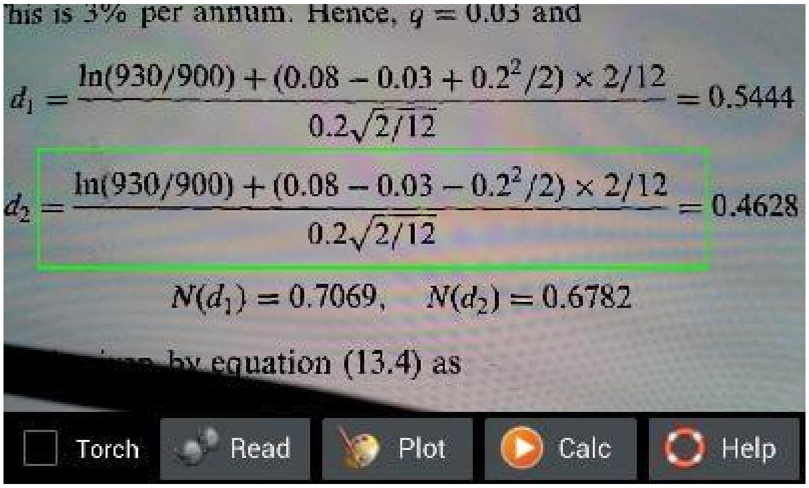
2.4 How to take photo of mathematical expressions and recognize them

If with back camera, Scientific Calculator Plus on Android provides very novel functionality to recognize printed mathematical expressions by taking photo of them. This allows user to input complicated expressions quickly and easily and makes calculation straight-forward. However, note that only printed expressions are supported. Scientific Calculator Plus is not able to recognize handwritings at this moment.

Steps to recognize mathematical expressions:

1. Start Smart Calculator, then tap the camera button left of the input box;

2. After Smart Calculator shows camera preview, tap one of the bottom buttons to take photo snapshot of selected range (inside the green rectangle) of a piece of white paper or computer screen. Note that user can adjust the size and position of the green rectangle. There are two things user has to keep in mind. First is that user should try to keep the expression(s) upright and avoid any inclination. Second is that the background (area excluding the expression(s)) in the green rectangle should be in lighter colour than the expression(s), e.g. background is white and expression is black, or background is light gray and expression is dark red. Moreover, background has to be as unique as possible. Light shade is acceptable. However, distinct colour change will be looked on as a stroke of math expression(s);



his is 3% per annum. Hence, $q = 0.03$ and

$$d_1 = \frac{\ln(930/900) + (0.08 - 0.03 + 0.2^2/2) \times 2/12}{0.2\sqrt{2/12}} = 0.5444$$

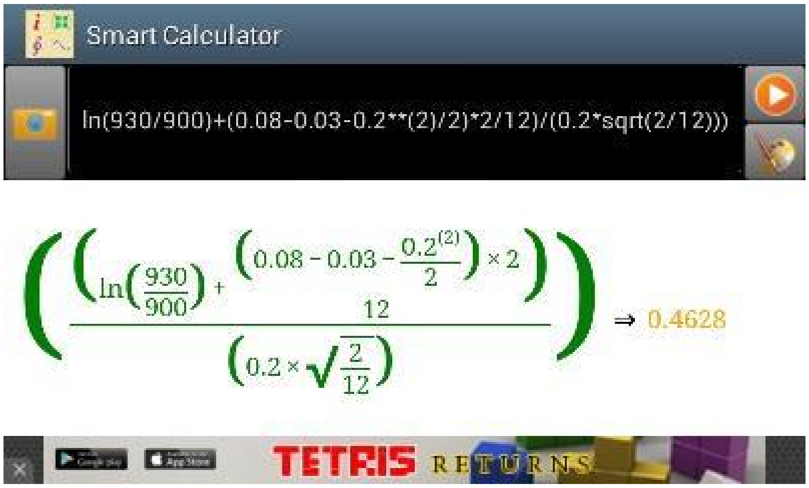
$$d_2 = \frac{\ln(930/900) + (0.08 - 0.03 - 0.2^2/2) \times 2/12}{0.2\sqrt{2/12}} = 0.4628$$

$$N(d_1) = 0.7069, \quad N(d_2) = 0.6782$$

by equation (13.4) as

Figure 1.6: Prepare to take phone of expressions.

3. Wait until Smart Calculator finishes recognition, then the app automatically proceeds to the Smart Calculator screen and continues to process the recognized expression. If user selected to plot chart, a 2D or 3D or polar chart will be drawn. If user selected to calculate, the calculated results will be shown. User can cancel the procedure at any time if it takes too long (if a simple expression needs long time to recognize, this generally implies misrecognition). After everything is done, if the recognized result is still not satisfactory, user can send email to us. User can also edit the recognized math expressions in the input text box and redo calculation;



```
ln(930/900)+(0.08-0.03-0.2**(2)/2)*2/12)/(0.2*sqrt(2/12)))
```

$$\left(\frac{\left(\ln\left(\frac{930}{900}\right) + \frac{\left(0.08 - 0.03 - \frac{0.2^{(2)}}{2}\right) \times 2}{12}\right)}{\left(0.2 \times \sqrt{\frac{2}{12}}\right)}\right) \Rightarrow 0.4628$$

Figure 1.7: Recognizing result.

Notes and requirements when taking photos:

1. Scientific Calculator Plus endeavors to recognize printed expressions. And printed formulas have to be as clear as possible (laser printing is preferred). Handwriting currently is not supported.

2. When taking photo of white paper, ensure that you are not in a dark environment. Otherwise, Smart Calculator cannot see clearly the expressions and recognition will fail. You can turn on flash by clicking the check box in the camera preview window. However, please note that ink on the white paper may reflect the flash light and, as a result, affected strokes become disconnected.

3. When taking photo of white paper, ensure that camera is about 10-30 cm above the paper (if paper lies on the table). Too far means characters are too small to recognize while too close may lead to a very strong shade of user's mobile projected on the paper and a very poor recognition as a result.

4. Taking photo of computer screen is more difficult than white paper because computer screen is not an integral surface but an array of pixels. Furthermore, screen is refreshing and unfortunately its refreshing cycle is close to camera's time of exposure. As such user should not place the camera too close to the computer screen. If one recognition is not successful, please try several more times.

Supported math expressions include:

1. Addition;

2. Subtraction;

3. Multiplication;

4. Division;

5. Fractions;

6. Roots;

7. Multiple linear expressions;

8. Trigonometry;

9. Polynomials;

10. Exponents;

11. Algebra;

12. Integration;

13. Derivative;

14. Summation (Σ);

15. Product (Π);

16. Matrix;

17. Complex value.

The math recognition functionality is still under improvement. Users are welcome to send email to us reporting any inaccurate recognition result. We also hope users be more patient and encourage us to make it better.

2.5 Output

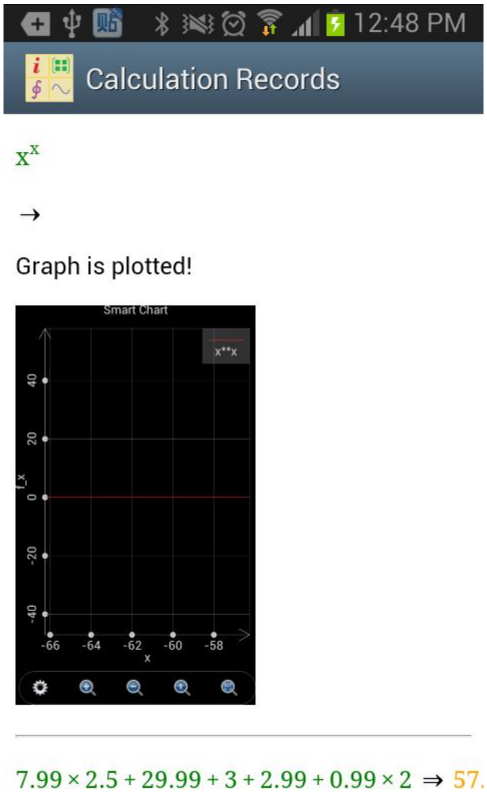
The output webview box of Smart Calculator shows user the calculated result or the snapshot of plotted chart (after exiting from the chart view). If the input of math expression is incomplete or incorrect, error message will be printed.

In order to accelerate inputting, user can tap any calculated results or expressions in the output box and copy their MFP strings into input box. If user taps the snapshot of a plotted chart, the full-size chart will be reloaded and shown.

One thing user has to keep in mind is that output of print, printf or any other printing functions cannot be shown in the output webview box. User has to use the Command Line tool to run scripts calling these functions.

2.6 Historical record

Historical record is a list of the activities user performs in the past. This record can be extracted by tapping menu key and selecting “History” entry. Similar to output box, user can tap any expression and copy its MFP string into input, or tap a chart snapshot to see the full-size chart.


12:48 PM

Calculation Records

x^x

→

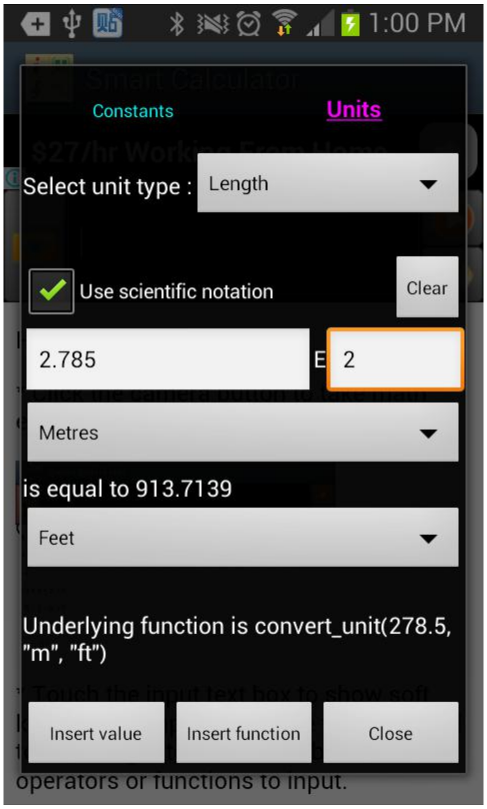
Graph is plotted!

$7.99 \times 2.5 + 29.99 + 3 + 2.99 + 0.99 \times 2 \Rightarrow$ 57.

Figure 1.8: Historical Record.

2.7 Calculation Assistant

User starts Calculation Assistant by tapping the menu key and selecting "Calc Assistant" entry in Smart Calculator. Calculation Assistant provides two tools. One is inserting a constant into input. The other is converting value from one unit to another unit. The unit conversion functionality is supported by an underlying function called convert_unit. If text in the input box is a valid real value, it is placed in the unit conversion tab as initial value to-be-converted. Otherwise, unit conversion tab does not place an initial value. If conversion is successful, user is able to select the converted value or conversion expression to insert.



Select unit type : Length

☑ Use scientific notation

2.785 E 2

Metres

is equal to 913.7139

Feet

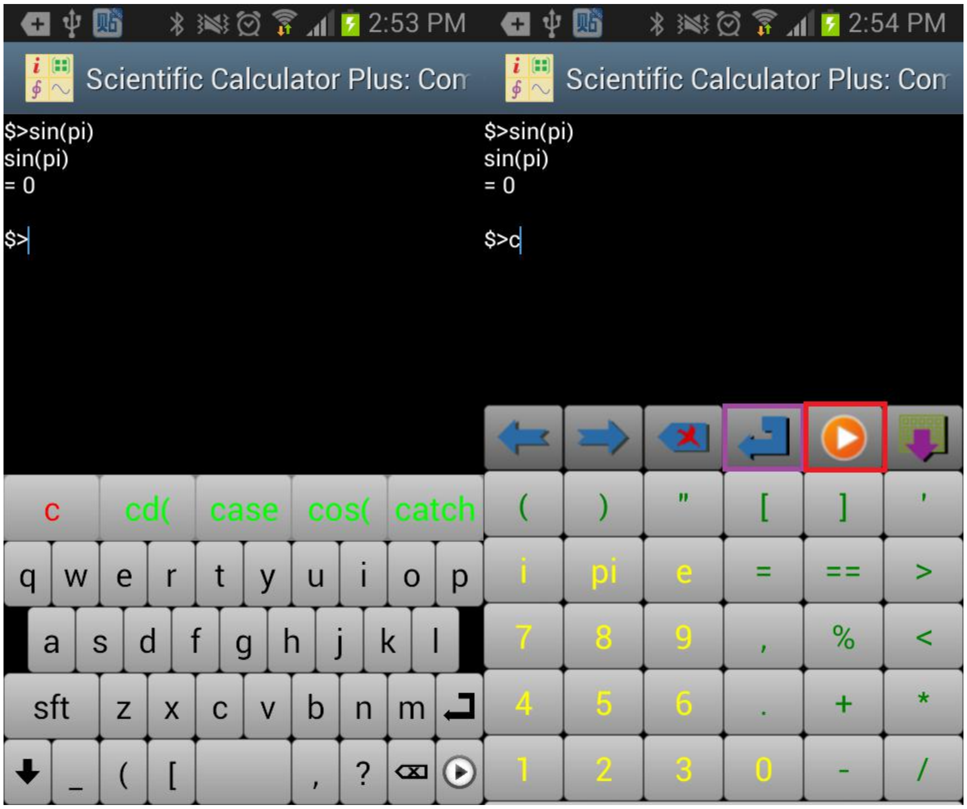
Underlying function is convert_unit(278.5, "m", "ft")

Figure 1.9: Unit conversion tool provided by Calculation Assistant.

Section 3 Usage of Command Line

Command Line is a useful tool besides Smart Calculator. It works like Windows command box, Unix terminal or Matlab command line window. User inputs something, presses ENTER, and sees the output and returned value in the following lines. The advantage of Command Line against calculator GUI is that command line window is able to show all the print outputs while calculator GUI only shows returned value and exception.

Similar to Smart Calculator, Command Line also provides its input pad. User swipes the input pad left and right to input numbers, letters or function names and key words. A function name dictionary is activated when inputting letters so that user is able to quickly input the whole function name by just typing the first few characters and selecting the right name from name candidate list. Input pad of Command Line includes both start button and ENTER button (the button in the purple rectangle in the following chart). Pressing start button triggers execution of current command or command batch. Pressing ENTER button does not run a command or command batch but finishes the input line of scanf or input function, or starts a new line when inputting a multi-line command batch. Here a multi-line command batch means the command includes several lines each of which can be any independent command statement except function and endf. These commands run as a batch.

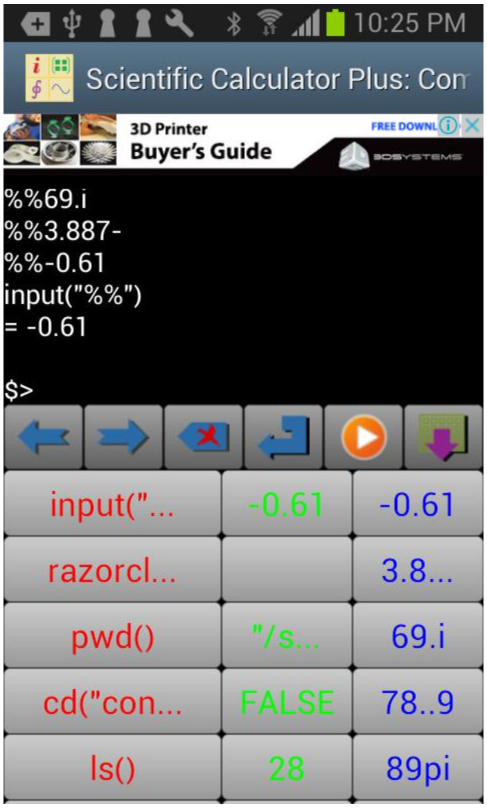


```
$>sin(pi)
sin(pi)
= 0

$>
```

```
$>sin(pi)
sin(pi)
= 0

$>c
```

Figure 1.10: Input and output of Command Line.

Some users have special requirements to input historical records swiftly and high-frequently. To this end, Command Line includes a history record input pad. All of user's inputs and calculation results are listed. The red column is the executed command; the green column is the output result; and blue one is user's input for input function. The max number of records shown in this input pad is determined by App settings. By default, it is 20 records.



```
%%69.i
%%3.887-
%%-0.61
input("%%")
= -0.61

$>
```

Figure 1.11: Quickly input historical records using historical record input pad.

It is quite possible that user needs to input some special characters or Unicode char not included in the input pad. If so, user can tap Android menu key and select "Show system soft keyboard" menu to activate system soft keyboard. If user wants to go back, simply tap Android menu key again and select "Hide system soft keyboard". Then next time when user inputs, input pad will be popped up.

GUI based Scientific Calculator Plus for JAVA is also a command line tool and supports multiple lines in one command batch. User can input more than one lines in one command batch using copy and paste, or type Shift-ENTER to start a new line in current command batch. Different from the Command Line tool in Android, typing ENTER in Scientific Calculator Plus for JAVA triggers the execution of current command or command batch.

Command Line supports global variables. However, different from Matlab, user needs to declare global variables before using them. For example, user can type the following command: Variable a, b, c , then assign values to the declared variables (i.e. input commands like a = "hello world", b = 4 and c = 5 + 3.7i), and then use a, b, c in other commands like print(a) or exp(c).

Like Matlab, there is also a pre-defined global variable "ans" which stores last result except if last command returns nothing. However, different from Matlab, assign statement in MFP programming language also has a return value which is the assigned value. For instance, statement c = 5 + 7.3i returns 5 + 7.3i. In this way, if user types c = 5 + 7.3i in Command Line, 5 + 7.3i will be assigned to "ans" variable.

Like Smart Calculator, user is able to access Calculation Assisant by tapping Android menu key and selecting "Calc Assistant" entry. User can insert a constant value or convert value from one unit to another unit with the help of this assistant. The initial value to-be-converted in units conversion tab is copied from last command's result if it is a valid real value. If not, initial value is copied from user's type-in for current command if it is a valid real value. Otherwise, unit conversion tab does not show an initial value.

Section 4 Chart Plotter

The Smart Calculator tool in Scientific Calculator Plus has been able to plot graphs. However, all the graphs plotted by Smart Calculator are based on independent mathematical equations. If user needs to draw some complicated charts in selected ranges, the independent Chart Plotter is the right choice.

The independent Chart Plotter can plot 2D, polar and 3D charts. The ways to draw different types of charts are quite similar. For example, in order to draw a 2D chart, firstly user is required to set chart name (i.e. chart file's name), chart title, x and y axis titles, and show grid or not. Then user needs to tap the "Add curve" button one or several times to add (at most 8) curves to draw.

In order to define a curve, user has to input curve's title, colour, shape of sample points and pattern of connection lines between sample points. All these configs are intuitive. The difficult part is what is t and how to config X(t) and Y(t).

It is well known that any non-branched 2D curve can be looked on as the track of a moving point. The projection of the track on x axis is the point's x-coordinate function of t, i.e. X(t). Similarly, the projection of the track on y axis is the point's y-coordinate function of t, i.e. Y(t).

Scientific Calculator Plus chooses this way to define a curve because sometimes an x value can be mapped to several points in a 2D curve (same as some y values). For example, a circle in a traditional 2D (not polar) coordinate cannot be drawn from single equation in Smart Calculator. However, using the independent chart plotter, two functions of t can be selected to define x and y. If the radius of the circle is 2.5, the center of the circle is (1.3, -1.7), then X(t) is 2.5*cos(t)-1.3 and Y(t) is 2.5*sin(t)+1.7, t changes from 0 to 2*pi. In this way, this circle is accurately defined and can be drawn by the chart plotter.

The above solution is able to draw simple curves as well as complicated graphs. The approach is letting X(t) equal t, then Y(t) becomes the to-be-drawn function, not of x but of t. For example, user could set X(t) to be t and Y(t) to be t**2 and let t change from –t to 5. Then a parabolic curve y == x**2 where x is from –5 to 5 is drawn.

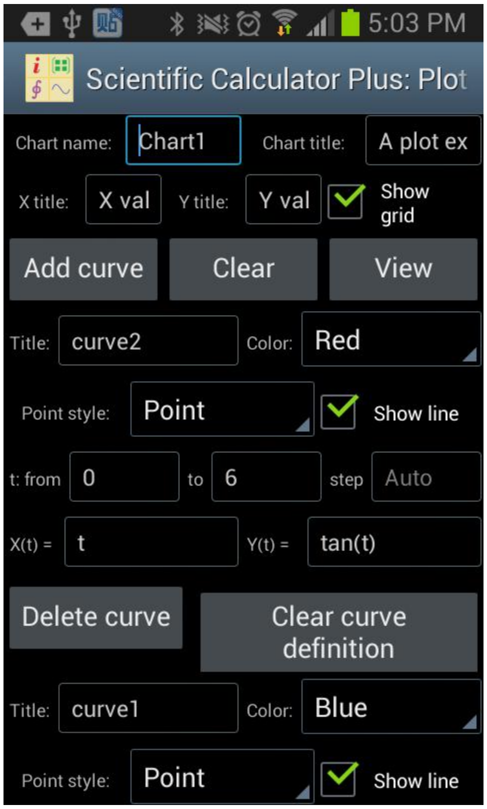


Chart name: Chart1 Chart title: A plot ex

X title: X val Y title: Y val ☑ Show grid

Title: curve2 Color: Red

Point style: Point ☑ Show line

t: from 0 to 6 step Auto

X(t) = t Y(t) = tan(t)

Title: curve1 Color: Blue

Point style: Point ☑ Show line

Figure 1.12: Using the independent Chart Plotter to draw 2D chart.

The follows are examples to draw different types of 2D curves using the independent Chart Plotter. For example, to draw a line segment from (3, 5) to (3, 15), user could set t from 5 to 15, set step Auto, X(t) is 3 and Y(t) is t.

If a curve includes singular points, i.e. y = tan(x), user could set t from -2*pi to 2*pi, set step Auto, X(t) is t and Y(t) is tan(t). The plotted curve is shown as the left part in the following chart.

However, if the step of t is not Auto, i.e. user sets a step value, e.g. 0.1, Scientific Calculator Plus will not automatically detect singular points. As such the plotted curve is shown as the right part in the following chart.

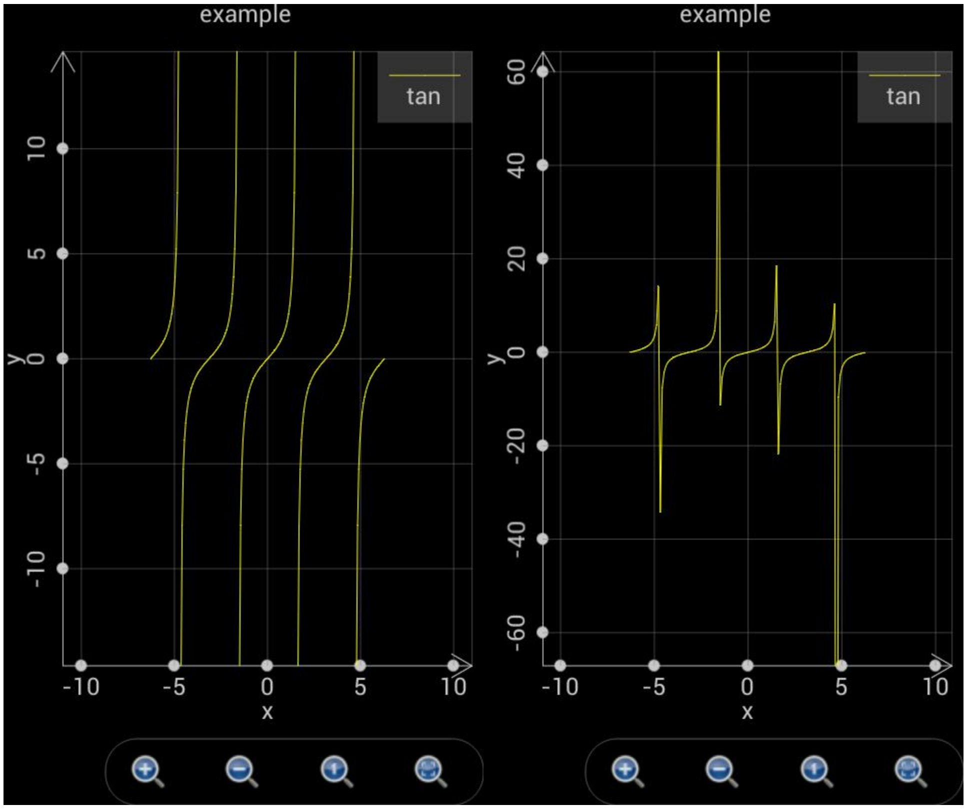


Figure 1.13: Scientific Calculator Plus is able to detect singular points only if the step is set Auto.

It is quite similar to draw polar charts as to plot 2D graphs. The only thing to keep in mind is that in a polar chart X(t) and Y(t) are replaced by r(t) and θ(t), where r is the radius and θ is the angle.

For example, if t changes from 0 to 2*pi, step is Auto, r(t) is cos(t) and θ(t) is t, a circle is plotted as the green curve shown in the following chart.

If t changes from -2*pi to 2*pi, step is Auto, r(t) is 2*sin(4*t) and θ(t) is t, a chamomile is drawn as the blue curve shown in the following chart.

If t changes from -1.5*pi to 1.5*pi, step is Auto, r(t) is t and θ(t) is t, a heart shape is plotted as the magenta curve shown in the following chart.

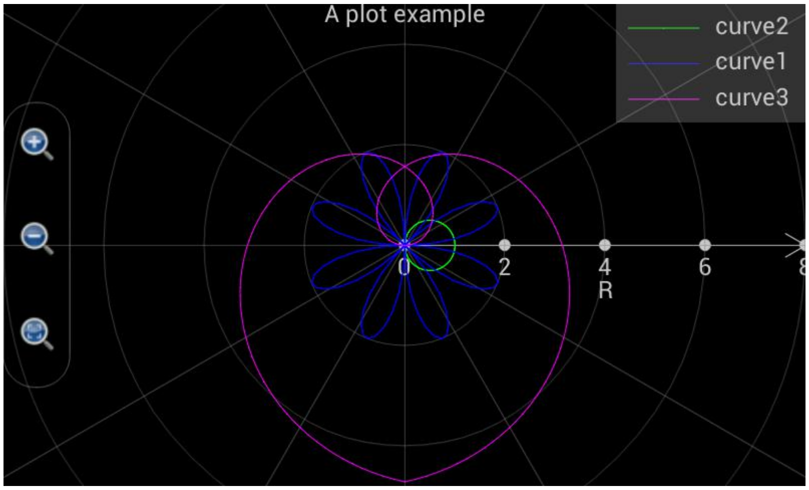
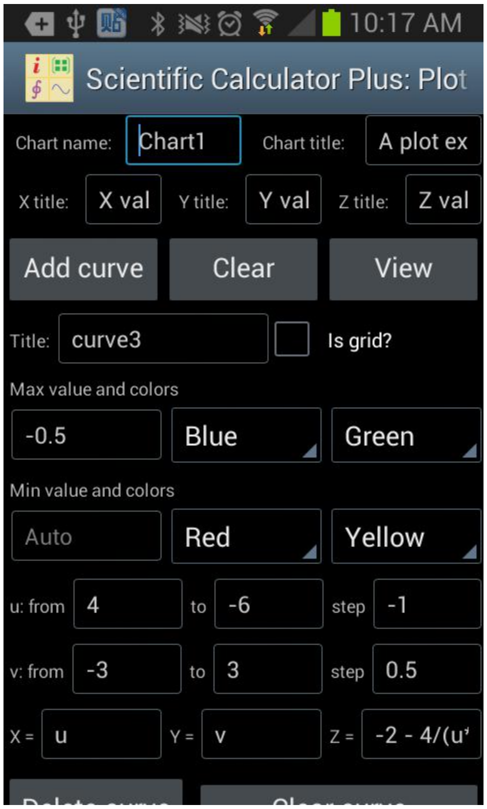


Figure 1.14: Draw various curves in a polar coordinate system.

Plotting a 3D graph is a bit more complicated. As shown in the following chart, user is required to input more settings. It is quite straight-forward to set up graph name, title, x, y and z axis title etc. In the curve settings, "Is grid?" means Scientific Calculator Plus will only draw grid, i.e. skeleton, of the curve without filling it. Max value and colours means, when Z value is no less than the max value, the colours on the front and back sides of the surface. Min value and colours means, when Z value is no greater than the max value, the colours on the front and back sides of the surface. If Z value is between max and min, the colour is a transition between max colour and min colour.

Different from a 2D chart, X, Y and Z are functions of internal variables u and v, not t. Scientific Calculator Plus introduces two internal variables instead of one because it is drawing a 3D surface instead of a 2D curve.

By setting X(u,v), Y(u,v) and Z(u,v), user is able to draw very interesting 3D surfaces. To plot 3D curve instead of 3D surface, user can set X, Y and Z functions of u only, i.e. the change of X, Y and Z are irrelevant to the dynamics of v.



10:17 AM

Scientific Calculator Plus: Plot

Chart name: Chart1 Chart title: A plot ex

X title: X val Y title: Y val Z title: Z val

Add curve | Clear | View

Title: curve3 □ Is grid?

Max value and colors

-0.5 | Blue | Green

Min value and colors

Auto | Red | Yellow

u: from 4 to -6 step -1

v: from -3 to 3 step 0.5

X = u Y = v Z = -2 - 4/(u'

Figure 1.15: Draw 3D chart using the independent Chart Plotter.

The following examples demonstrate how to plot 3D charts. If user wants to draw a flat surface at x = 10 and in parallel with y and z, u can change from 0 to 10 with auto step, v is also from 0 to 10 with auto step, x is 10, y is v and z is u. The plotted flat surface is shown as the following figure.

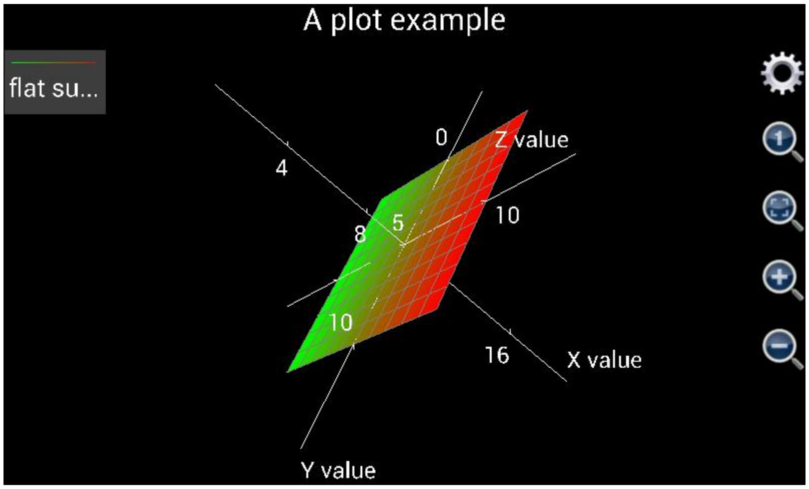


Figure 1.16: A flat surface at x = 10.

User may need to draw a line segment from one point to another point in a 3D coordinate system, e.g. from (x,y,z)=(1,5,6)to (x,y,z)=(10,3,9). In this case, u is set from 1 to 10, step is auto. Since x, y and z are irrelevant to v, v can be arbitrarily configed. However, user may keep in mind that the number of calculations equals u's number of steps times v's number of steps. It is important to keep the number of calculations minimum to accelerate the plotting. Therefore, if x, y and z are irrelevant to v, v can change from 0 to 1 with a step of 1 so that v's number of steps is only 1. Then x is configured as u, y is (u-1)/(10-1)*(3-5)+5 and z is (u-1)/(10-1)*(9-6)+6 so that x and y and x and z are linearly correlated. Another thing user has to keep in mind is that the "Is grid?" check box must be selected. Otherwise, the line segment will be gray colored.

The plotted line segment is shown in the following chart.

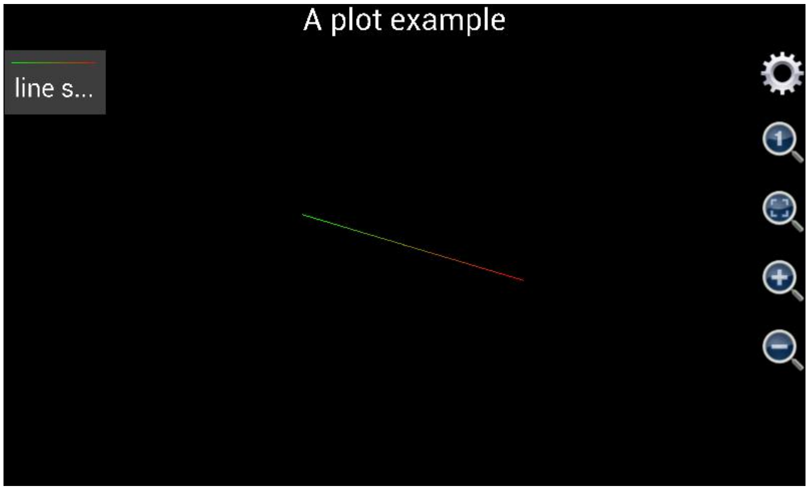
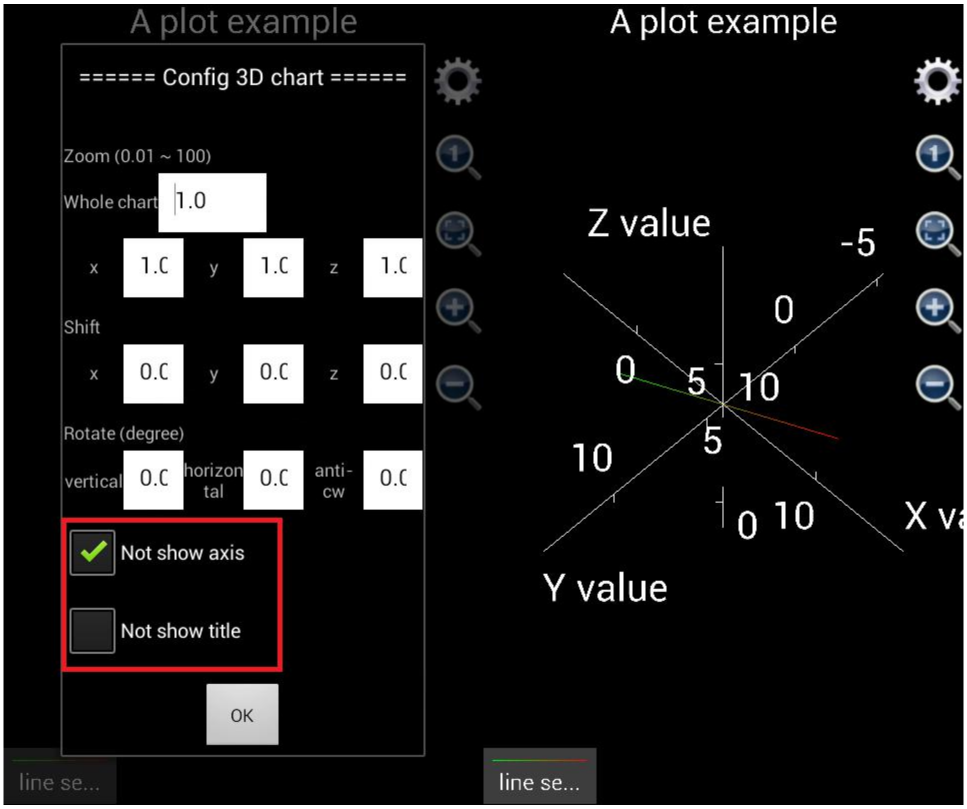


Figure 1.17: Draw line segment in a 3D coordinate system.

Please note that, before version 1.6.7, Scientific Calculator Plus shows x, y and z axes by default. If the axes are not wanted, user can tap the gear button and select "Not show axes and title" check box. This action will hide the axes as well as the title of the chart. Since version 1.6.7, Scientific Calculator Plus automatically hides the axes after the graph is plotted. If user wants to see the axes, s\he has to tap the gear button and uncheck the "not show axis" box. Similarly, user can hide and show title.



A plot example

====== Config 3D chart ======

Zoom (0.01 ~ 100)

Whole chart 1.0

x 1.C y 1.C z 1.C

Shift

x 0.C y 0.C z 0.C

Rotate (degree)

vertical 0.C horizontal 0.C anti-cw 0.C

☑ Not show axis

☐ Not show title

OK

Figure 1.18: Show and hide axes and title in a 3D chart.

In order to draw more complicated shapes, e.g. sphere, user may look on u as the degree of longitude on the ball surface, and v as the degree of latitude. Then x can be set as 3*cos(v)*cos(u) where 3 is the radius of the ball, y is 3*cos(v)*sin(u), z is 3*sin(v), u is from 0 to 2*pi with auto step, v is from –pi/2 to pi/2 with auto step.

If using version 1.6.6 or earlier, the plotted graph is shown as the left part of the following chart. It is not a ball but an ellipsoid. The reason is that x, y and z axes have different unit lengths. User has to tap the "adjust to 1:1:1" button (in the red circle) to see the right shape. However, from version 1.6.7, x, y and z axes have been automatically adjusted to 1:1:1 when the graph is plotted. Therefore the plotted chart would be similar to the right part of the following figure.

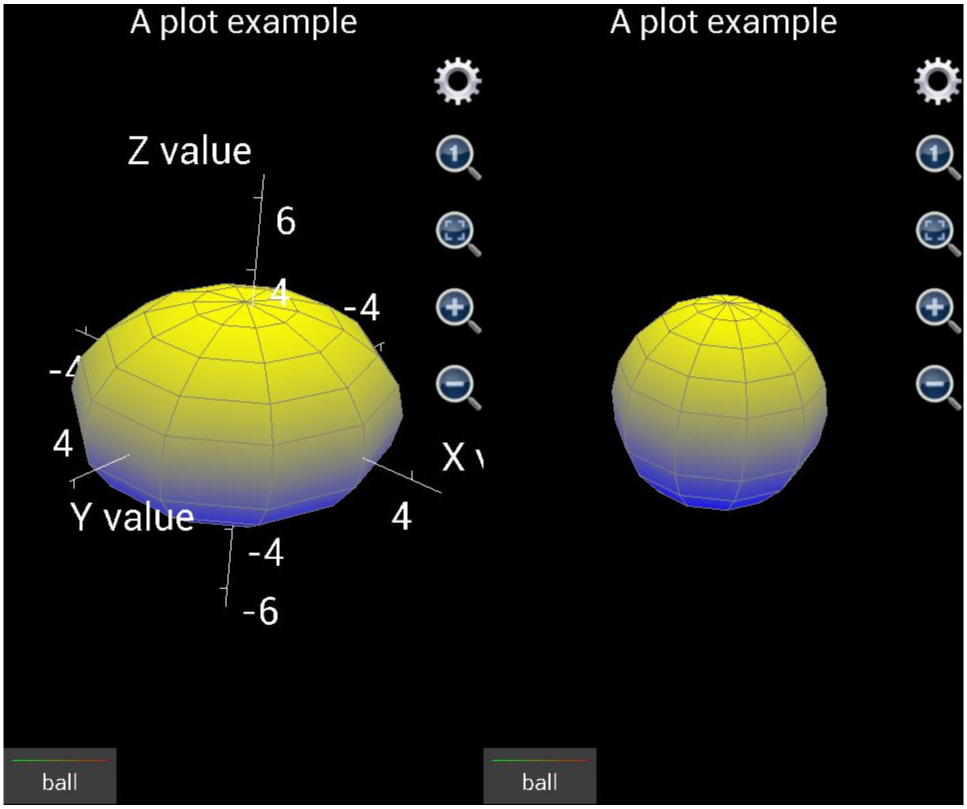


Figure 1.19: Draw a ball in a 3D coordinate system.

If user wants to draw a ball similar to earth, i.e. Arctic and Antarctic poles are white but close to equator it is green, two hemispheres (curves) have to be drawn. The first curve is the northern hemisphere. The “Is grid?” box should be unchecked. Max value and colours should be Auto, White and White (White means ice cap in Arctic) respectively. Min value and colours should be Auto, Green and Green (Green means the forests in the area around equator) respectively. X is 3*cos(v)*cos(u) where 3 is the radius. Y is 3*cos(v)*sin(u) and z is 3*sin(v). Here u is from 0 to 2*pi with auto step, and v is from 0 to pi/2 with auto step.

The second hemisphere is southern hemisphere. Similar to northern hemisphere, “Is grid?” box should be unchecked. Min value and colours should be Auto, White and White respectively, max value and colours should be Auto, Green and Green respectively. X is 3*cos(v)*cos(u) where 3 is the radius. Y is 3*cos(v)*sin(u) and z is 3*sin(v). Here u is from 0 to 2*pi with auto step, and v is from –pi/2 to 0 with auto step.

The plotted chart is shown below:

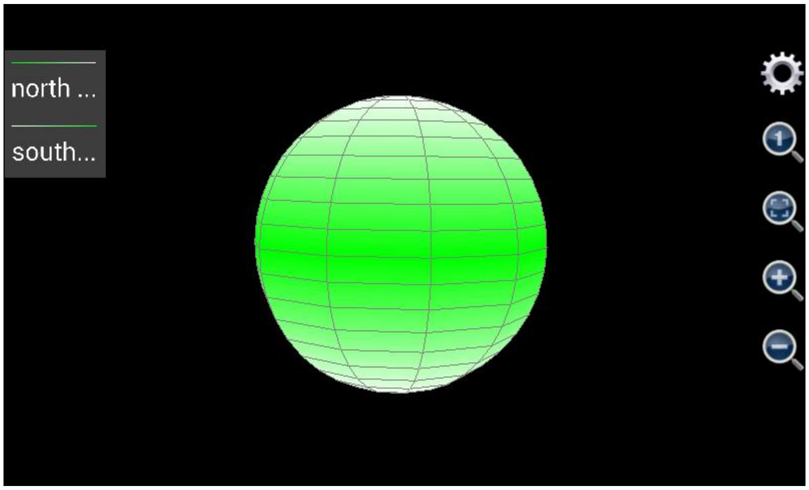


Figure 1.20: Draw a ball similar to earth.

Though, the approach is a bit different from drawing a ball, Chart Plotter is able to draw a cylinder. A ball has only one surface while a cylinder has three. For example, the cylinder to draw has a radius of 5, its bottom surface's elevation is 0, and its height is 20. Since the bottom surface is a solid circle, let u be the angle ranging from 0 to 2 with a step length equal to 0.05 (this means angle changes from 0 to 2*pi and a changing step is 0.05*pi), and let v be the radius ranging from 0 to 5 with a step length equal to 5. Then x should be v*cos(u*pi), y should be v*sin(u*pi), z is the altitude so that it is always 0. Max value and colours are auto, red and red respectively, same as min value and colours because the thickness of the surface is 0.

Drawing the top surface of the cylinder is almost the same as the bottom one. Let u be the angle ranging from 0 to 2 with a 0.05 step length. Let v be the radius which is from 0 to 5 with a step equal to 5. Again, z is the altitude whose value is always 20. Max value and colours are auto, blue and blue respectively, same as min value and colour.

In order to draw the side surface, let u still be and angle changing from 0 to 2 with a step being 0.05. Now v is the altitude of each point on the side surface. Since the height of the cylinder is 20, v varies from 0 to 20 and step length is 20. Since the radius of any point on the side surface is always 5, x is 5*cos(u*pi) and y is 5*sin(u*pi). Once again, z is the altitude so that it always equals v. Since the upper surface is blue and lower surface is red, the max value and colour of side surface should be auto, blue and blue respectively, and the min value and colour should be auto, red and red respectively. The graph is shown as follows.

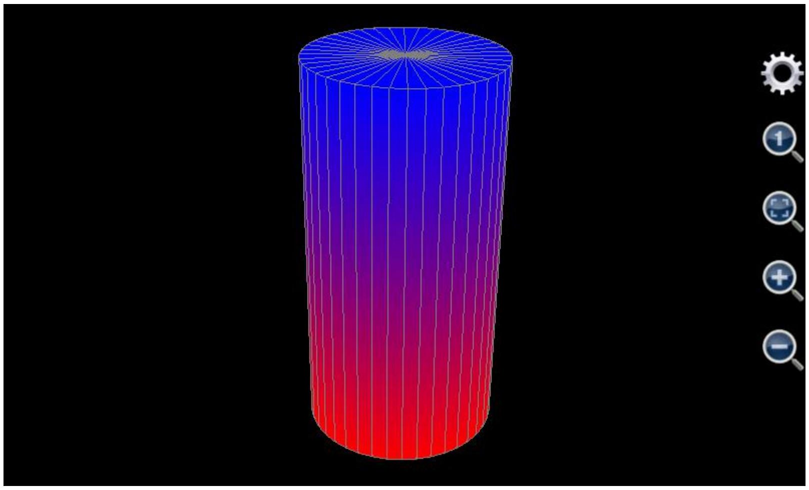


Figure 1.21: Draw a cylinder.

User can also draw cone(s) using this tool. A cone has two surfaces, i.e. bottom surface and side surface. The bottom one is drawn in the same way as a cylinder whereas the radius of point on the side surface is not constant but monotone decreasing with the point's altitude.

If, for example, the maximum radius (which should be at the bottom) of the cone to draw is 5, altitude of the bottom is 0, its height is 20. Its bottom surface is the same as the bottom surface of the cylinder in Figure 1.20: . For each point on its side surface, u is its angle varying from 0 to 2 with a constant step equal to 0.05 (this means the angle's variation range is from 0 to 2*pi and variation step length is 0.05*pi). V is the altitude of each point on the side surface. Its variation range is from 0 to 20 and step length is 20. Because radius of point on the side surface monotonically decreases with height, the radius should equal 5*(20-v)/20. As such, x is 5*(20-v)/20*cos(u*pi), y is 5*(20-v)/20*sin(u*pi), and z is v. The maximum value and colours of the side surface is auto, blue and blue respectively. The minimum value and colours of the side surface is auto, red and red respectively so that they match the colour of the bottom surface. The plotted chart is shown as below.

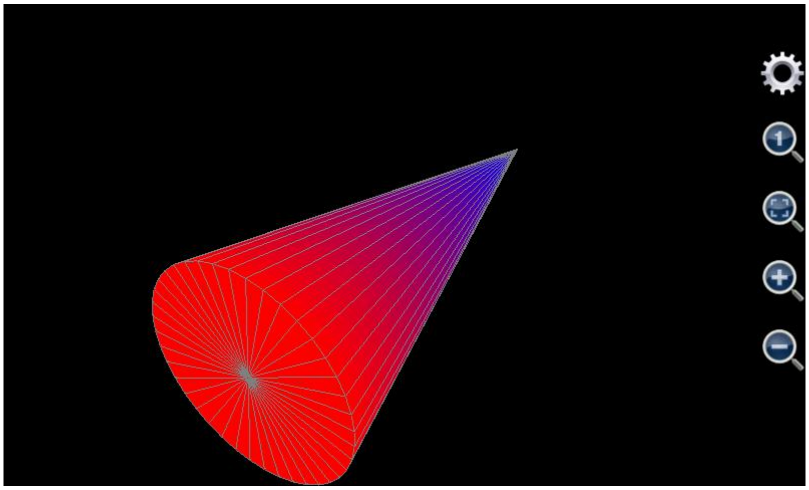


Figure 1.22: Draw a cone.

This tool can also draw many funny and interesting shapes. For instance, if u's range is from 0 to 2*pi, v is from 0 to 10, x is v*cos(u), y is v*sin(u), and z is 6*cos(v)*exp(-v/10), the graph it generates would be like follows.

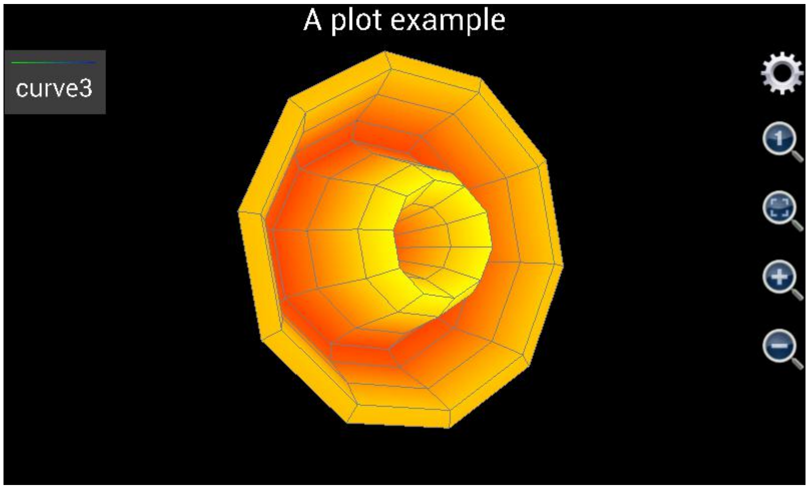


Figure 1.23: Draw a flower like graph.

Another example is using 3D Chart Plotter to draw a spiral line. To this end, user may set v from 0 to 10 with a step length 0.1. X is v*cos(v), y is v*sin(v), and z is v. U can be any value. However, as explained above, to ensure minimum calculation time, u should be from 0 to 1 with a step length equal to 1. And the "Is grid?" box must be checked otherwise the 3D line can only have a gray colour.

The plotted graph is shown as below:

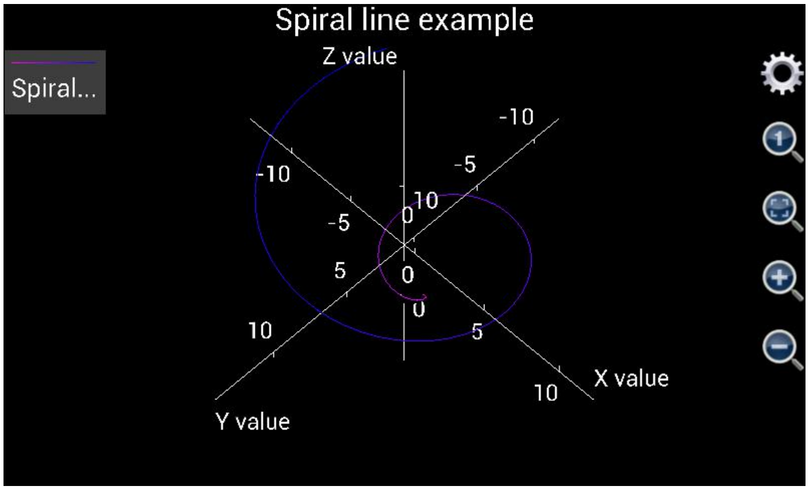


Figure 1.24: Draw a 3D Spiral line.

In the end of this section, a more complicated 3D graph example is demonstrated. This example shows user step by step how to draw the Oriental Pearl Radio & TV Tower in Shanghai. The photo of this building is shown below:

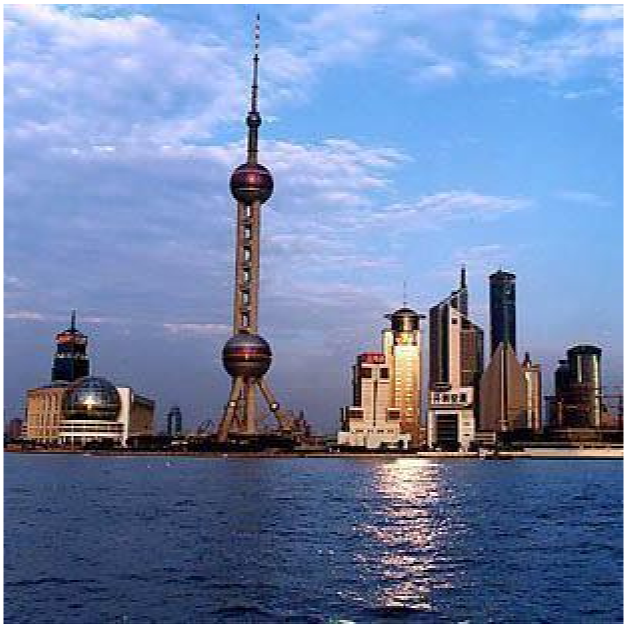


Figure 1.25: Photo of the Oriental Pearl Radio & TV Tower in Shanghai, P.R.C.

In order to draw the TV Tower, first of all user needs to find out what the building is comprised of.

The bottom of the building includes three leaning pillars to support the whole building. To draw them, the title can be empty, minimum colour is red and maximum colour is yellow, minimum and maximum values are auto (i.e. user does not input). Assume the radius of each pillar is 3, angle of inclination is 45°, and height is 20. The central points of the three pillars' bottom surfaces are the vertexes of a regular triangle. Their coordinates are (-20*sqrt(3)/2,-10,0), (0,20,0) and (20*sqrt(3)/2,-10,0) respectively. Let u be the angle, v be the radius, u varies from 0 to 8 which means that angle ranges from 0 to 8*pi, and u's step length is 0.25 (i.e. 1/4*pi). V's range is from 0 to 20 and step is 20. As such, x's expression is iff(u<=2,3*cos(u*pi)-(20-v)*sqrt(3)/2,and(u>=3,u<=5), 3*cos(u*pi), u>=6, 3*cos(u*pi)+(20-v)*sqrt(3)/2, Nan), y's expression is iff(u<=2,3*sin(u*pi)+(20-v)/2,and(u>=3,u<=5), 3*sin(u*pi)-(20-v)*sqrt(3)/2, u>=6, 3*sin(u*pi)+(20-v)/2, Nan). And z's expression is v.

User needs to keep in mind that the above x, y and z expressions draw three surfaces together. So question is, why use one instead of three groups of x, y and z to draw the three surfaces? Answer is, the number of surfaces in the Oriental Pearl TV Tower is more than 8, while user can input at most 8 groups of expressions in the 3D Chart Plotter. Thus at least one group of expressions need to draw more than one surfaces.

In order to draw multiple surfaces by one group of x, y and z expressions, iff (i.e. if function) must be employed to determine the condition to draw each surface. For the three supporting pillars, u's range is from 0 to 8 which means the angle varies from 0 to 8*pi. Since a supporting pillar's angle variation range only covers from 0 to 2*pi, by using iff function the first pillar is drawn when u is from 0 to 2 (which means angle ranges from 0 to 2*pi), the second pillar is drawn when u is from 3 to 5 (which means angle ranges from 3*pi to 5*pi), and the last one is drawn when u is from 6 to 8 (which means angle ranges from 6*pi to 8*pi). In this way, the x, y and z expressions differs at different u ranges. When u is between 2 and 3 or between 5 and 6, x, y and z are all Nan.

So another question is, why do we need the gaps between 2 and 3 and between 5 and 6 in u's variation range. In other words, why not set u to change from 0 to 6 so that range from 0 to 2 is for the first pillar, range from 2 to 4 is for the second one and range from 4 to 6 is for the last pillar?

The answer is, although using iff function can draw different surface at different u range, the surfaces still connect to each other if no gap between them. By introducing the range gaps and setting point value to be Nan in the gaps, we disconnect the three pillar surfaces because Nan point cannot be drawn.

The (20-v)*sqrt(3)/2 and (20-v)/2 parts in x and y's expressions realize the inclination of the pillars. They imply that the x and y coordinates of the three pillar drift with increment of z.

Notice that between the three leaning pillars an erect column stand connecting the ground and upper part of the TV tower. To draw it, user may set its title blank, uncheck the "is grid?"box, and set the minimum colour green, maximum colour yellow, and the minimum and maximum values both auto. U value ranges from -1 to 1 with a step length equal to 0.25, and v value ranges from 0 to 20 with a step length equal to 20. Assume the radius of the erect column is 2, then x is cos(u*pi)*2, y is sin(u*pi)*2, and z is the altitude which always equals v.

Above the columns of the TV tower user see a big ball. Its title is blank. The "is grid?" box is unchecked. The minimum colour is cyan and minimum value is auto. The maximum colour is red and maximum colour is auto. Assume the radius of the ball is 10. And the center of sphere is (0,0,20). U changes from –pi to pi with a step length of pi/10. V varies from –pi/2 to pi/2 with a step length of pi/10. The expression of x is 10*cos(u)*cos(v). The expression of y is 10*sin(u)*cos(v). The expression of z is 10*sin(v)+20.

Above the sphere we see three erect columns. Once again, a single group of x, y and z expressions is applied to draw the three columns in one breath. The title of the columns is blank. "Is grid?" box is unchecked. Max value and colour are auto and blue respectively. Min value and colour are auto and green. Assume the radius of each column is 1.5. X and y coordinates of center of each column are (-2,2/sqrt(3)), (0,4/sqrt(3)) and (2,2/sqrt(3)) respectively. Let u be the angle and v be the radius. Using the same approach as drawing the three leaning pillars, user sets u from 0 to 8 (i.e. angle changes from 0 to 8*pi) with a step length of 0.25. V ranges from 20 to 70 with a step length of 50. Therefore, expression of x is iff(u<=2,1.5*cos(u*pi)-2,and(u>=3,u<=5), 1.5*cos(u*pi), u>=6, 1.5*cos(u*pi)+2, Nan), expression of y is iff(u<=2,1.5*sin(u*pi)+2/sqrt(3),and(u>=3,u<=5), 1.5*sin(u*pi)- 4/sqrt(3), u>=6, 1.5*sin(u*pi)+2/sqrt(3), Nan), and z is v.

A small ball stands above the three erect columns. Its title is empty. "Is grid?" is unchecked. Minimum colour and value are auto and magenta respectively. Maximum colour and value are auto and white respectively. The center of sphere is (0,0,70) and ball's radius is 6. U is from –pi to pi and its step length is pi/10. V is from –pi/2 to pi/2 and its step length is also pi/10. X's expression is 6*cos(u)*cos(v), y's expression is 6*sin(u)*cos(v) and z's expression is 6*sin(v)+70.

User sees a smaller erect cylinder above the small ball. Its title is empty. It is not drawn as a grid. Minimum value and colour are auto and yellow respective. Maximum value and colour are auto and green respective. Assume height of the cylinder is 15. X and y coordinates of the cylinder's center points are (0,0). The radius is 1.5. Let u be the angle and v be the radius. U varies from 0 to 2 (i.e. the angle changes from 0 to 2*pi) and its step length is 0.25. V changes from 70 to 85 with a step of 15. X's expression in this way is cos(u*pi)*1.5 and y is sin(u*pi)*1.5. Z is still v.

A tiny ball sits just on top of the erect cylinder. Like other components, its title is empty and it is not a grid. Minimum value and colour are auto and red respectively. Maximum value and colour are auto and cyan respectively. The center of sphere is at (0,0,85). Radius is 2. U is from –pi to pi and step of u is pi/10. V is from –pi/2 to pi/2 and its step is pi/10. Then x is 2*cos(u)*cos(v), y is 2*sin(u)*cos(v) and z is 2*sin(v) + 85.

The last component to draw is the antenna. Its title is empty and it is not a grid. Its minimum value and colour are auto and red respectively. Maximum value and colour are auto and light gray (ltgray) respectively. The radius at the bottom of the antenna is 0.5. The height of the antenna is 30. The center of the bottom surface is (0,0,85). Let u be the angle and change from –pi to pi with a step of pi/5. Let v be the altitude and change from 85 to 115 with a step of 10. Assume the radius of antenna at any height cannot be smaller than 0.2 times its maximum radius (i.e. the radius at its bottom which equals 0.5). Then x's expression is 0.5*max(0.2,(115-v)/30)*cos(u*pi), y's expression is 0.5*max(0.2,(115-v)/30)*sin(u*pi) and z is v.

The above settings are very complicated and almost unfeasible to manually input into a mobile device. Fortunately, since version 1.6.7, Scientific Calculator Plus provides a short-cut to input all the above expressions. After starting 3D Chart Plotter, user simply needs to tap Android menu button and select "fill example", then all the entries are automatically filled in. Then user taps the "View" button to see the chart. Since the TV tower is a very complicated graph, it may take several minute to draw, depending on the hardware performance.

If user is using Scientific Calculator Plus version 1.6.6 or earlier, the plotted graph is shown as the left part of the following chart. Clearly, x, y and z are not in the right ratio. User has to tap the "adjust ratio to 1:1:1" icon (circled in red). Then user may tap the gear icon (circled in green) to hide axes and title. The adjusted graph is demonstrated as the right part of the chart below.

If user is using Scientific Calculator Plus version 1.6.7 or newer, the x, y and z ratio is automatically adjusted to 1:1:1 and chart title and axes are automatically hidden on the spot. As such the graph is the same as the right part of below and is very similar to the photo of Shanghai Oriental Pearl TV tower.

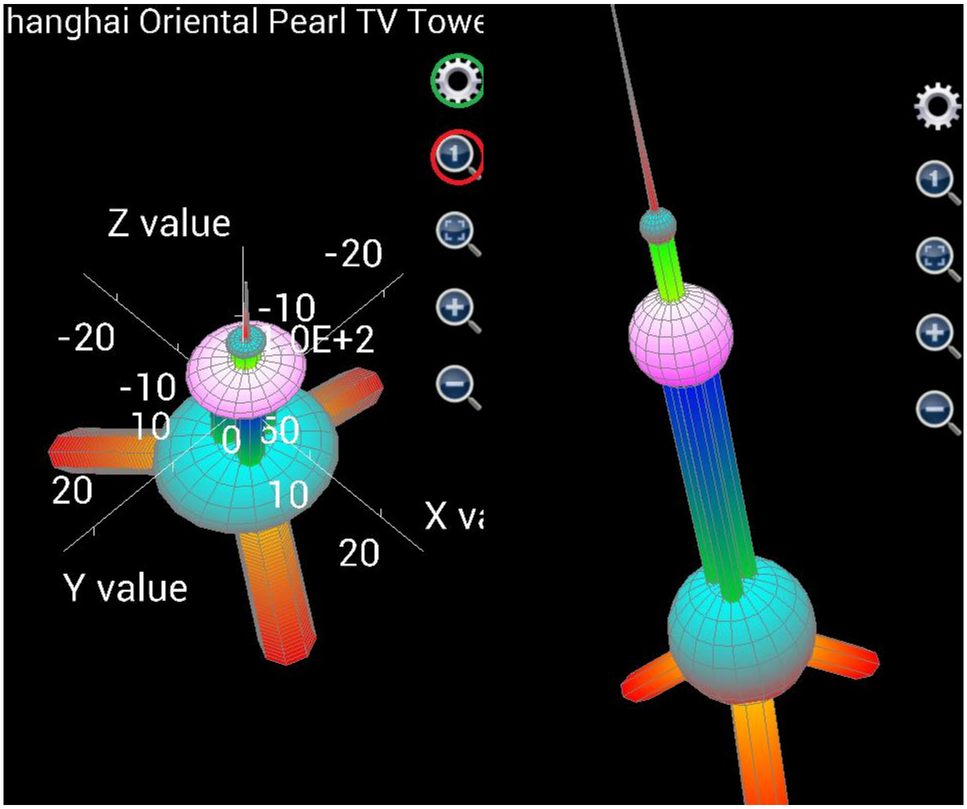


Figure 1.26: Draw Shanghai Oriental Pearl TV Tower.

Nevertheless, typing complicated expressions in a mobile device is never an easy job. So running an MFP script to call plot3d or plot_3d_surface function is the best alternative. If user knows how to program MFP language and how to use plot3d and plot_3d_surface, a simple MFP function can be implemented in a text file and saved into the mobile device. Then by simply typing the user-defined MFP function name in the Command Line tool (only several letters to input), user will be able to draw the TV tower. Detailed script has been provided in Chapter 5 in this manual.

Section 5 Derivative/Integral Calculator

Scientific Calculator Plus has a very easy-to-use Derivative Calculator tool. It is able to calculate first, second and third derivative expression and value. If user does not input variable value, the Derivative Calculator gives out derivative expression. Otherwise, derivative value is provided. This tool is simply a wrapper of derivative (to calculate derivative expression) and deri_ridders (to calculate derivative value) functions.

The built-in Integral Calculator tool in Scientific Calculator Plus supports both indefinite integral and up to triple definite integral. When calculating definite integrals, the starting and ending values can be real number, complex number or even positive or negative infinite. The number of steps in the calculation must be a non-negative integer. If the number of steps is 0, or the integral's starting and/or ending value is infinite, Gauss-Kronrod method is employed. Otherwise, Riemann Sum method is selected. Gauss-Kronrod method is much more accurate than Riemann Sum and can handle some singular points, but it is much slower so that it is not recommended for a double or triple integral.

In essence, Integral Calculator is the graphic interface of the integrate function provided by MFP programming language. For example, if the integrated expression is x+3, and the integrated variable name is x, Integral Calculator actually calls Integrate("x+3", "x") and gets the result of "3*x+0.5*x**2", which means the result expression is 3 times x plus half of x square. Another example is calculating definite integration of exp(x) from –infinite to 0. User simply inputs exp(x) as the integrated expression, x as the integrated variable, starting value is –inf and ending value is 0. The number of calculation steps can be any non-negative integer because Gauss-Kronrod method is used anyway. And the final result is 1.

Section 6 Inputpad Config

User is able to define several inputpads and use the inputpads to rapidly input functions in calculator. Each inputpad has its own portrait mode and landscape mode for portrait and landscape orientation respectively. User is also able to add or delete an inputpad from menu.

Each inputpad has a name which is used internally by calculator and must be unique. There are also long name, wrapped name (lines 1 and 2) and short name. These name are for backward and forward compatibility and are not used at this moment. If user unticks "Visible" check box, the inputpad will be hidden.

Each inputpad is a table includes a number of columns (the number of columns may be different in portrait mode from in landscape mode). Each input key is a cell in the inputpad table. Each key has a shown name (which will be shown in the key button) and an input name (which will be input to calculator). User can edit an input key by tapping it or cut/copy/paste/delete an input key by long-clicking it and selecting a context menu. The last button in each table is not an input key. Its role is adding a number of buttons to the tail of the input key list.

The inputpad config file is stored in mobile device's SD card/AnMath/config folder with the name of inputpad.cfg (for Smart Calculator) or inputpad_cl.cfg (for Command Line). If this file does not exist, Scientific Calculator Plus loads default inputpads. Thus, user may manually delete this file to revert back to the default inputpads provided by Scientific Calculator Plus.

Section 7 File Manager and Script Editor

Scientific Calculator Plus stores user's data files in an external storage, e.g. SD card. When Scientific Calculator Plus starts, it checks if its data folder is available. If not, it tries to create its folder in the first external storage device. If unsuccessful, an error message will be prompted. The path of the folder can be seen in the settings view.

User can view and manage data files using an internal file management tool. In File Manager, user is able to select a file or folder by tapping its icon. If a file or folder is selected, its background colour turns to orange. User can run/open (depends on the file type) a file or folder by long-clicking its icon or selecting the item first then clicking “run” or “open” menu. User can also delete or rename a selected file or folder using “delete” or “rename” menu.

User can open a program source file to view or edit it. Scientific Calculator Plus has an internal editor with basic editing capability. User can open a chart file and view the chart. In order to study the format of a chart file, user may copy the chart file to PC and open the chart file by a text editor to see what is inside.

Section 8 Settings

Scientific Calculator Plus has the following Settings:

1. Bits of precision. This setting determines how many effective bits after decimal point should be shown. For example, if it is 4, 0.003204876 will be rounded to 0.003205. Note that this setting is also applied to the Command Line tool.

2. Scientific notation. This field sets the value range that scientific notation, e.g. 2.1e-37, should be used to demonstrate result. Note that this setting is also applied to Command Line.

3. Record length. The record length gives out the number of historical calculations recorded by Smart Calculator. Note that this setting is also applied to Command Line. But Command Line does not share its records with Smart Calculator.

4. Variable range to plot charts. In Smart Calculator, if user wants to plot chart for expression(s), initial variable range is set here. By default, it is from -5 to 5.

5. Auto start program. The auto start program is the component which is automatically launched when user starts this app. By default, no component is automatically loaded so that user sees the main panel.

6. Vibrate when press calculator button. Check this item if user wants the mobile device to vibrate when pressing a calculator button.

7. Do not hide input pad when running a command in command line. By default, this feature is disabled so that each time when user runs a command, input pad is hidden to maximize output visible area. Then after the command finishes, user has to tap the screen to show input pad again. Check this item if user wants to run commands high-frequently and avoid the time cost to activate input pad after a command finishes.

8. Folders' paths. User is able to select which storage is used for the application data folder. There are also several read-only settings for folders' paths. First one is app data folder, second is script folder, third is chart folder, fourth is apk file folder and last is signature file folder. These settings will be editable in the future.

Section 9 Running JAVA based Scientific Calculator Plus in PC or Mac

Scientific Calculator Plus supports both Android mobile devices and desktop computers. From version 1.1, Scientific Calculator Plus for Android includes a JAVA/Swing based Scientific Calculator Plus. Any computer, whether running Windows, MacOSX or Linux, can run this calculator if JAVA 6 or above version is installed. From version 1.7.1, a system console based Scientific Calculator Plus for JAVA is added. This is a pure MFP scripting language interpreter. With this tool, an mfps script can run from a Dos box or a Linux terminal like Python or Perl. User can start the "Run in PC or Mac" tool to copy the two Scientific Calculator Plus for JAVA tools into the AnMath fold in SD card.

Each time user upgrades Scientific Calculator Plus in an Android device, Scientific Calculator Plus for JAVA is automatically copied to SD card. As such, normally there is no need to launch the "Run in PC or Mac" tool. However, sometimes Scientific Calculator Plus for JAVA may not be properly copied, or it is deleted by user carelessly. The "Run in PC or Mac" tool becomes useful in these scenarios.

The "Run in PC or Mac" tool includes two steps. The first step only copies Scientific Calculator Plus for JAVA into the SD card. The second step asks user to confirm. After the two steps finish, user needs a USB cord to connect Android device to a PC with JAVA installed, as shown in the following chart:

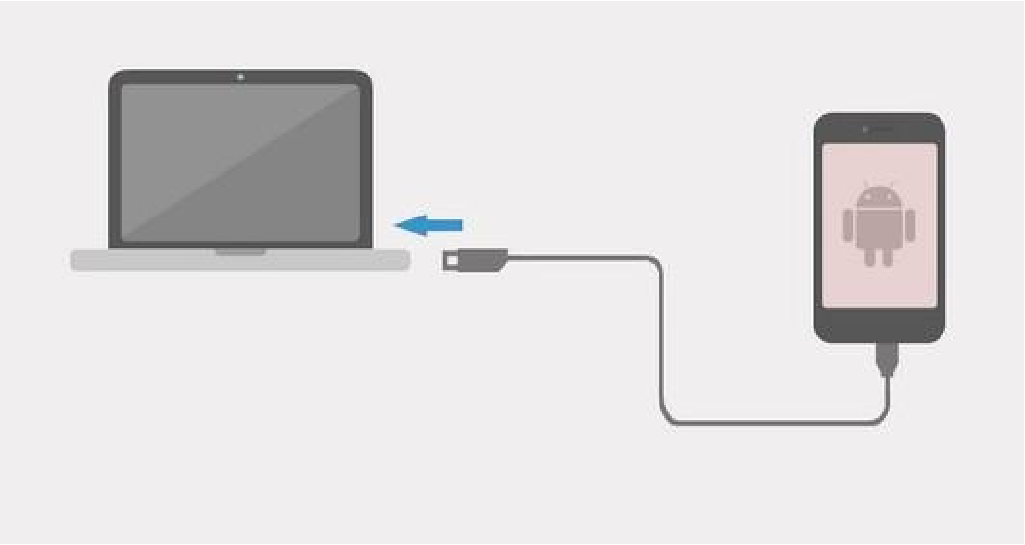
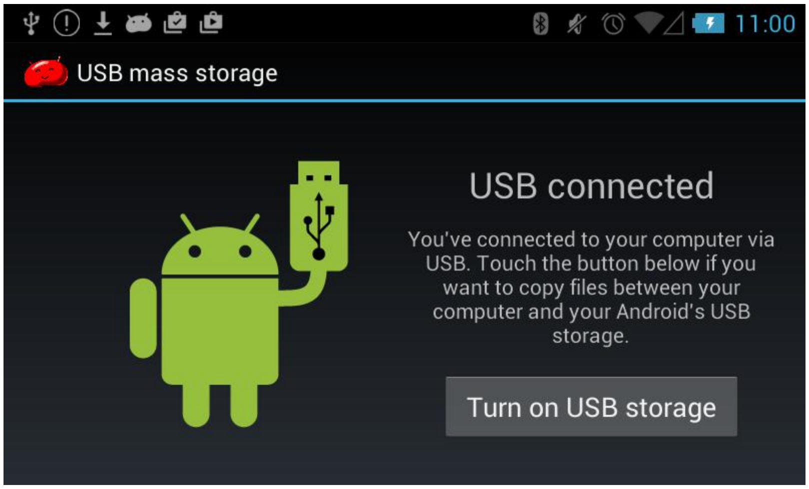


Figure 1.27: Connect an Android device to a PC with JAVA installed using a USB cable.

After connected to a PC, SD card of some Android devices can be automatically founded by computer while other Android devices only prompt user about to exchange files with PC. In the second situation user needs to tap to confirm, then computer is able to locate SD card in the Android device, as shown in the following chart:



USB connected

You've connected to your computer via USB. Touch the button below if you want to copy files between your computer and your Android's USB storage.

Figure 1.28: Some Android devices ask user to confirm turning on USB storage and allowing computer to find Android's SD card after connected to computer.

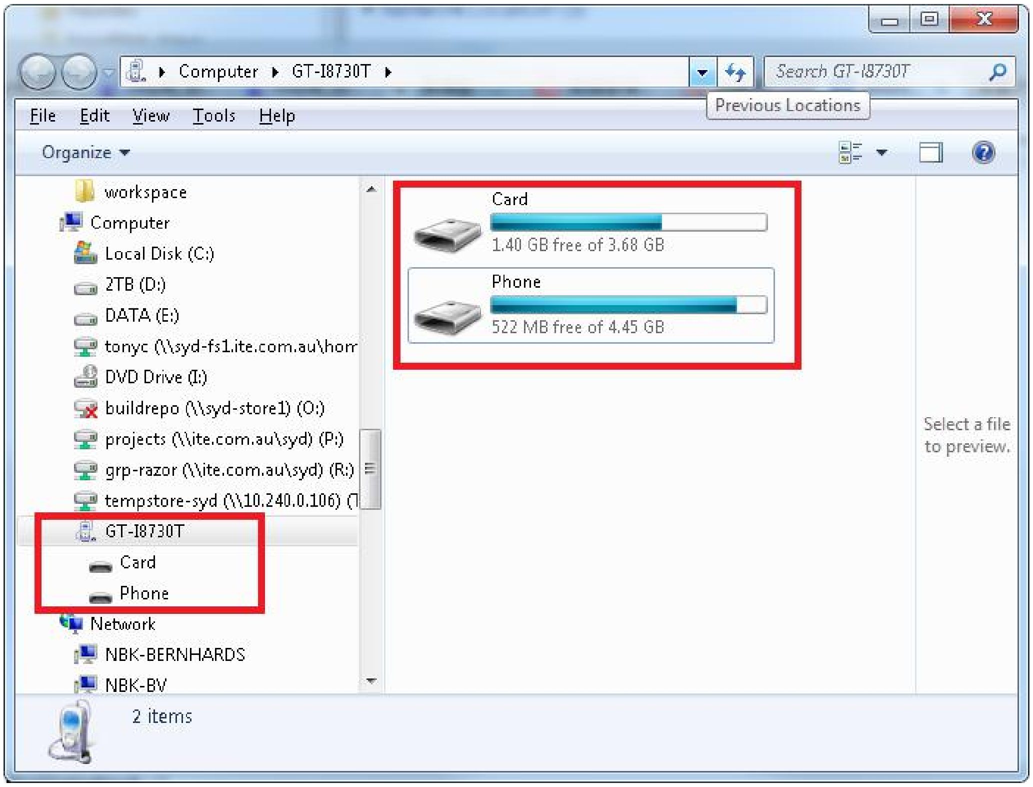


Figure 1.29: Computer finds Android's SD card.

If the above steps are successfully done, user will see the mapped partition(s) of Android SD card(s) in My Computer, as shown in the above picture.

Please note that some Android devices have more than one SD cards. First of them is the built-in (internal) storage, and others are extended (external) storage(s). By default, Scientific Calculator Plus for JAVA is copied to the first SD card, i.e. the built-in storage. However, user is able to change the target SD card in settings view. In the above example, name of first USB storage (SD card) is Phone. User can find the AnMath folder in this SD card, and then run JMathCmd.jar or mfplang to start Scientific Calculator Plus for JAVA. User is able to write and test MFP scripts in PC using JAVA based Scientific Calculator Plus, and run the tested MFP programs in the Android device later on if the program files are properly copied into the AnMath\scripts folder. The whole procedure is illustrated in the following chart.

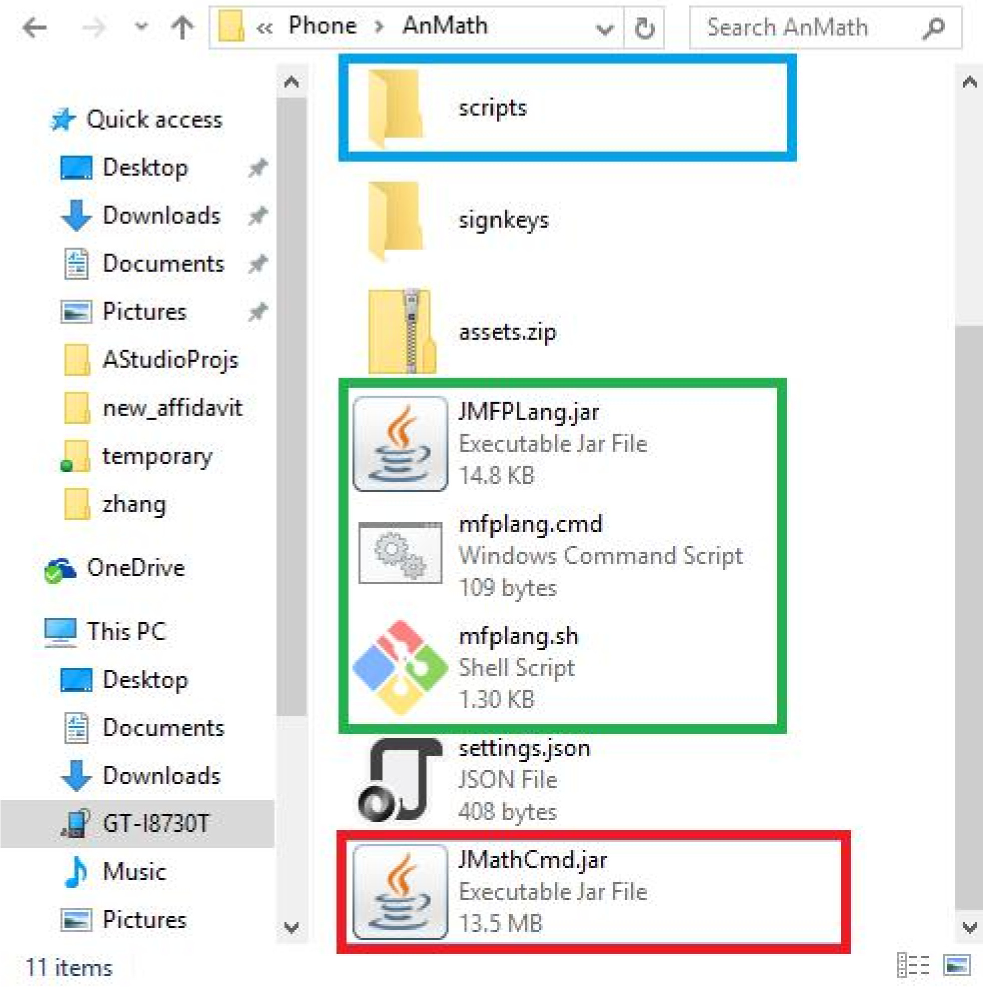


Figure 1.30: User finds JMathCmd.jar, JMFPLang.jar, mfplang.cmd, mfplang.sh and scripts folder in the AnMath directory in the SD card of the Android device.

Some Android devices, e.g. Samsung Galaxy Express, do not allow user to run any executable files from SD card after they are connected to computer. Solution is copying the whole AnMath folder from SD card to a folder physically located in computer, and then running Scientific Calculator Plus for JAVA there. User may create new .mfps scripts and debug and test. After everything is done, user copies the whole AnMath folder back to SD card so that any updates of user defined programs are available in the Android Device.

If user double-clickes the JMathCmd.jar file in the AnMath folder, the starting screen of Scientific Calculator Plus for JAVA is shown as the following graph.

Figure 1.31: Starting screen of GUI based Scientific Calculator Plus for JAVA tool.

GUI based Scientific Calculator Plus for JAVA tool is essentially a command line box. User inputs a command, i.e. one or several math expressions and/or function calls, and then presses ENTER, Scientific Calculator Plus for JAVA will print outputs and returned value in the following lines. In Scientific Calculator Plus for JAVA, a command (or a command batch) may include one or several lines. One line is a math expression, a MFP statement or a function call. However, a command (batch) cannot include any function definition. For example, the following 6 statements comprise a program blog. User can copy and paste them into Scientific Calculator Plus for JAVA and run them as a single command batch:

Variable a

if 3 > 2

a = 10

```
else

a = 9

endif
```

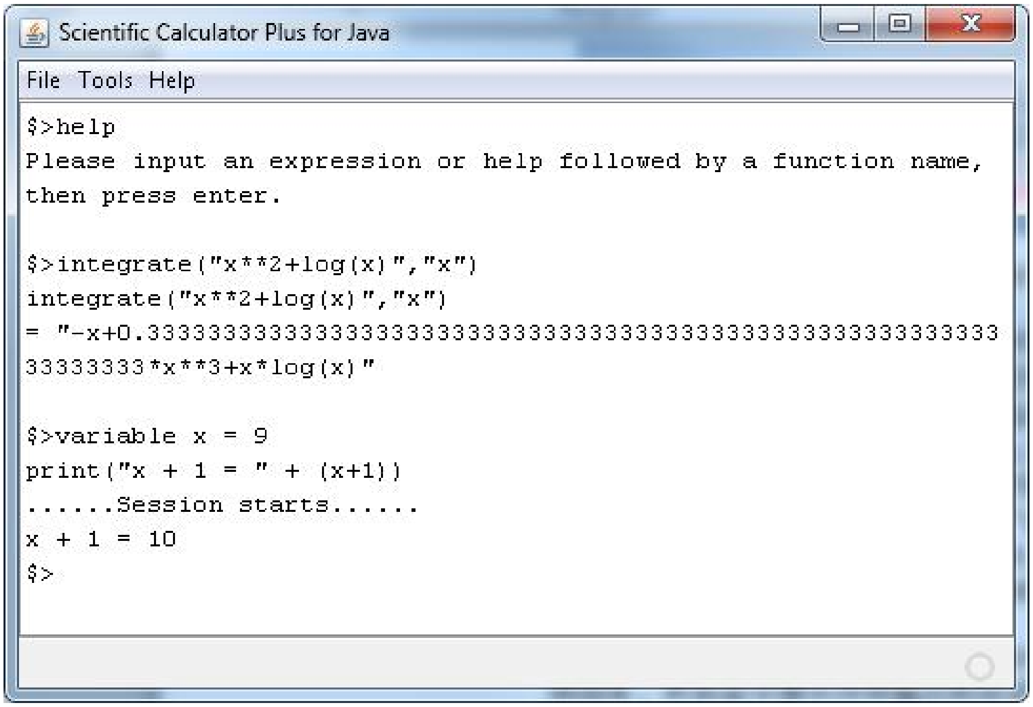
. However, function definition, e.g.

```
Function abcde()

Return 3

Endf

Variable a = abcde()
```

, is not allowed in command batch. If user runs the above four statements in a command batch, an error will be prompted.

If user wants to type a multi-line command batch instead of using copy and paste, Shift-ENTER can be employed to start a new line in the same command batch. Comparatively, simply pressing ENTER key, no matter cursor is at the tail or in the middle of the command (batch), the command (batch) will be executed. This is slightly different from the Command Line tool in an Android device where Shift key may not be supported. In Android, an ENTER, whether with Shift or not, will start a new line in the same command batch. User has to tap the “Start” key in inputpad to execute a command (batch).

The GUI of this Scientific Calculator Plus for JAVA tool is shown as follows.



```
$>help
Please input an expression or help followed by a function name,
then press enter.

$>integrate("x**2+log(x)","x")
integrate("x**2+log(x)","x")
= "-x+0.333333333333333333333333333333333333333333333333333333333
33333333*x**3+x*log(x)"

$>variable x = 9
print("x + 1 = " + (x+1))
......Session starts......
x + 1 = 10
$>
```

Figure 1.32: GUI of Scientific Calculator Plus for JAVA tool.

The system console based Scientific Calculator Plus for JAVA tool works like a Python interpreter. In Windows, user can starts a command line or powershell box, then type

Path\to\mfplang.cmd

after prompt, where path\to\mfplang.cmd is the path to the mfplang.cmd file. Then an interactive interpreting environment starts. In any Unix, MacOSX, Linux or Cygwin machine, the command to start interactive interpreting environment is

Path\to\mfplang.sh

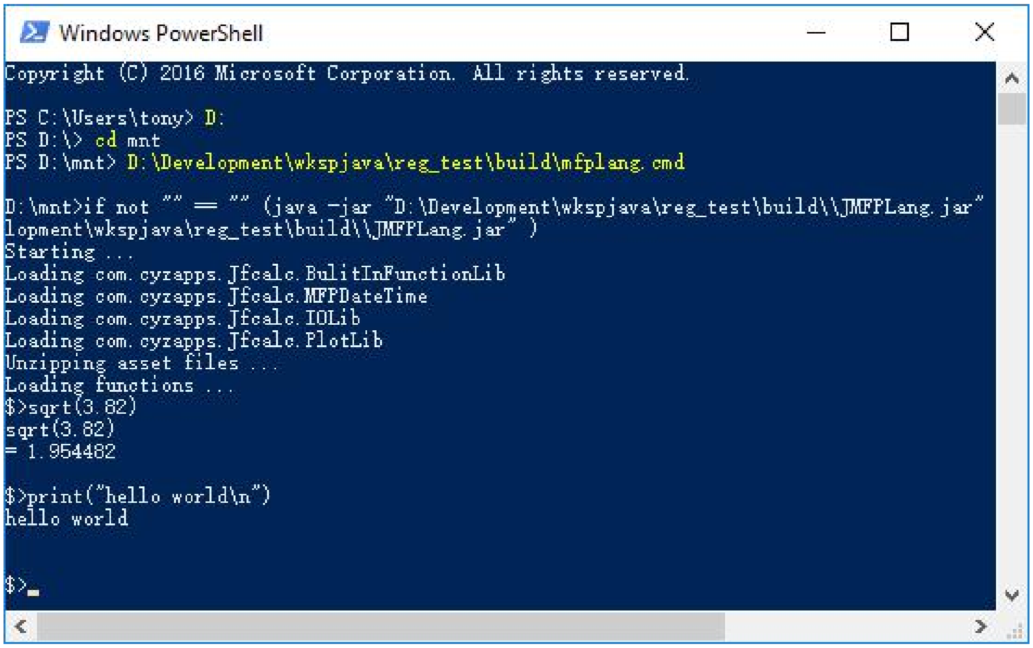
. Note that user may need to set mfplang.sh executable first using command

chmod 777 path\to\mfplang.sh

Also note that system console based Scientific Calculator Plus for JAVA does not support Shift-ENTER key.

If user wants to quit the interactive interpreting environment, simply type quit and ENTER, or press Ctrl-C key to exit.

In windows, the interactive MFP interpreting environment is shown as the following chart:



```
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\tony> D:
PS D:\> cd mnt
PS D:\mnt> D:\Development\wkspjava\reg_test\build\mfplang.cmd

D:\mnt>if not "" == "" (java -jar "D:\Development\wkspjava\reg_test\build\\JMFPLang.jar"
lopment\wkspjava\reg_test\build\\JMFPLang.jar" )
Starting ...
Loading com.cyzapps.Jfcalc.BulitInFunctionLib
Loading com.cyzapps.Jfcalc.MFPDateTime
Loading com.cyzapps.Jfcalc.IOLib
Loading com.cyzapps.Jfcalc.PlotLib
Unzipping asset files ...
Loading functions ...
$>sqrt(3.82)
sqrt(3.82)
= 1.954482

$>print("hello world\n")
hello world


$>_
```

Figure 1.33: Interactive MFP interpreting environment in Windows.

If user only wants to run an mfps script from console, in Windows the command is

Path1/to/mfplang.cmd path2/to/script.mfps param1 param2 …

, in Unix, MacOSX, Linux or cygwin the command is

Path1/to/mfplang.sh path2/to/script.mfps param1 param2 …

, where Path1/to/mfplang.cmd is path to the mfplang.cmd file, pth2/to/script.mfps is path to user's mfps script file, note that the script file name is not necessarily script.mfps, and param1, param2, … are parameters to run the script if there are any. In order to run an mfps script, user must declare @execution_entry in the file above MFP code but below the shebang line and file help. For example, content of an mfps file is

#!/usr/bin/mfplang

# This is file level help info. It must be located below

# shebang line but above @execution_entry annotation.

@execution_entry ::test_cs::test_f (#, #)

Citingspace ::test_cs

Function test_f(a, b)

```
    Return a + b

Endf

EndCs
```

. Assume user is using Windows, the file's name is myscript.mfps and it is located in the current working folder. The AnMath folder is in user's path searching list. Then user only needs to type

```
Mfplang.cmd myscript.mfps 3 4
```

and the result of 7 is shown.

If user is using Unix/Linux/MacOSX/Cygwin, user may manually set a soft link named mfplang in /usr/bin folder linking to the mfplang.sh file. Then user simply types

```
mfplang myscript.mfps 3 4
```

and can also get 7.

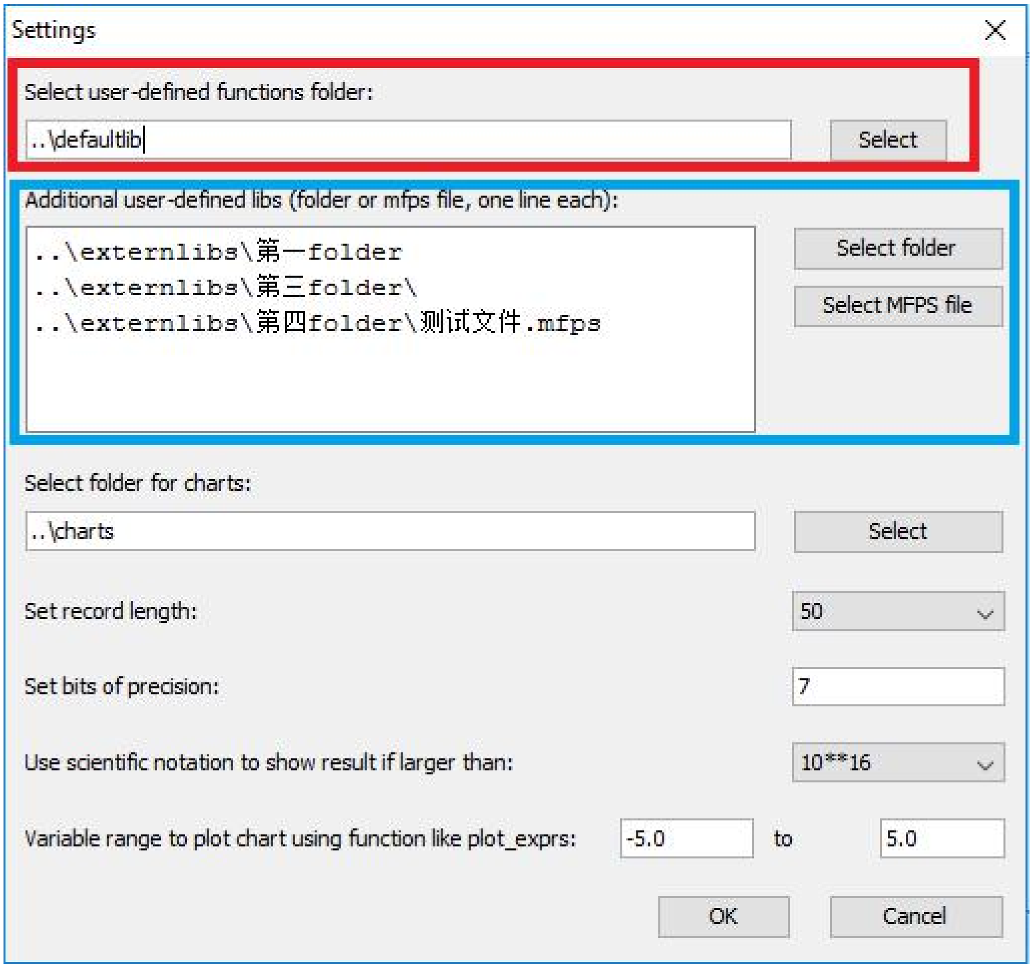
The usage of @execution_entry will be detailed in the following chapter.

The configuration file of JAVA based Scientific Calculator Plus (whether it is the GUI tool or the interactive MFP interpreting environment) is settings.json. This file is saved in the AnMath folder, same folder as mfplang.cmd and mfplang.sh. If this file does not exist, default settings would be used. A sample settings.json file is shown below:

```
{

"CHART_FOLDER_PATH":"..\\charts",

"PLOT_EXPRS_VARIABLE_FROM":"-5.0",

"PLOT_EXPRS_VARIABLE_TO":"5.0",

"ADDITIONAL_USER_LIBS":[

        {"LIB_PATH":"..\\externlibs\\第一 folder"},

        {"LIB_PATH":"..\\externlibs\\第三 folder\\"},

        {"LIB_PATH":"..\\externlibs\\第四 folder\\测试文件.mfps"}

        ],
"SCIENTIFIC_NOTATION_THRESHOLD":"16",

"HISTORICAL_RECORD_LENGTH":"50",

"SCRIPT_FOLDER_PATH":"..\\defaultlib",

"BITS_OF_PRECISION":"7"

}
```

. User may be interested in the fields of SCRIPT_FOLDER_PATH and ADDITIONAL_USER_LIBS. Before version 1.7.1, all the mfps scripts should be saved in AnMath/scripts folder for JAVA based Scientific Calculator Plus. However, from version 1.7.1, the script lib folder path is configurable and user may include more than one script lib paths. Except the first script lib path, which must be a folder, all other script lib paths can be either a file or a folder. This makes MFP more flexible in PC. Moreover, if a script lib path is not absolute, but relative, e.g. "..\\defaultlib" in the above example, it is relative to the AnMath folder, not to user's current working folder.

User is not required to manually create or modify the settings.json file. A straightforward way is to open the GUI based Scientific Calculator Plus for JAVA tool, select "tools" menu and "settings" sub-menu, a config dialog box would pop up. See the chart below. The setting entry in the red rectangle is the first lib path, by default it is scripts folder. The blue rectangle includes additional lib paths, every line is a new lib path, which can be either a folder or an mfps file.



Settings

Select user-defined functions folder: ..\defaultlib

Additional user-defined libs (folder or mfps file, one line each):

```
..\externlibs\第一folder
..\externlibs\第三folder\
..\externlibs\第四folder\测试文件.mfps
```

Select folder for charts: ..\charts

Set record length: 50

Set bits of precision: 7

Use scientific notation to show result if larger than: 10**16

Variable range to plot chart using function like plot_exprs: -5.0 to 5.0

Figure 1.34: Set script lib path(s) for JAVA based Scientific Calculator Plus tool.

Nevertheless, in Android user cannot see the path easily. As such in Scientific Calculator Plus for Android, all the mfps files should be stored in the scripts folder or one of its sub-folders. Otherwise, they will not be loaded at start-up. However, if an mfps script not located in scripts folder includes a correctly defined @execution_entry annotation, it can be executed by long-clicking in the File Manager tool. After it finishes running, its content will be unloaded and in Command Line or Smart Calculator user still cannot see any function defined in this file.

Section 10 MFP App Builder

To extend more convenience to user, Scientific Calculator Plus is able to compile and pack an MFP function (whether it is a built-in or a user-defined function) into an independent Android APK installation file. The APK file can be installed, shared with other people or even published in Android app distributing websites like Google play.

Three steps are required to build an APK package. In the first step, user inputs application name, app package id and version. Note that app package id should be 20 characters long and unique. Otherwise, the app cannot be published. User can also select an icon and type help information for the app. If user selects to use default help information, MFP function help will be shown in the app's help page.

In step 2 user sets up function name and parameters to input. Note that the parameters here may not be exactly the same as function parameters. Parameters here are the input that the APK final user should type in. APK creator can set default values for some of the function parameters to save APK final user's input.

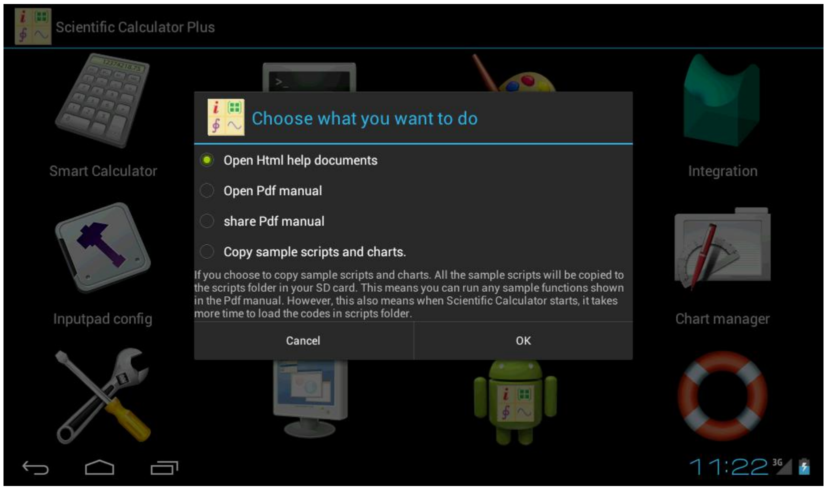
The final step is to set APK file name and determine how to sign the apk. If user selects test key, the generated APK file can be installed but cannot be published. User is able to use an existing public key or create his or her own key. All the created keys are stored in AnMath\signkeys folder.

After the above 3 steps finish, a dialog will be shown and ask user to install or share the app or simply exit. If user selects exit, he or she can still access the generated APK file stored in AnMath\apks folder later on.

MFP App Builder is a very useful tool for user with programming background. So after the chapters for MFP programming language, this manual will provide more detailed information about how to build MFP apps. User with strong interest in this field may refer to Chapter 8.

Section 11 Help and Manual

Scientific Calculator Plus provides a complete html format manual. If using version 1.6.6 or earlier, after launching Scientific Calculator Plus, user taps the help ring icon to read manual. Because it is html based, user can tap the hyperlinks to navigate.



Choose what you want to do

● Open Html help documents

○ Open Pdf manual

○ share Pdf manual

○ Copy sample scripts and charts.

If you choose to copy sample scripts and charts. All the sample scripts will be copied to the scripts folder in your SD card. This means you can run any sample functions shown in the Pdf manual. However, this also means when Scientific Calculator starts, it takes more time to load the codes in scripts folder.

Figure 1.35: Scientific Calculator Plus provides various choices to obtain help information.

From version 1.6.7, Scientific Calculator Plus also provides PDF based manual, i.e. this document. By tapping the help ring icon, user sees a dialog box to select reading html manual, reading or sharing PDF manual, or copying sample codes, as shown in the abve chart.

Besides independent manual, the modules in Scientific Calculator Plus have their own help pages. User can start a module, e.g. Smart Calculator or Integral Calculator, tap Android menu key and select help to read help information.

Moreover, quick help key is available in the inputpad of Smart Calculator and Command Line. User taps the key and then inputs a function name, an operator name or an MFP key word, on-line help for the function (or the operator or the MFP keyword) will be shown after tapping the "Start" key.

Summary

Scientific Calculator Plus is developed for users with any background, even without much knowledge about programming. This chapter introduces the usage of this app without any programming required.

The emphasis of this chapter is, first, how to use Smart Calculator to calculate, recognize math expression and plot graph; second, how to run function or mfps script in Command Line in Android or any other OSes supporting JAVA; third, how to draw 2D/3D/polar graphs using the independent Chart Plotter; fourth, how to calculate derivative, definite and indefinite integrals; fifth, how to setup input pad. Other parts, e.g. managing MFP script files and building MFP apps, are related to programming. User may skip them if not interested in.

The most attractive capability of Scientific Calculator Plus is drawing 3D charts. This chapter demonstrates user how to draw cylinder, ball, cone and some complicated buildings like Shanghai Oriental Pearl TV tower. User may also try to draw a 3D pyramid using Chart Plotter which is a very good practice to learn this software.

Chapter 2 MFP Programming Language Fundamental

As demonstrated in Chapter 1, user without any background of programming can still easily use Scientific Calculator Plus. However, if some simple codes could be written to assemble functions provided by this application, user will be able to achieve more great aims which seems unrealistic to traditional calculators.

Section 1 Introduction of MFP Programming Language

MFP programming language is the built-in programming language and the underlying mathematical engine of Scientific Calculator Plus. It is powerful but only includes around 10 types of statements so that very easy to learn. User is able to develop his/her own functions to call built-in functions and achieve a very complicated aim. In other words, function is the entry point of a user-defined procedure.

Definition of a function includes function name and parameter(s) (if the function has parameter(s)). After a function is coded up, user can open Command Line or Smart Calculator or Scientific Calculator Plus for JAVA and type

function_name(parameter1, parameter2, …, last_parameter)

or if the function does not include any parameter:

function_name()

, and then tap "Start" button in Command Line or Smart Calculator, or press ENTER key in Scientific Calculator for JAVA to execute the function. Please note that, in Smart Calculator, log output from print or printf function will not be shown so that Command Line and Scientific Calculator Plus for JAVA are recommended.

MFP programming language supports functions, variables, condition statements, loop statements, help comments as well as many operators including +, -, * (multiplication), / (division), ** (power to), & (bit and), | (bit or), ^ (xor), = (assignment), == (equal to) etc. Besides, MFP is able to solve mathematical equation(s), handle binary numbers (starting with 0b, e.g. 0b0011100), octal numbers (with 0 as initial, e.g. 0371.242), and hexadecimal numbers (starting with 0x, e.g. 0xAF46BC.0DD3E), process complex numbers, calculate arrays (matrices), and read, write and analyze strings. This language is case-insensitive.

One thing that user needs to keep in mind is, except in comments and strings, all the characters in MFP must be ASCII instead of Unicode. This means Unicode characters like Chinese words or even some Greek letters are not allowed in a function or variable name. Moreover, sometimes a character looks like a single byte ASCII letter but actually is a double byte Unicode character， for example, １ is a Unicode number different from ASCII number 1. If in MFP program １ is used in function or variable name, error will be reported. For this reason, user has to be very careful when copying codes from emails, Word documents or web-pages because they usually include some Unicode characters.

MFP programming language has the following statements:

function, endf, return

variable

if, elseif, else, endif

while, loop, do, until, for, next

break, continue

select, case, default, ends

try, throw, catch, endtry

solve, slvreto

help, endh, @language

citingspace, using citingspace

@compulsory_link

@build_asset

@execution_entry

Each statement in MFP should occupy at least one line. If a statement is too long, it can be divided into several lines, and to the end of each non-last line a string " _" must be appended. For example, assume there is a function with 10 parameters. The declaration of this function is too long to fit in a single line:

```
function abcde(para1, para2, para3, para4, para5, para6, para7, para8, para9, para10)
```

. To make the program reader-friendly, declaration of the function can be divided into several lines like:

```
function abcde(para1, para2, para3, _
        para4, para5, para6, _

        para7, para8, para9, para10)
```

. And we can still add a comment at the end of each line. For the above example, comments can be added like:

```
function abcde(para1, para2, para3, _// line 1 has 3 parameters.

        para4, para5, para6, _                // 3 parameters too.

        para7, para8, para9, para10) // 4 parameters.
```

. And each line can have at most one statement. Not like C/C++, MFP doesn't have any statement divisor, e.g. ";".

MFP also has some annotation keywords, i.e. @language and @compulsory_link. Annotations do not affect running of program. They take effect when generating help information or at compiling time.

Last, MFP source file extension must be .mfps (case insensitive), e.g. test.mfps, myFunc.mFPS etc. If the file extension is not right, Scientific Calculator Plus will not be able to load the source code.

Section 2 Data Types in MFP Programming Language

MFP supports the following data types:

1. Boolean (TRUE or FALSE).

2. Real value. Note that real value can be an integer or a fractional value. User need not worry whether the type of the value is integer or float. MFP will look after it and perform any necessary conversion. MFP is able to handle arbitrarily large value, e.g. 1000^{1000}. However, resolution of MFP is 5*10E-48 which means a value smaller than 5*10E-48 may be, although not necessarily, recognized by MFP as zero.

   As mentioned before, MFP supports decimal numbers, binary numbers, octal numbers and hexadecimal numbers, whether the numbers are integer or float. Binary numbers start with 0b, e.g. 0b0011100; octal numbers start with 0, e.g. 0371.242; and hexadecimal numbers starts with 0x, e.g. 0xAF46BC.0DD3E. In calculation, MFP automatically converts a non-decimal value to decimal, and the calculated result is also decimal. If user wants to convert a decimal value to binary, octal or hexadecimal, functions conv_dec_to_bin (for decimal to binary conversion), conv_dec_to_hex (for decimal to hexadecimal conversion), and conv_dec_to_oct (for decimal to octal conversion) can be used.

Also, a Boolean value can be looked on as a real number. Boolean TRUE equals 1 and FALSE equals 0. On the other hand, if converts a real value to Boolean, 0 will be converted to FALSE and other real values will be mapped to TRUE. Conversion between Boolean and real is carried out automatically by MFP. User need not interfere.

Last, MFP includes a special real value, NAN. This means the value is not defined. For example, result of 1/0 is NAN.

3. Complex value. A complex number is actually a combination of two real values. One of them is real part, the other is image part. A complex number in MFP can be written either as a + b * i or a + bi, where a and b here can be any real value. Note that bi, i.e. no space between b and i, is correct in MFP while b i, i.e. space between b and i, is wrong. If a is 0, the complex number can be simplified as b*i or bi. If b is 0, MFP will convert a + bi to a real value a if necessary. On the other hand, MFP may also automatically convert a real value to a complex value with a zero image part. However, if a complex has a non-zero image part, it cannot be converted to a Boolean value. And like real, there is a special image value, NANi, which means the image value is not defined.

4. Matrix and array. MFP supports matrix calculation. A matrix is made up of a list of elements separated by ",". The boundaries before and after the elements are square bracket, i.e. "[", and close square brackets, i.e. "]". Consider the following three examples, [1, 2, 3+i], [[1, 2, 3+i]] and [[4, 5], [sqrt(8.9), -i], [1.71, stdev(2,3,4)]]. In the first example, [1, 2, 3+i] is a 1-dimentional matrix, i.e. vector, with three elements. The second example, [[1, 2, 3+i]], is a 1*3 matrix. The third example, [[4, 5], [sqrt(8.9), -i], [1.71, stdev(2,3,4)]], is a 3*2 matrix.

   To access elements in matrix, programmer needs to use index. Index is a list of non negative integers surrounded by "[" and "]". For example, a matrix a is [[4, 5], [sqrt(8.9), -i], [1.71, stdev(2,3,4)]], then a[2,0] is 1.71, a[1] is [sqrt(8.9), -i]. Note that index always starts from 0. For another example, assume b is [1, 2, 3+i], then b[2] is 3+i while b[0,2] is invalid.

   MFP supports matrix adding, subtraction, multiplication, division and transposition. For example, [[1, 2], [3, 4], [5, 6]] + [[2, 3], [4, 5], [6, 7]] == [[3, 5], [7, 9], [11, 13]] (matrix adding), [[2, 3], [4, 5]] * 2 == [[4, 6], [8, 10]] (matrix multiplication), [1, 2] * [[3, 4]] == [11] (matrix multiplication), [[5,6],[7,8]]/[[1,2],[3,4]] == [[-1, 2], [-2, 3]] (matrix division) and [[2,3], [4, 5]]' == [[2,4], [3, 5]] (matrix transposition). Note that the size and dimension of the operands in matrix adding and subtraction must exactly match while in matrix multiplication, last dimension size of the first operand should equal first dimension size of the second operand, and only square matrix supports matrix division.

Different from Matlab, MFP does support automatic size/dimension change during matrix element assignment. For example, assume an array variable a is [1, 2], then assigning a new value to a[1] is valid while to a[2] or a[0,1] is invalid. Fortunately MFP has many built-in and predefined functions which are able to access and modify array elements even beyond array size so that user can use them to raise array size/dimension.

5. String. MFP supports string operations. A string is quoted by double quotation marks. However, please keep in mind that double quotation mark has to be a single byte ASCII character, i.e. ". Unicode characters like “ or ” are not allowed. For example, "abc" means a string of abc. Different from C/C++, a string in MFP is not an array of characters. MFP provides a number of functions to process strings.

6. NULL. NULL means a not-existing value. Generally this means the returned value is not expected. NULL cannot be converted to any other data types.

Section 3 Operators Supported by MFP

MFP programming language supports the following operators:

Operator	Definition	Examples
=	Assign operator which assign a value to a variable.	x = 3 + 4 assigns 7 to variable x.
==	Equal sign which means left part has the same value as right part.	x + 1 == 7 + 3 means x + 1 has the same value as 7 + 3 (i.e. 10). Then x can be solved which is 9.
(	Left parenthesis. Expression part inside a pair of parentheses should be calculated first.	x * (y + 3)
)	Right parenthesis. Expression part inside a pair of parentheses should be calculated first.	x * (y + 3)
[	Square bracket which starts an array.	[[1,2,3],[4,5,6]] is a 2*3 matrix whose first line includes elements 1, 2 and 3

		and second line includes elements 4, 5 and 6.
]	Close square bracket which ends an array.	[[1,2,3],[4,5,6]] is a 2*3 matrix whose first line includes elements 1, 2 and 3 and second line includes elements 4, 5 and 6.
,	Comma character which separates individual elements in an array.	[2,3,4,5] has four elements separated by commas.
+	Plus sign, which supports complex number, matrix and string.	2.3 + 4.6 + 2i = 6.9 + 2i [1, 2] + [3, 4] = [3, 6] [1, 2] + ″ hello″ = ″[1,2] hello″
-	Minus sign, which supports complex number and matrix.	4.6 - 2.3 = 2.3 [1, 2] - [3, 4] = [-1, -1]
*	Multiplication sign, which supports complex number and matrix and has higher priority than + (plus sign) and - (minus sign).	1 + 2 * 3i = 1 + 6i [[1, 2]] * [[3], [4]] = [[11]]
/	Division sign, which supports complex number and matrix and has higher priority than + (plus sign) and - (minus sign).	6 - 3 / 2i = 6 + 1.5i [[10,0.5]]/[[1,2],[3,4]] = [[-19.25,9.75]]
\	Left division (mainly for matrix to calculate x from Ax=b (i.e. x = A\b), note that first operand is divisor, do the same operation to division if divisor is not a matrix).	6 - 3 / 2i = 6 + 1.5i [[1,2],[3,4]]\[[11],[32]] = [[10], [0.5]]
&	Bit wise and, which has higher priority than + (plus sign), - (minus sign), *, /. Note that it	4 + 5.2 & 3.7 = 5

	only accept non-negative operands. Moreover, if any of its two operands is not integer, it will be converted to an integer first.	
|	Bit wise or, which has higher priority than + (plus sign), - (minus sign), *, /. Note that it only accept non-negative operands. Moreover, if any of its two operands is not integer, it will be converted to an integer first.	2.8 / 5.2 | 3.7 = 7
^	Bit wise xor, which has higher priority than + (plus sign), - (minus sign), *, /. Note that it only accept non-negative operands. Moreover, if any of its two operands is not integer, it will be converted to an integer first.	14.2 / 5.2 ^ 3.7 = 2.3667
**	Power, which has higher priority than + (plus sign), - (minus sign), *, /, &, |, ^. Note that both the first and the second operands can be complex number.	2 * 4 ** 3 = 128 (-4)**0.5 = 2i 3**(2+i) = 4.0935 +8.0152*i
+	Positive sign, which has higher priority than all the binary operators, e.g. + (plus sign), *, ^ and **. Note that positive sign supports	4 - +3 = 1 +[4,3] = [4, 3]

	arrays.	
-	Negative sign, which has higher priority than all the binary operators, e.g. + (plus sign), *, ^ and **. Note that negative sign supports arrays.	4 ** -1 = 0.25 -1 ** 4 = 1 -[3.71i, [4, 5.33-6i]] = [-3.71*i,[-4, -5.33+6*i]]
%	Percentage, which has higher priority than all the binary operators and + (positive sign) and - (negative sign). Note that this operator does not support complex number.	-401.78% = -4.0178 5.77% = 0.0577
!	Not sign, which has higher priority than all the binary operators and + (positive sign) and - (negative sign). If its operand is non-zero, it returns false, otherwise, it returns true. Note that this operator does not support complex number.	!-0.2 = False
~	Bit wise not, which has higher priority than all the binary operators and + (positive sign), - (negative sign) and ! (not sign). Not that it only accepts non-negative operand. Moreover, if the operand is not integer, it will be converted to an integer first.	~0 = -1 ~0.1 = -1 ~9 = -10
!	Factorial sign, which has	7! = 5040

	higher priority than all the binary operators and left unary operators. Not that it only accepts non-negative operand. Moreover, if the operand is not integer, it will be converted to an integer first.	
’	1D or 2D matrix or single value transpose, which has higher priority than all the binary operators and left unary operators. Note that it converts a 1D matrix to 2D matrix but returns the original value of single value operand.	(3 + 4i)’ = (3 + 4i) [1, 2+3i]’ = [[1], [2+3i]] [[1, 2],[3,4]] = [[1,3], [2, 4]]

Section 4 Function, return and endf Statements

Function statement in MFP is the start of a function. Its usage is:

Function name(parameter1, parameter2, parameter3, ...)

. Note that "..." here means there are various number of parameters following parameter 3. If there is no optional parameter, "..." shouldn’t be used. For example:

Function abcd(para1, para2, para3, para4)

is a function named abcd with four parameters, while

Function abcdx(para1, para2, para3, para4, ...)

is a function with >= 4 parameters. The number of optional parameters (excluding para1 to para4) is stored in a system variable named opt_argc. The first to fourth parameters have defined name which are para1, para2, para3 and para4 respectively. The first optional parameter's value is stored in opt_argv[0] where opt_argv is another system array variable. Similarly, the second optional parameter's value is opt_argv[1], the third is opt_argv[2], etc.

A function may return a value or return nothing. However, different from Matlab, user needs not to declare the return variable name in function statement.

Return statement in MFP is called to exit from a function and return a value (which can be any data type) to caller if needed. For example:

Return "Hello word"   // return a string "Hello word"

or

Return   // return nothing

.

Endf statement is the end of function block. It does not take any parameter.

The following example prints a "hello world" string and returns 1. Because print function is called, Command Line or JAVA based Scientific Calculator Plus are recommended to run it. If it runs in Smart Calculator, no string will be printed. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder (AnMath/scripts/manual folder in SD card).

```
Help
@language:
  my_name is user's name input when running the function.
@end
@language:simplified_chinese
    my_name 是用户在运行本函数时输入自己的名字
@end
endh
function Helloword(my_name)

  //Although in the code no unicode char (e.g. Chinese characters)
  //is allowed, but in strings and comments unicode chars are
  //allowed. Also note that " must be single byte Ascii char
  //instead of unicode ” or “.
  print("Hello world!(Chinese:世界你好！) my name is " + my_name)

 return 1

endf
```

If user types ::mfpexample::Helloword("Bob") in Command Line or Scientific Calculator Plus for JAVA and then runs, the output would be

```
Hello world!(Chinese:世界你好！) my name is Bob
```

.

Section 5 variable Statement

Variable statement defines one or several variables. Its syntax should be always like

variable var1{=expr1}, var2{=expr2}, var3{=expr3}, ..., varN{=exprN}

, where var1, var2, var3, ..., varN are variables' names. N can be any positive integer. And {=expr*} means the initial value assignment can be neglected (with NULL as default initial value). For example, statement

variable a = "hello, world", b, c = a + 7, d=[2,3,[5,8]]

declares four variables whose names are a, b, c, and d and whose initial values are "hello, world", NULL, "hello, world7" and [2, 3, [5, 8]] respectively. In particular, c's initial value is obtained by appending character 7 to the tail of string a so that it is "hello, world7".

The following example defines two variables a and b and prints their values. As print function is used, Smart Calculator is not recommended to run the example. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder (AnMath/scripts/manual folder in SD card).

```
function DeclareVariable()

	variable a = [1,2,3], b

	b = 4.75-0.67i

	print("a = " + a + "; b = " + b + "\n") //\n means starting a new line.

endf
```

User types ::mfpexample::DeclareVariable() in Command Line or Scientific Calculator Plus for JAVA and runs it, then the output would be:

```
a = [1, 2, 3]; b = 4.75 - 0.67i
```

.

Section 6 if, elseif, else and endif Statements

If, elseif, else and endif comprise conditional block in MFP. They should be used like the following example:

```
If conditionA

    // if conditionA is satisfied, MFP executes statements in this

    // block until it sees an elseif or else, then it jumps to the

    // statement following endif. Otherwise, it jumps to the next

    // conditional statement (i.e. elseif or else) or the

    // statement following endif if no elseif or else available.

 ......

elseif conditionB

    // if conditionB is satisfied, MFP executes statements in this

    // block until it sees an elseif or else, then it jumps to the

    // statement following endif. Otherwise, it jumps to the next

    // conditional statement (i.e. elseif or else) or the

    // statement following endif if no elseif or else available.

 ......

elseif conditionC

    // if conditionC is satisfied, MFP executes statements in this

    // block until it sees an elseif or else, then it jumps to the

    // statement following endif. Otherwise, it jumps to the next

    // conditional statement (i.e. elseif or else) or the

    // statement following endif if no elseif or else available.
```

```
   ......
......
......
......
else
      // if none of the above conditions is satisfied, MFP executes
      // statements in this block.
   ......
endif
```

, where conditionA, conditionB, conditionC, etc. are expressions having Boolean values. If they are not Boolean typed, MFP tries to convert them to Boolean. If a condition expression (e.g. a string) cannot be converted to Boolelan, an exception is thrown.

The following example uses the above conditional statements to determine the value range of user's input. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder (AnMath/scripts/manual folder in SD card).

```
Help
@language:
 a input by user when this function called. It should be a real value.
@end
@language:simplified_chinese
  a 是一个实数值，由用户作为调用函数时的参数输入
@end
Endh
function IfStatement(a) //a must be a real value.
        if a < 0
                print("a < 0!\n")
        elseif a < 1
                print("0 <= a < 1!\n")
        elseif a < 10
                print("1 <= a < 10!\n")
        else
                print("a >= 10!\n")
        endif
endf
```

If user inputs ::mfpexample::IfStatement(-1) to run, the output would be:

a < 0!

, if user inputs ::mfpexample::IfStatement(0.5) to run, the output would be:

0 <= a < 1!

, if user inputs ::mfpexample::IfStatement(1) to run, the output would be:

1 <= a < 10!

, if user inputs ::mfpexample::IfStatement(20) to run, the output would be:

a >= 10!

.

Section 7 while, loop; do, until and for, next Statements

While and loop, do and until, and for and next are three pairs of looping statements in MFP. They should be used in the following way:

```
While condition // condition must be Boolean or can be converted
          // to Boolean. If condition is TRUE, MFP enters the
          // the while loop.
   ......
```

```
Loop

Do

    ......

Until condition // condition must be Boolean or can be converted

                // to Boolean. If condition is FALSE, MFP jumps

                // back to the statement after do. If it is TRUE,

                // MFP continues executing the statement after

                // until.

For variable var = from_value to to_value step step_value

    // the variable statement is not needed if var has been

    // declared before

    ......

Next
```

, where condition is an expression which has boolean value (or can be converted to boolean value). Var is the name of for statement's index variable. From_value is the initial value of var, step_value is how much var increases each time. Note that step_value can be negative. To_value is the destination value of var variable. Var variable stops to change if it is beyond to_value. Note that if var is defined before, the variable keyword following for can be ignored. The following code is an example of for block. Note that when for statment's index variable's value equals to_value, statments inside for block are still executed. Only if index variable's value is beyond to_value the for loop finishes.

```
variable idx

for idx = 1 to -1 step -2

    print_line("idx == " + idx)

next
```

By running the above code, the output is

```
idx == 1

idx == -1
```

Note that all the three looping blocks support break and continue statements. When break is hit, MFP jumps out the current inner-most looping block. When continue is hit, MFP ignores the following statements in the inner-most looping block and goes back to the beginning of the looping block.

The following example uses the above looping statements as well as break and continue. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual’s sample code folder (AnMath/scripts/manual folder in SD card).

```
function Testloops(a)

 if a == 0

  // a==0 we test for, break and continue statements

  for variable idx = 0 to -8 step -1

   if idx > -4

    continue

   elseif idx < -6

    break

   else

    print("idx = " + idx + "\n")

   endif

  next

 elseif a == 1

  // a == 1 we test while and do statements

  variable idx = 4

  while idx < 8

   variable idx1 = idx

   do

    idx1 = idx1 + 2

   until idx1 > 11

   idx = idx + idx1 /5
  loop
        //print the idx value
  print("after while and do, idx value is " + idx + "\n")
 endif
endf
```

If user inputs ::mfpexample::testloops(0) to run, the output is:

```
idx = -4

idx = -5

idx = -6
```

, if user’s input is ::mfpexample::testloops(1)， the output would be:

```
after while and do, idx value is 8.88
```

.

Section 8 break and continue Statements

Break statement is used in looping blocks (i.e. while ... loop, do ... until, for ... next) or select ... case ... default ... ends conditional block. MFP jumps out the inner-most looping block or select ... case ... default ... ends conditional block if break is hit.

Continue statement is only used in looping blocks (i.e. while ... loop, do ... until, for ... next). MFP ignores the following statement and goes back to the beginning of the inner-most looping block if continue statement is hit.

Example of break and continue has been listed in Section 7 .

Section 9 select, case, default and ends Statements

Select, case, default and ends comprise another type of conditional block besides if. They should be used like the following example:

```
// expr can be a value or an expression resulting a value

select expr

case value1

    // if expr’s value is value1, continue running the following
```

```
        // statements until a break is hit.
    ......
    break // jump to the statement after ends if break is hit.
case value2
        // if expr's value is value2, continue running the following
        // statements until a break is hit.
    ......
    break // jump to the statement after ends if break is hit.
......
......
......
default
        // if expr does not equal to any of the above values, run the
        // following statements.
    ......
ends // end of conditional block.
```

. Note that if there is no break at the end of a case block, MFP will continue to execute the statements in the following case or default block.

The following example uses select, case and default statements to check user's input. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder.

```
Help
@language:
 a input by user when this function called. It should be a real value.
@end
@language:simplified_chinese
  a 是一个实数值，由用户作为调用函数时的参数输入
```

```
@end

endh

function TestSelect(a)

 select a

 case 1

   print("a is 1\n")

   break

 case 2

   // since no break after this statement, it will continue to

   // case 3 until see a break or ends.

   print("a is 2\n")

 case 3

   print("a is 3\n")

   break

 default

   print("a is other value\n")

 ends

endf
```

Run the above example. If user inputs ::mfpexample::TestSelect(1), the ouput would be

```
a is 1
```

; if user's input is ::mfpexample::TestSelect(2), since case 2 does not include a break statement, the output would be

```
a is 2

a is 3
```

; if input is ::mfpexample::TestSelect(3), output would be

```
a is 3
```

; if input is ::mfpexample::TestSelect(4), default block runs so that output is

a is other value

.

Section 10 try, throw, catch and endtry Statements

Try and try related statements detect and process run-time exceptions. The concept of run-time exception may be too advanced for beginners. Since conditional statements plus printing statements can also be employed to detect and handle most of errors, users may skip this section.

In MFP programming language, exception is error happening at run time, e.g. undefined variable, or cannot convert value to a particular type. Any exception triggered by MFP in the try block will be thrown to the following catch statements until one of the catch statements processes it. If no catch statement following this try block is able to process the thrown exception, or the exception is not triggered inside a try block, it is thrown to the higher level.

Throw statement has a string parameter. The running MFP program exits if a throw statement is hit, but the exception cannot be handled by a catch block. And the string following throw keyword is printed.

Catch statement may have an expression as parameter. If it does not take any parameter, it catches any exception. If it is followed by an expression, the expression is an exception filter which identifies an exception should be processed by this catch block or not. If exception filter value is true, the exception is caught. Otherwise, the exception is conveyed to the next catch statement, or thrown to upper level. Catch statement provides three string typed internal parameters, i.e. level, type and info. Parameter level is exception level, its value can be either "LANGUAGE" (i.e. language level exception like no endif statement follows an if statement or a user defined expression to throw a string) or "EXPRESSION" (i.e. expression level exception like divided by zero or lack of right parenthesis). Parameter type is the exception type internally defined in MFP language. Parameter info is the information provided by the exception. If developer uses a throw statement to throw a string, info value is the string. These three parameters can only be used in exception filter in a catch statement. However, exception filter can use any variables defined in the name scope. If a variable has the same name as any of the three internal parameters, it's overridden by the internal parameter.

Endtry statement finishes a try/catch block. It does not take any parameter.

Follows are an example for try/throw/catch/endtry. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder.

```
Help
 test try ... catch statement
endh
function testTryCatch()
 Variable a, b, c
 a = 3
 Try
  Select a
  Case 3
   print("a == 3\n")
   Try
    dbc = a + 4
   Catch  // catch all the exceptions
    print ("dbc is undefined\n")  //undeclared variable dbc
   EndTry
  EndS
  Throw "my exception"  //throw an exception by user
 Catch (1+2)==4
  // here (1+2)==4 is an exception filter. If an exception
  // satisfies (1+2)==4 it is caught here.

  //will never be here because 1+2 never equals 4.
  print ("Exception satisfying (1+2) == 4 is caught")
 catch false
  // here false is another exception filter. Clearly,
  // no exception will be caught.
  print ("Exception satisfying false is caught")
 Catch and((b=level)=="LANGUAGE", (c=info) == "my exception")
  // here the exception filter has two conditions.
  // exception level should be "LANGUAGE" and info should
  // be "my exception". we can define many different
  // exception filters using logic functions like and, or,
  // and exception level and info.

  // we cannot use level and info directly because they
  // are valid only in a catch statement. However, we can
  // assign their values to some variables and access the
  // variables later on.
  print ("Exception caught, level = " + b + ", info is " + c)
  print ("\n")
  Try
   // Unlike other languages, note that divided by 0 will
   // not cause an exception because MFP supports INF
   // (infinite) and Nan.
   c = 3/0
  Catch
   print ("Divided by zero!\n")  // will not be here
  EndTry
 Endtry
Endf
```

If user inputs ::mfpexample::testTryCatch() and runs the program, the output would be:

```
a == 3

dbc is undefined
```

```
Exception caught, level = LANGUAGE, info is my exception
```

.

Section 11 solve and slvreto Statements

Solve statement starts an in-line solver. It takes an arbitrary number of variables as parameters. These variables must be declared before the solver. These variables will be the to-be-solved variables in the solve block. In the following example, user first defines three variables x, y and z, then starts a solve block to solve the values of variables x, y and z.

```
variable x = 3, y, z = [2, 7]

solve x, y, z

...
```

It doesn't matter what the initial values of x, y and z are. If in the solve block x, y or z is solved it will be assigned a new value. If the solve block cannot solve any or all of the variables, the value of the variable(s) does not change.

Slvreto statement finishes an in-line solver. It has one optional parameter, i.e. a variable declared before the solve block. This variable stores all the roots of each to-be-solved variables (i.e. variables as parameters in the corresponding solve statement). However, to retrieve roots of a variable or a result set, user should use built-in functions including get_num_of_results_sets, get_solved_results_set and get_variable_results. For example, both of the following statements are valid:

```
slvreto

slvreto all_results
```

The follows are a solver example. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder.

```
function testSolve()

 Variable a, b, c, x, y, z

 a = 3

 b = 4

 c = 5

 x = 6

 y = 7
```

```
z = 8
// x, y and z are unknown variables to be solved.
// a, b and c are also used in solve block. However,
// they are not unknown so that their values won't
// change after solve block.
solve x, y, z
  // Note that in the following equations
        // == must be used instead of =.
  a * x**2 + 7 * log(b) *x + 6.5 == 8
  y * b - z + 6 == 3.7 + x/(a + 7)
  y * x + z/(c - 3) == 6 + a + y
slvreto a  // a, which is negligible, is used to store
       // all the roots of all variables.
print("\nx == " + x + "\ny == " + y + "\nz == " + z)
print("\nnumber of result sets is ")
// print number of result sets
print(get_num_of_results_sets(a))

// It is possible that solve block cannot solve the
// equations. If so, slvreto returns an empty value.
// Note that can only use system provided function
// get_num_of_results_sets to determine number of
// result sets because data structure of the returned
//  value may change in a future edition.
if (get_num_of_results_sets(a) > 0)
  // 0 means first result set, 1 is 2nd result set, ...
  print("\nThe second result set is ")
```

```
    print(get_solved_results_set(a, 1))
    // Note that function get_variable_results has two
    // parameters. First is solve block's return which
    // includes all result sets. Second tells the function
    // which unknown variable to return: 0 means the first
    // unknown variable, 1 means the second, etc. It is
    // possible that some unknown variables can be solved
    // while others cannot. In this case value of unknown
    // which cannot be solved is NULL.
    // Now print all roots of y
    print("\nAll roots of y are ")
    print(get_variable_results(a, 1))
  else
    // cannot solve
    print("\nSorry, cannot solve x, y and z")
  endif
  return
Endf
```

Run the above example user can get

```
x == 0.14781939
y == 6.84549421
z == 29.66719489
number of result sets is 2
The second result set is [-3.38250623, -3.22386342, -10.25720306]
All roots of y are [6.84549421, -3.22386342]
```

. To write a solve block, user needs to know the following things:

1. There are two types of variables in a solve block, normal variables and to-be-solved variables. To-be-solved variables are the parameter variables in the solve statement definition. In the above example, to-be-solved variables are x, y and z while a, b and c are normal variables. Values of normal variables are known to solve block while values of to-be-solved variables will be assigned in solve block. Also, both normal variables and to-be-solved variables should be declared before solve block.

2. To set up a to-be-solved expression in a solve block, '==' should be used instead of '=' since '=' assigns value to a variable. However, not recommended though, it is fine if user assigns value to a variable (whether normal or to-be-solved) in a solve block. An example is:

```
variable a = 3, b = 4, c = 5, x, y

solve x, y

        a * x + y / c == 9

        c = 7

        y * b - x * c  == 6

slvreto
```

In the above example, if we comment the line c = 7, we actually solve an expression group which includes 3 * x + y / 5 == 9 and y * 4 - x * 5 == 6. But if we have the line c = 7, we actually solve an expression group which includes 3 * x + y / 5 == 9 and y * 4 - x * 7 == 6.

If user assigns value to a to-be-solved variable, this to-be-solved variable is no longer unknown so that it becomes a normal variable. However, user has to be very careful because assigning may affect expressions before it. An example is:

```
variable a = 3, b = 4, c = 5, x, y

solve x, y

        a * x + y / c == 9

        x = 7

        y * b - x * c  == 6

slvreto
```

In the above example, if we comment the line x = 7, we actually solve an expression group which includes 3 * x + y / 5 == 9 and y * 4 - x * 5 == 6. But if we have the line x = 7, we actually solve a single expression which is y * 4 - 7 * 5 == 6. This is because, clearly, MFP cannot solve x and y values only from the first expression a * x + y / c == 9 so it goes to the next statement. In the second statement, x is assigned a value, which is 7, and becomes a normal variable. In the third statement, variables b, c and x are normal variables with valid value so that MFP uses their values and gets that y should be 10.25. Then MFP goes back to the first expression a * x + y / c == 9. But now both x and y become normal variables and have known values so that MFP uses their values and simplifies the expression to 3 * 7 + 10.25 / 5 == 9. This is a comparing expression and its value is FALSE. In this way, MFP solves x should be 7 and y should be 10.25.

3. As described above, all the roots of all the variables are stored in the returned variable in slvreto statement. In current MFP language, the returned variable is actually a 2-D matrix, each row is a root set for every variable. However, please note that in the future data structure of solver returned value may change. As such assure to use functions provided by system (i.e. get_num_of_results_sets, get_solved_results_set, get_variable_results, etc.) to obtain the roots. After solve block finishes, the value of each to-be-solved variable is the root of this variable stored in the first root set. Only exception is that in the first root set this variable cannot be solved.

MFP in-line solver is not almighty. It is possible that MFP cannot solve some mathematical expressions. If so, slvreto returns an empty result to its returned variable. Function get_num_of_results_sets returns zero. The to-be-solved variables keep their original values. It is also possible that some to-be-solved variables are solved while others cannot be solved. In this case, the unsolved to-be-solved variables keep their original values. If we use get_solved_results_set or get_variable_results function to obtain the unsolved variable values, we get NULL.

4. The number of results sets returned by solve block is determined by multiplying number of roots of each individual to-be-solved variable. For example,

```
variable x, y, z

solve x, y

        log(x) == 3

        y**3 + 3 * y**2 + 3 * y+ 1 == 0

slvreto z
```

returns three sets of results. All of them are [20.08553692318766792368478490971028804779052734375, -1]. This is because y**3 + 3 * y**2 + 3 * y+ 1 == 0 has three roots but with same value. Although log(x) == 3 has only one root, each y root should has a corresponding x value so that we get three sets of results.

Section 12 help and endh Statements and @language Annotation

Help is the start of a help block in MFP language and endh is the end of help block.

Note that although the statements in a help block does not have any impact in the running of a function, it shows user help information when user types "help function_name" or "help function_name(number_of_parameters_in_this_function)" if this help block is right above the function declaration. MFP is also able to feed different help information for different system language. For example (This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder), assume a help block is right above function testHelp:

```
Help
This line will be shown for any system language.
@language:
This line will be shown for default system language.
@end
   This line will also be shown for any system language.
@language:simplified_chinese
这一行将在系统语言为中文时显示（This line will be shown when system language is simplified Chinese.）。
@end
This line is also a line for any system language.
Endh
function testHelp(x, y)
Endf
```

. If user inputs help ::mfpexample::testHelp or help ::mfpexample::testHelp(2), and if system language is English and help default language is also English, the following help information will be shown:

This line will be shown for any system language.

This line will be shown for default system language.

This line will also be shown for any system language.

This line is also a line for any system language.

. If system language is Japanese but Japanese is not help default language, user will see:

This line will be shown for any system language.

This line will also be shown for any system language.

This line is also a line for any system language.

. If system language is Simplified Chinese but Simplified Chinese is not help default language, user will see:

This line will be shown for any system language.

This line will also be shown for any system language.

这一行将在系统语言为中文时显示（This line will be shown when system language is simplified Chinese.）。

This line is also a line for any system language.

. If Simplified Chinese is both system language and help default language, user will see:

This line will be shown for any system language.

This line will be shown for default system language.

This line will also be shown for any system language.

这一行将在系统语言为中文时显示（This line will be shown when system language is simplified Chinese.）。

This line is also a line for any system language.

.

Similar to C/C++/JAVA, if user simply wants to append comment to the end of code line, “//” can be used. Content after “//” is comment. “//” can also start a line. If a line is started by “//”, the whole line is a comment.

Section 13 citingspace and using citingspace Statements

User may have noticed that, from v. 1.7, MFP programming is slightly different from before. For example, in old user manual(s), sample function names always start from mfpExample_. This avoids name conflict between sample functions and user defined functions.

From v. 1.7, sample function names do not start with mfpExample_. In the above example, testing function name is as simple as testHelp. Therefore, How the MFP programming language identifies different testHelp functions if user also defines a function with the same name, and accidentally it has the same number of parameters as the sample function?

A good approach is using different citingspaces for these functions. MFP believes they are the same if, and only if, two functions are within the same citingspace, share the same name and have the same number of parameters. For example, starting from this version, all the sample functions are defined in citingspace ::mfpexample. To this end, citingspace ::mfpexample is added to the head of all the sample source files, and endcs clause is appended to the tail. This means, all the sample functions included in this manual are defined / implemented in citingspace ::mfpexample. There would be no name conflict even if other citingspaces include same name functions.

Citingspace is a concept of hierarchy. The name of the top level citingspace is an empty string. If in an mfps file no citingspace is declared, MFP believes all the defined functions in this source file are located in the top level citingspace.

Inside the top level citingspace is where lower level citingspaces are defined. MFPExample is a typical example of lower level citingspace. Note that citingspace name, like function name, is case insensitive so that MFPExample is the same as mfpexample or MfpExample.

Usually user has to use its absolute citingspace path to locate a particular citingspace. Construction of absolute citingspace path is:

top_level_citingspace_name::lower_level_citingspace_name::further_lower_level_citingspace_name::……::target_citingspace_name

. In the above example, citingspace mfpexample is one level lower than top level citingspace so that its absolute path is ::mfpexample. The top level citingspace name is an empty string so that nothing is before the double colons (::).

When user runs an MFP sample function, the citingspace has to be explicitly included. For example, the following function has been defined in the citingspace ::mfpexample:

```
Function testCS1(a)

            print("This is ::mfpexample::testCS1, a=" + a + "\n")

endf
```

. If user wants to call the above function, the command should be ::mfpexample::testCS1(a). In other words, it is absolute citingspace path followed by double colons (::) plus function name and parameters.

The question is, How does MFP keep backward compatible with old source codes developed when citingspace was unavailable?

The solution is using default citingspaces. If a function is defined inside in a default citingspace, or a function's citingspace is a child/grand-child citingspace of a default citingspace, absolute citingspace path is not required in the calling statement.

In version 1.7, MFP's default citingspaces are the top level citingspace, mfp citingspace (::mfp) and all the sub citingspaces below mfp. Any function defined in these citingspaces needs no citingspace path when being called.

In version 1.6.7 or older, no mfps source file includes citingspace declaration. In version 1.7, all the functions in these mfps files are assumed to be located in the top level citingspace. Because top level citingspace is an MFP default citingspace, no citingspace path is required when a function is called.

Assume, for example, user defined a function named Myf_abc() in Scientific Calculator Plus 1.6.7, and called this function in some other functions. Clearly, the calling statement is my_abc(). In the new Scientific Calculator Plus (version 1.7 or newer), user can still call it in this way because my_abc() is defined in the top level citingspace. Nonetheless, including citingspace path, i.e. ::my_abc() is also valid.

The built-in and predefined functions in the new Scientific Calculator Plus are placed in a child (or grandchild) citingspace of ::mfp. For example, function tan(x) in MFP version 1.7 is located at ::mfp::math::trigon, and function log(x) is placed in ::mfp::math::log_exp. Because all these functions are in an MFP default citingspace, no citingspace path is required when calling any of these functions. Again, this feature keeps backward compatibility.

From version 2.0, MFP starts to support Object-Oriented Programming. class statement has been introduced since then. class is compatible with citingspace. Actually a class is a citingspace. All the static functions inside a class can be accessed by their citingspace paths. However, non-static member functions cannot be accessed by this means as they need an object of the class to be referred to. Also, please note that at this moment static member variables are not supported so that all the member variables cannot be accessed by citingspace path. The following code is an example.

```
class ABC
Function f0()
    print("\nThis is ::ABC::f0()\n")
endf
Function f1(self)
    print("\nThis is ::ABC::f1(self)\n")
Endf
endcs
```

In the above code, it is valid to call f0() using ::ABC::f0(). But developer cannot call f1 in this way as function f1 needs object of class ABC to be referred to.

Besides default citingspaces, user is able to insert additional citingspaces into the citingspace searching list. To this end, using citingspace statement is required. For example, using citingspace ::abcd tells MFP citingspace ::abcd is also in the searching list besides top level citingspace, ::mfp and its child citingspaces.

Both citingspace statement and using citingspace statement can use relative citingspace. For example, in citingspace ::mfpexample, user defines another citingspace abcd as below:

```
citingspace Abcd
Function testCS1(a)
    print("\nThis is ::mfpexample::abcd::testCS1, a=" + a + "\n")
endf
Endcs
```

. Here Abcd is a relative citingspace path. Because citingspace abcd is defined in citingspace ::mfpexample, its absolute citingspace path is ::mfpexample::abcd.

For another example, assume user calls function abcd::testCS1(a) in a function located in ::mfpexample. Also assume no using citingspace statement is before the calling statement. Then the relative citingspace used in the calling statement equals to citingspace ::mfpexample::abcd, and the whole calling statement can be converted to ::mfpexample::abcd::testCS(a).

If user defines a several level citingspace hierarchy, and declares a few used citingspaces (with using citingspace statements) in each citingspace structure, then the citingspace searching order should be:

1. The direct (i.e. Innermost) citingspace of a function call;

2. Used citingspaces (by calling using citingspace statement) in the innermost code block of the function call. Note that the using citingspace statements have to be above the function call. Closer to the function call, higher searching priority the used citingspace has;

3. Used citingspaces (by calling using citingspace statement) in the upper code block(s) of the function call. Clearly, used citingspaces in an outer code block has lower searching priority than used spaces in an inner code block. Moreover, the using citingspace statements have to be above the function call. And closer to the function call, higher searching priority the used citingspace has;

4. Used citingspaces inside the direct citingspace of the calling function but out of the calling function body. Similar to the above points, the using citingspace statements have to be above the function call. And closer to the function call, higher searching priority the used citingspace has;

5. Non-direct citingspaces of the calling function (i.e. The citingspaces until the top level citingspace that include the innermost citingspace) and used citingspaces in each non-direct citingspace. Note that the priority in detail is innermost non-direct citingspace (i.e. The citingspace right above the direct citingspace) > used citingspaces in this citingspace > upper level citingspace > used citingspaces in the upper level citingspace ……. For example, assume a citingspace hierarchy in an mfps file is:

```
Citingspace level1 //absolute path is ::level1

Using citingspace aaaa //absolute path is ::level1::aaaa

Using citingspace bbbb //absolute path is ::level1::bbbb

Citingspace level2 //absolute path is ::level1::level2

Using citingspace cccc //absolute path is ::level1::level2::cccc

Using citingspace dddd //absolute path is ::level1::level2::dddd

Citingspace level3 //absolute path is ::level1::level2::level3

Function asmallfunc()

endf

Endcs

Endcs
```

```
Endcs
```

. Also assume the mfps file only includes the above code. Then from function asmallfunc's perspective, the searching order of the citingspaces is:

::level1::level2::level3 > ::level1::level2 >

::level1::level2::dddd > ::level1::level2::cccc > ::level1 >

::level1::bbbb > ::level2::aaaa > top level citingspace.

6. MFP's default citingspaces.

User has to keep in mind a few things. First of all, if using citingspace statement with a low priority citingspace is called well above a function call, from the function call's perspective, the priority of the citingspace is enhanced. For example, a function is defined in citingspace ::aaaa. This function is going to call another function. Before the function call, a using citingspace statement introduces ::bbbb in the citingspace searching list. The sourcecode would be like:

```
Citingspace ::aaaa

……

Function callsomething()

……

Using citingspace ::bbbb

Anotherfunc() // here function callsomething calls Anotherfunc()

……

Endf

……

Endcs
```

.Clearly, in the above example, when Anotherfunc() is called, the citingspace searching order is (from highest to lowest):

1. ::aaaa

2. ::bbbb

3. Top level citingspace

4. Citingspace ::mfp and its children

. If user adds another using citingspace statement below using citingspace ::bbbb,the modified source code would be:

```
Citingspace ::aaaa

……

Function callsomething()

……

Using citingspace ::bbbb

Using citingspace  // this means using top level citingspace

AnotherFunc()  // here function callsomething calls Anotherfunc()

……

Endf

……

Endcs
```

. Because statement using citingspace is closer to function call (i.e. statement AnotherFunc()) than statement using citingspace ::bbbb, the top level citingspace now has higher searching priority than citingspace ::bbbb. Now the new citingspace searching order becomes (from highest to lowest):

1. ::aaaa

2. Top level citingspace

3. ::bbbb

4. Citingspace ::mfp and its children

. Also note that the top level citingspace still has lower priority than ::aaaa because it is the innermost, i.e. direct, citingspace of the function. Direct citingspace always has the highest searching priority.

Second is the code block. A block of code is a part of program with beginning and end statements. For example, code between function/endf is a code blog. Other code block boundary statement pairs include if/endif (or if/elseif, if/else, elseif/elseif, elseif/else, elseif/endif and else/endif), for/next, while/loop, do/until, try/catch, catch/endtry and select/ends. Code block can be nested. For example,

```
If a == b
    For idx = 0 to 10 step 1
    Next
Endif
```

Is a for block nested in an if block. In MFP language, using citingspace statements in outer blocks are always visible to function calls in inner blocks if they are above the function calls. Nevertheless, outer blocks used citingspaces have lower searching priority than inner blocks used citingspaces. Comparatively, using citingspace statements in inner blocks are invisible to function calls in outer blocks regardless of their places (i.e. above or below the function calls).

Now assume user declares using citingspace ::aaaa in the if block, and declares using citingspace ::bbbb in the for block. Then in the for block the searching order is ::bbbb first and ::aaaa second, while in the if block only ::aaaa is searched.

```
If a == b
    // visible to for block, but lower priority
    Using citingspace ::aaaa
    For idx = 0 to 10 step 1
        // invisible to if block
        Using citingspace ::bbbb
    Next
Endif
```

Third, citingspace searching order in one mfps file has no effect on another mfps file. Consider, for example, two mfps files. Content of one file is:

```
Citingspace ::aaaa
Using citingspace ::bbbb
Citingspace ::cccc
Function funcA()
```

```
Endf

Endcs

Endcs
```

. Another file is:

```
Citingspace ::aaaa

Citingspace ::cccc

Using citingspace ::bbbb

Function funcB()

Endf

Endcs

Endcs
```

. Then from function funcA's perspective, citingspace searching order is ::cccc > ::aaaa > ::bbbb. From function funcB's perspective, citingspace searching order is ::cccc > ::bbbb > ::aaaa.

Fourth, all the functions in a citingspace are visible if the citingspace is used, even if the functions are defined in different mfps files. In the above example, funcB is located in a different mfps file from funcA. However, because funcB is defined in citingspace ::bbbb and funcA uses this citingspace, funcB is visible to funcA. And funcA can call funcB without explicitly providing funcB's citingspace path. Certainly, here we assume there is no another funcB without any parameter defined in citingspace ::cccc.

Fifth, citingspace can only be declared in mfps source files. It cannot be used in any other places. However, using citingspace statement can be used in the Command Line tool and JAVA based Scientific Calculator Plus. It cannot be used in Smart Calculator. In these interactive tools, user can use command

```
Shellman list_cs
```

to check all the citingspaces used, and use

```
Shellman add_cs citingspace_path
```

to add a new citingspace in the citingspace searching list. Here citingspace_path is a relative or absolute citingspace path. This command is the same as using citingspace statement.

User can also delete a used citingspace from citingspace searching list by command

```
Shellman delete_cs
```

. When using citingspace statement (or shellman add_cs command) is called in Command Line or JAVA based Scientific Calculator Plus, a new citingspace searching path is added. However, the newly added citingspace always has lower priority than the top level citingspace because the top level citingspace is the direct citingspace of the commands. And also because of this reason, when user inputs a command in Command Line or JAVA based Scientific Calculator Plus, the first two colons (::) are negligible. For instance, assume a function with full citingspace path is defined as ::abcdef::ghijk::lmn(a,b). In Command Line or JAVA based Scientific Calculator Plus, user is able to input, like abcdef::ghijk::lmn(1,2), to call this function. This is the same as inputting ::abcdef::ghijk::lmn(1,2).

The last thing user has to keep in mind is that citingspace and using citingspace cannot be used in solve block.

The follows are a citingspace example. This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder.

```
Function testCS1(a)

        print("This is ::mfpexample::testCS1, a=" + a + "\n")

endf



citingspace Abcd

Function testCS1(a)

        print("This is ::mfpexample::abcd::testCS1, a=" + a + "\n")

endf



Function testCS2(a, b)

        print("This is ::mfpexample::abcd::testCS2, a="+a+", b="+ b+"\n")

endf

endcs
```

```
citingspace ::Abcd
Function testCS1(a)
        print("This is ::abcd::testCS1, a=" + a + "\n")
endf

using citingspace abcd
Function testCSRef()
        // get error here because testCS2(a,b) is only defined in citingspace
        // ::mfpexample::abcd. Indeed using citingspace abcd has been declared
        // before the function. This function is defined in citingspace ::Abcd
        // . Because using citingspace abcd means use relative citingspace
        // abcd, so the absolute citingspace should be ::Abcd + abcd =
        // ::Abcd::abcd. This citingspace does not exist, so the function will
        // fail.
        testCS2(2,3)
endf
using citingspace ::mfpexample::abcd
Function testCSRef1()
        // here testCS2(a,b) is called after citingspace ::mfpexample::abcd is
        // declared to use. So MFP will be able to find the right function
        // which is ::mfpexample::abcd::testCS2
        testCS2(2,3)
endf
endcs

citingspace ::__efgh__
Function testCSRef1()
```

```
        // here the first testCS2(a,b) (testCS2 inside if block) is called
        // after citingspace ::mfpexample::abcd is declared to use in the if
        // block. So MFP will be able to find the right function which is
        // ::mfpexample::abcd::testCS2. The second testCS2(a,b) is in the
        // nested for block. Because the if block above the for block has
        // declared to use ::mfpexample::abcd so the second testCS2(a,b) can
        // still be found. However, the last testCS2(a,b) is out of if block
        // so that it cannot see the using citingspace ::mfpexample::abcd
        // statement so user will get error at the last testCS2 function.
        variable a = 3
        if a == 3
                using citingspace ::mfpexample::abcd
                print("::mfpexample::abcd is declared to use in this if block\n")
                testCS2(2,3)
                for variable idx = 0 to 1 step 1
                        print("::mfpexample::abcd is declared to use in the above if block\n")
                        testCS2(2,3)
                next
        endif
        print("::mfpexample::abcd is not declared to use out of if block\n")
        testCS2(2,3)
endf

Function testCSRef2()

        variable a = 3
        // call ::mfpexample::testCS1 because testCSRef2() is inside (both
    // ::__efgh__ and) ::mfpexample. Citingspace ::__efgh__ has a higher
    // priority than ::mfpexample. If there is function named testCS1
    // with one parameter is defined in ::__efgh__ it will be called.
    // However, there is not. So MFP looks for testCS1 with one parameter
    // in citingspace ::mfpexample. And there is. So ::mfpexample::testCS1
    // is called.
    testCS1(a)
    if a == 3
        using citingspace ::abcd
        // Because citingspace ::abcd has been explicitly declared to use
        // in the innermost block, it has higher priority than
        // ::mfpexample(but sill lower priority than the innermost
        // citingspace declaration, in this case it is ::__efgh__). Thus
        // ::abcd::testCS1 is called.
        testCS1(a)
    endif
    
    using citingspace ::mfpexample::abcd
    // citingspace ::mfpexample::abcd has been explicitly declared to use
    // and it is the closest using citingspace statement to the below
    // testCS1 function. As such ::mfpexample::abcd has higher priority
    // than any other citingspaces except ::__efgh__. Since ::__efgh__
    // doesn't include a testCS1 function with a single parameter,
    // ::mfpexample::abcd::testCS1 is called.
    testCS1(a)
endf
endcs
```

```
using citingspace ::mfpexample::abcd


citingspace ::__efgh__


Function testCSRef3()
        // function testCS2(a,b) is defined in the citingspace
        // ::mfpexample::abcd and statement using citingspace ::mfpexample::abcd
        // is called in the above citingspace, i.e. ::abcd (not the innermost
        // citingspace, i.e. ::__efgh__, but ::abcd as a citingspace includes
        // citingspace ::__efgh__) before this function call. Therefore, function
        //testCS2(2,3) can be found.
        testCS2(2,3)
endf
Endcs
```

In the Command Line tool or JAVA based Scientific Calculator Plus, if user runs ::mfpexample::testCS1(3), the output will be:

```
This is ::mfpexample::testCS1, a=3
```

, then if user inputs and runs

```
Using citingspace ::mfpexample
```

, the following message will be shown:

```
Citingspace has been added. Now it has higher priority than any other citingspace except the top one.
```

Then user may run testCS1(3) to see the same output as running ::mfpexample::testCS1(3).

If user runs abcd::testcs1(3), the output will be:

```
This is ::mfpexample::abcd::testCS1, a=3
```

. If user runs abcd::testcsref(), the following error message will be shown:

```
Function cannot be properly be evaluated!
```

```
In function abcd::testcsref :

D:\Development\NetBeansProjs\JCmdLine\build\scripts\manual_scripts\MFP fundamental\examples.mfps Line 275 : Invalid expression

Undefined function!
```

. However, if user runs abcd::testcsref1(), no error will be reported. The output will be:

```
This is ::mfpexample::abcd::testCS2, a=2, b=3
```

. If user runs __efgh__::testCSRef1(), the result will be:

```
::mfpexample::abcd is declared to use in this if block

This is ::mfpexample::abcd::testCS2, a=2, b=3

::mfpexample::abcd is declared to use in the above if block

This is ::mfpexample::abcd::testCS2, a=2, b=3

::mfpexample::abcd is declared to use in the above if block

This is ::mfpexample::abcd::testCS2, a=2, b=3

::mfpexample::abcd is not declared to use out of if block

Function cannot be properly be evaluated!

In function __efgh__::testcsref1 :

D:\Development\NetBeansProjs\JCmdLine\build\scripts\manual_scripts\MFP fundamental\examples.mfps Line 318 : Invalid expression

Undefined function!
```

. If run __efgh__::testCSRef2(), the result will be:

```
This is ::mfpexample::testCS1, a=3

This is ::abcd::testCS1, a=3

This is ::mfpexample::abcd::testCS1, a=3
```

. If run __efgh__::testCSRef3(), the result will be:

```
This is ::mfpexample::abcd::testCS2, a=2, b=3
```

Section 14 class and endclass Statements

1. Class declaration

The class and endclass statements define the boundary of an MFP class. The class statement is the beginning of an MFP class. If there is no super class included in the declaration, which means the class is derived from object, basic-most MFP type, the class statement is like:

class Class_Name

. If, on the other hand, the class is directly derived from one or multiple super classes, the class statement is like:

class Class_Name: Super_Class1, Super_Class2, ..., Super_ClassN

. Here, directly derived means this class is child but not grand child of the super class(es). Also, super class name could include part or full citingspace path. For example,

class Class_Name: aaa::bbb::Super_Class1, ::ccc::Super_Class2, ..., Super_ClassN

is totally valid. MFP will find out super class definitions by parsing the citingspace path based on the citingspace context, i.e. in which citingspace this class statement is and how many using citingspace statements have been declared.

2. Nested class and class members

Inside a class body are nested classes and/or class members, i.e. variables and functions. To nested class, the host class is just a citingspace. For example, if class A is defined in citingspace ::AAA::bbb and a nested class B is defined in class A, then the full citingspace path of class A is ::AAA::bbb::A and the full citingspace path of class B is ::AAA::bbb::A::B. Except the similarity of citingspace path, a nested class is independent of host class and is always visible inside and outside of the host class.

Class members are parts of class definition. There are two access modes for class members, i.e. private and public. Private members can only be accessed by same class member functions while public members can be accessed by any functions inside and outside the class. For example:

public variable self memberA = 7, memberB = "Hello", memberC

private function memberFunc(a, b, c)

...

endf

. If no access mode keyword exists, the class memeber is public.

Class member variable is declared a bit different from variable declaration statements in function. First, an access mode keyword, i.e. public or private, may exist in the beginning of the statement. Second, a self keyword must follow the variable keyword, which means the variable(s) declared in this statement are NOT static. At this moment, MFP does NOT support static member variable so that without a self keyword, the whole variable declaration statement is ignored. Last, a class member variable can be initialized in declaration statement. However, its initializer has to be a pure value, and cannot be a function. For example,

variable self varA = [[1,2]]

is right while

variable self varA = func(3,4)

is wrong. More examples of member variable declaration are:

variable self varA, varB = "Hello", varC = [[1,2],[3,4]]

private variable self varD

Like member variables, class member function declaration may have an access mode keyword, i.e. public or private in front. Also, if its parameter list starts with self, this function is not static. Inside the member function, MFP uses keyword self followed by a dot to access other members in the class. For example:

```
public function memberFunc(self, a, b, c)

		self.MemberA = a

		self.MemberB = b

		return self.MemberA * self.MemberB * self.memberFunc(c)

endf
```

. Without self in the parameter list, the member function is static. Clearly, a static member function cannot access non-static members in the same class. An example for static member function is shown below:

```
public function memberStaticFunc(a, b, c)

		return a+b+c

endf
```

.

The self keyword can access members in the class and public members in the super class(es). However, if the class and one of its super classes have the same name member, self keyword can only access the member in the class not the super class. For example:

```
class SuperClassA
        public function memberFunc(self, a)
                return a
        endf
endclass
class SuperClassB
        public function memberFunc(self, a)
                return 2*a
        endf
endclass
class ChildClass : SuperClassA, SuperClassB
        public function memberFunc(self, a)
                return 3*a
        endf
        public function memberFunc1(self)
                // call memberFunc in ChildClass, not SuperClassA or SuperClassB
                return self.memberFunc(3)
        endf
endclass
```

. To access same name super class member, super member variable is provided. This member variable is an array whose first element is the object of the class's first super, second element is the object of its second super, etc. Here, an object of its super class is a sliced object, which means the super class object is actually a part of the object of the class. As such, in the above example, if developer wants to call memberFunc in the super classes, the code should be:

```
class SuperClassA
```

```
    public function memberFunc(self, a)
        return a
    endf
endclass
class SuperClassB
    public function memberFunc(self, a)
        return 2*a
    endf
endclass
class ChildClass : SuperClassA, SuperClassB
    public function memberFunc(self, a)
        return 3*a
    endf
    public function memberFunc1(self)
        // call memberFunc in SuperClassA
        variable x = self.super[0].memberFunc(3)
        // call memberFunc in SuperClassB
        variable y = self.super[1].memberFunc(4)
        return x + y
    endf
endclass
```

. If a class is not declared with any super class, its only super class is object. In this case self.super[0] returns a sliced object of object type.

Both member variables and member functions can be overridden in MFP. In this way, when MFP refers to an object's member (variable or function), it always calls the most "bottom" member. For example, if class A is derived from class B, and both A and B have a member variable named C, and we have defined a function as

```
function func(objOfClass)
```

```
        print(objOfClass.C)
endf
```

, then if an object of A is passed into this function, A's C value is printed. If an object of B is passed into this function, B's C value is printed.

However, the above rule sometimes causes confusions. Consider, for example, class B in the above example has a public member function reading member variable C's value. The developer of class B doesn't know another developer will derive class A from B. Thus he assumes that self.C in class B's memeber function always refers to B's member variable C. However, if the third developer creates an object from class A and calls the member function of class B (which is super class of class A), class A's C value is read.

```
class B
        variable self C = 1
        function printC(self)
                print("self.C = " + self.C + "\n")
        endf
endclass


class A : B
        variable self C = 2
endclass


function printABC()
        variable bObj = B(), aObj = A()
        bObj.printC()      // self.C = 1
        aObj.printC()      // self.C = 2
endf
```

If the first developer wants to ensure that self.C only refers to class B's C member variable in class B's member function(s), this variable should be used. Basically this variable returns a (sliced) object of current class where the calling function is located, as shown in the following code:

```
class B
        variable self C = 1
        function printC(self)
                print("self.this.C = " + self.this.C + "\n")
        endf
endclass


class A : B
        variable self C = 2
endclass


function printABC()
        variable bObj = B(), aObj = A()
        bObj.printC()       // self.this.C = 1
        aObj.printC()       // self.this.C = 1
endf
```

. Different from super member variable, which is public, this member variable is private.

3. Constructor and magic functions

When creating an object from an MFP class, constructor function should be called. Different from other programming languages, MFP class constructor is a built-in function with no parameter. Developer CANNOT define, override or overload constructor. For example, if a class Abcd is defined in citingspace ::AAA::bbb, then the class constructor is function Abcd() and the full citingspace path of the function is ::AAA::bbb::Abcd(), as shown in the following code:

```
citingspace ::AAA::bbb
class Abcd
        variable self a = 1, b = "Hello", c
        public function printMembers(self)
        print("self.a = " + self.a + " self.b = " + self.b + " self.c = " + self.c)
    endf
endclass
endcs
function printABC()
    variable obj = ::AAA::bbb::abcd()
    obj.printMembers()        // self.a = 1 self.b = Hello self.c = NULL
endf
```

What a constructor does is simply setting member variables to be the values assigned to them in the variable statement. If no initializing value in the variable statement, the member variable is set as NULL. Constructor returns an object of the class.

Since user cannot define or overload a constructor, customized initialization work has to be done in a member function. The name, return type and parameter list of an initialization member function is up the the developer. However, MFP recommends to use __init__ as the function name and return the object itself. Keep in mind that __init__ is just a normal user defined function with no magic. It can be called multiple times at anytime and can be overloaded. For example:

```
citingspace ::AAA::bbb
class Abcd
    variable self a = 1, b = "Hello", c
    public function printMembers(self)
        print("self.a = " + self.a + " self.b = " + self.b + " self.c = " + self.c)
    endf
    public function __init__(self)
        self.a = 7
        self.c = (3-i) * self.a
        return self
    endf
    public function __init__(self, a, b, c) // like any user defined member function, __init__ can be overloaded.
```

```
        self.a = a
        self.b = b
        self.c = c
        return self
    endf
endclass
endcs
function printABC()
    using citingspace ::AAA::bbb
    variable obj = abcd().__init__()
    obj.__init__(3, 2, 1)
    // self.a = [5, 4] self.b = [2, 3] self.c = WWW
    obj.__init__([5,4],[2,3],"WWW").printMembers()
endf
```

MFP class has a number of magic built-in member functions which can be overridden. Function __to_string__ converts the object to a string. It is called when adding the object to a string or when MFP built-in function to_string is called with the object as the only parameter.

Function __deep_copy__ returns a deep copy of the object. It is called when MFP built-in function clone is called with the object as the only parameter.

Function __equals__ identifies if the object equals to the value of a variable. It is called when operator == is used to identify the equality of two variables.

Function __hash__ returns hash code of the object. It is called when MFP built-in function hash_code is called with the object as the only parameter.

Function __copy__ returns a shallow copy of the object. Its default behavior is to create a new object but all the member variables refer to the corresponding member variables of the old object.

Function __is_same__ identifies if the object is the same as another object referred by the parameter variable. This function simply compares reference so that its action is the same as the default behavior of operator ==. This function is useful when developer overrides __equals__ and wants to compare reference inside the overridden __equals__ function. MFP does NOT recommend to override this function.

The following example demonstrates the usage of the above functions:

```
class SampleClass
    variable self a = 1, b = 2
    public function __equals__(self, o)
        print("User defined __equals__\n")
        if self.__is_same__(o) // identify if self and o refer to the same object.
            return true
        elseif null == o
            return false
        elseif get_type_fullname(self) != get_type_fullname(o)
            return false
        elseif or(self.a != o.a, self.b != o.b)
            return false
        else
            return true
        endif
    endf
    public function __to_string__(self)
        return "Class SampleClass, a = " + a + " b = " + b
    endf
    public function __hash__(self)
        print("User defined __hash__\n")
        return a + b * 19
    endf
    public function __copy__(self)
```

```
                print("User defined __copy__\n")
                return self
        endf
        public function __deep_copy__(self)
                print("User defined __deep_copy__\n")
                variable o = SampleClass()
                o.a = self.a
                o.b = self.b
                return o
        endf
endclass
function testOverriddenMagicFunctions()
        variable obj1 = SampleClass()
        print(obj1)         // will output Class SampleClass, a = 1 b = 2
        variable obj2 = clone(obj1) // will output User defined __deep_copy__
        // will first output User defined __equals__, then output obj1 == obj2 is true
        print("obj1 == obj2 is " + (obj1 == obj2))
        print(hash_code(obj2)) // will first output User defined __hash__, then output 39
endf
```

Section 15 call and endcall Statements

The call and endcall statements define the boundary of a MFP call block which will be executed in a new thread of the current process, a different process in local host or a remote process in a different host. The call statement begins a call block. It is followed by a local keyword if the thread of call block runs in the current process; or a connection object, then on keyword, then the call statement parameter variables if the call block runs in a different process whether remote or local. The call block parameter variables are normal variables declared before the call block. Each variable can only be used by one call block. The endcall statement ends a call block. It has an optional parameter which is returned variable. The returned variable is a normal variable declared before the call block. It cannot be in the call statement parameter variable list and cannot be used by any other call statement or endcall statement.

Basically a call statement works like a function but runs in a different thread. The current thread which triggers the call statement simply starts the call block in a new thread if local keyword is used in the call statement. Otherwise, the call block is sent to a remote process which is connected to the local process by the connection object included in the call statment. Like running in the current process, a call block running in a different process can see the snapshot of values of all the variables in the current process at the moment when the call block starts. However, a call block running in the current process can modify any visible variables (except the endcall returned variable), while a call block running in a different process can only update the values of the call statement parameter variables. If a call block runs in a different process, the current process can see all the updates of the call statement parameter variables made by the call block but MFP doesn't do real-time sync nor guarantee the sync order. Similarly, the current process can also update values of the call statement parameter variables. The remote call block can see all the updates but MFP doesn't do real-time sync nor guarantee the sync order. MFP can only guarantee that one update of a variable in one process is atomic. Please note that this atomic feature only exists inside process. Since remote process and the current process both have a copy of call statement parameter variables, it is valid that the two copies of a call block variable are updated in different processes simultaneously.

A call block finishes its running when a return statement is hit or when the call block ends. If the return statement running in the call block does return a value, the endcall statment in the current thread will receive the returned value from the call block thread. If the endcall statement includes a returned variable name, the value of the returned variable will be updated by the received returned value.

Please note that, different from the parameter variables in the call statement, the returned variable included in the endcall statement is a blocked variable. This means that any statement trying to use its value will be blocked until the call block returns (It doesn't matter if the call block returns a value or not).

An example of local call block, i.e. starting a new thread, is shown below:

```
variable a = 3, b = 4

//Because the call block starts a new thread in the current process, every variable,

//except the endcall returned variable, is readable and writable. As such no need to

//use on keyword and declare call block parameters.

call local

    a = "HELLO"

    suspend_until_cond(a) // suspend call block thread until variable a's value is changed.
    // sleep 1 second to ensure that the thread which launches this call block can arrive at suspend_until_cond

    sleep(1000)

    b = 24 // set b value 24 so that the thread which launches this call block can continue

endcall


sleep(1000) // sleep 1 second to ensure that call block thread can arrive at suspend_until_cond

a = 9 //change variable a's value so that call block thread can continue

//block current thread at variable b. If b's value is updated to 24 then continue

suspend_until_cond(b, false, "==", 24)

//Now a and b have been updated in the call block. The current thread can see the updated values

print_line("a = " + a + " b = " + b)
```

Another example is for inter-process call block, i.e. call block is sent to run in a different process in local or a remote machine:

```
variable local_interface, remote_interface, ret

// client address

local_interface = ::mfp::paracomp::connect::generate_interface("TCPIP", "192.168.1.101")

ret = ::mfp::paracomp::connect::initialize_local(local_interface)

print("initialize_local ret = " + ret + "\n")

// server address

remote_interface = ::mfp::paracomp::connect::generate_interface("TCPIP", "192.168.1.107")

ret = ::mfp::paracomp::connect::connect(local_interface, remote_interface)          // connect to server from client

print("connect ret = " + ret + "\n")

// return of connect function is a dictionary. The value of "CONNECT" key is the connection object definition

// if connect failed, the value of "CONNECT" key is NULL.

variable conn = ::mfp::data_struct::array_based::get_value_from_abdict(ret, "CONNECT")
variable a = "hekko, 48", b = 3+7i, c=["LCH"]
variable d = 27     // variable d is used as a lock for synchronization between call block thread and the current thread
call conn on a, b, d // only updates of variables a, b and d in the call block thread can be seen by the current thread
    print("Before suspend_until_cond(d, false, \"==\", 888), d = " + d + "\n")
    suspend_until_cond(d, false, "==", 888)     // block until d's value is updated to 888
    print("After suspend_until_cond(d, false, \"==\", 888), d = " + d + "\n")
    sleep(5000)     //sleep 5 seconds to ensure that the thread which launches this call block can arrive at suspend_until_cond
    d = 213         // change d's value. The thread that launches this call block will receive the new value and then it can resume
    //sleep the call block thread again. Now the thread which launches this call block should
    //have arrived at print_line("c = " + c). Because the call block hasn't returned, the thread which
    //launches this call block cannot read c's value so it is blocked again.
    sleep(5000)
    a = 88
    b = "KIL"
    return 54
endcall c


sleep(10000)     //sleep 10 seconds to wait for call block thread to start and arrive at suspend_until_cond
d = 888 //change d's value to 888. After d's new value is received by the call block, it can continue
suspend_until_cond(d)     // block the current thread waiting for change of d's value
print_line("New value of d is " + d)//when we can get back c's value, call block must have returned
 // the current thread is blocked here. It will continue after c's new value is returned from the call block
// only after value of c has been retrieved, we can print a and b's values and we can see
```

```
print_line("c = " + c)

// that they have been updated. If we print a and b's values before print("c = " + c), we

// MAY see a and b are not updated.

print("a = " + a + " b = " + b)



close_connection(conn)      // close connection

close_local(local_interface) // close local interface
```

The above code is for the client process. In the server side, we have to run the following code to accept connection and run call block:

```
variable local_interface, ret

local_interface = ::mfp::paracomp::connect::generate_interface("TCPIP", "192.168.1.107")    // server address

ret = ::mfp::paracomp::connect::initialize_local(local_interface)

print("initialize_local ret = " + ret + "\n")// listen to any connection request. A listen thread will work in the background

ret = ::mfp::paracomp::connect::listen(local_interface)

print("listen ret = " + ret + "\n")// This input statement prevents server to quit if server code is a simple MFPS script

// and runs from bash or Windows command line. In Android or MFP JAVA GUI, the following

// input statement is not required as long as Scientific Calculator Plus app / MFP

// JCmdline program is still running, because server is not terminated anyway.

input("Press any key to exit\n", "S")
```

Start server side code first and then run client side code in a different device. Make sure server address and client address are both correct. In the server side two messages are printed. One is Before suspend_until_cond(d, false, "==", 888), d = 27, the other is After suspend_until_cond(d, false, "==", 888), d = 888. In the client side variables a, b, c and d's new values are printed. In particular, variable c is returned as a dictionary with returned value, 54, included.

Section 16 @compulsory_link Annotation

When creating an APK package, Scientific Calculator Plus does not copy all user defined .mfps files. Instead, it extracts related .mfps files which include all referred functions. In some situations, e.g. calling integrate or plot_exprs function, parameter is a string or string based variable so that Scientific Calculator Plus is not able to identify which function will be called in the run at compiling time. User, therefore, needs to add an annotation, @compulsory_link, before the calling statement explicitly telling Scientific Calculator Plus what functions should be linked into package. For instance

...

@compulsory_link get_functions("::mfp_example::expr1", "::mfp_example::expr2(2) ")

integrated_result = integrate(expression_str, variable_str)

...

In this example, ::mfp_example::expr1 and ::mfp_example::expr2 are user-defined functions which may be called by function integrate to be integrated. All the functions with full name (i.e. citingspace plus function name) ::mfp_example::expr1 will be compiled into APK package, no matter how many parameters they have. However, only function whose full name is ::mfp_example::expr2 and who has exactly 2 parameters or has optional parameter(s) are compiled into APK package.

If user wants to include all user defined functions, please use annotation statement

@compulsory_link get_all_functions()

. Scientific Calculator Plus will link-in all the functions. However, this means that user has to ensure that all referred functions have been defined. Otherwise, there will be compiling error.

Note that get_functions and get_all_functions are both MFP's built-in functions. However, they are located in citingspace ::mfp_compiler, not ::mfp. This citingspace will only be automatically loaded when packaging MFP script into APK. In other scenarios, user is not able to see them except using their full function name or load the ::mfp_compiler citingspace manually before calling the functions.

Also note that the @compulsory_link annotation should be located inside the body of the called function. If it is above function statement or below endf statement, it will not take any effect.

Section 17 @build_asset Annotation

When using MFP language to develop games or some applications that need sounds or images, MFP script needs to read and use sound or image files. These files are called resource files. If the MFP script is only stored locally, that is, running on a hard disk or ROM, developer only needs to place resource files in script's directory, or any other directory and then indicate the full paths of the resource files in the code. However, MFP script can be packaged into Android App and may also be sent to a remote sandbox to run. Here, sandbox is the concept of MFP parallel computing, referring to an MFP session running in a remote device. When an MFP script is packaged or sent to a remote device, its associated resource files must also be packaged or sent to the far end. Therefore, developer must use @build_asset annotation to tell MFP the new location of a resource file after packaging or sending it, and then tell MFP how to find the resource file at the runtime in different scenarios.

The following code snippet demonstrates how to properly copy an resource file, i.e. food.png, to the destination location and how to load the resource file at run time. Note that, as an annotation, the @build_asset statement is executed at compiling time, i.e. when we are building an APK from the MFP script or when the MFP script is sending source code(s) and resource file(s) to a remote device. Also note that the @build_asset statement is very long so that it is broken into three lines using MFP's line breaker, i.e. space followed by an underscore character.

The @build_asset statement considers three cases using function iff whose details can be obtained by typing help iff in a command line. First is that compiling occurs in a remote session, or sandbox in the terminology of MFP. This can happen when the remote session launches another remote session so that it needs to transfer the resource file to the new remote session. In this case function is_sandbox_session() returns true and the resource file must be located in the resource sub-folder of a temporary directory whose path is returned by function get_sandbox_session_resource_path(). The second case is that compiling happens in an MFP app. This is the situation when the MFP app kicks off a remote session and prepares to transfer the resource file. In this case function is_mfp_app() returns true. Also, as mentioned above, the source path in this case is not a string but a three element array. The array's first element is 1 which means the source resource file is in an Android app's APK. The second element is a function call, i.e. get_asset_file_path("resource"), which returns the path of resource.zip file in Android app's asset. The last element is the zip entry of the source resource file to the resource.zip in Android app's asset. The third case is that compiling occurs when the MFP script is running on JVM or in Android but as a standalone script (i.e. not as an Android app). Because in this example the resource file is located in the same folder as the source script in this case, function get_src_file_path() is called to return source script full path, and then call function get_upper_level_path to obtain the path of the folder where the source script and the resource file are both located.

```
@build_asset copy_to_resource(iff(is_sandbox_session(), get_sandbox_session_resource_path() + "images/food.png", _

            is_mfp_app(), [1, get_asset_file_path("resource"), "images/food.png"], _

            get_upper_level_path(get_src_file_path()) + "food.png"), "images/food.png")
```

```
if is_sandbox_session()
	foodImage = load_image(get_sandbox_session_resource_path() + "images/food.png")
elseif is_mfp_app()
	foodImage = load_image_from_zip(get_asset_file_path("resource"), "images/food.png", 1)
else
	foodImage = load_image(get_upper_level_path(get_src_file_path()) + "food.png")
endif
```

The lines following the @build_asset statement are executed at run-time. Similiarly three cases are considered. The first case is running in a remote session, or called a sandbox by MFP. In this case the file food.png is located in a folder named images in the resource folder of a temporary directory. The resource folder's path is returned by function get_sandbox_session_resource_path(). The second case is running as an MFP app. In this case the resource file, i.e. food.png, is located in resource.zip file in the app's asset. Function call get_asset_file_path("resource") returns the path of resource.zip file in Android app's asset. "images/food.png" is the zip entry of the source resource file to the resource.zip in Android app's asset. The third case is that the game is running on JVM or in Android as a standalone script. In this example, the resource file is located in the same folder as the source script in this case, function get_src_file_path() is called to return source script full path, and then call function get_upper_level_path to obtain the path of the folder where the source script and the resource file are both located. Note that only in case 2, i.e. the game is running as an Android app, the resource file is saved in a zip file as a zip entry. In the other two cases, the resource file is a normal file in hard disk or ROM. So in case 2, function load_image_from_zip is called while in the other two cases function load_image is called to load the image. To get detailed use of the two functions, simply type help load_image_from_zip and help load_image in MFP command line.

Like @compulsory_link annotation, @build_asset should be located inside the body of the called function. If it is above function statement or below endf statement, it will not take any effect.

Section 18 @execution_entry Annotation

As demonstrated in the previous chapter, an mfps script can be executed like any other scripting languages, e.g. Perl and Python. However, when an MFP interpreter calls a mfps script file, it needs to know which function is the entry point. The @execution_entry annotation is the statement telling an MFP interpreter which function to run.

The syntax of @execution_entry is

```
@execution_entry function_name(param_string1, param_string2, …)
```

, where function_name is the function name with / without (partial) citingspace. Because @execution_entry statement must be located above any citingspace or using citingspace declaration, MFP interpreter therefore only searches default citingspaces (e.g. :: and ::mfp) to locate the function. So if full citingspace path is not provided, user needs to ensure that MFP interpreter can still find the function. Also, the function is unnecessarily defined in the same mfps script file. It can be implemented in another script file or even a built-in function. If MFP interpreter can find the function, then the script file can run.

The param_string1, param_string2, …, are parameters for the execution entry function. Note that these parameters should be written in the same way as in normal function call except with some placeholder characters, i.e. # and @. For instance, in the following statement, the execution entry function is create_file. This function includes two parameters, the first parameter is a string based file name, the second parameter is a boolean. Then "Date_" + @ means when an MFP interpreter runs the script file from system console, the first parameter is treated as a string, and it is appended to Date_ to construct full file name. Note that user cannot use "Date_@" for the first file parameter because placeholder becomes a plain character in double quotes. Different from @, # means the second parameter is treated as a numeric value.

```
@execution_entry create_file("Date_" + @, #)
```

So when user calls the script file (assuming the file name is myscript.mfps) in a console using the following command:

```
Mfplang.cmd myscript.mfps 20161015.log false
```

, MFP interpreter will actually call

```
create_file("Date_20161015.log", false)
```

. The benefit to use placeholder characters instead of traditional $args variable (which is diffused in almost every scripting language) is clear and significant. First it avoids manual parameter parsing work, second it is smart enough to identify a string from a numeric value, third it is compatible with control characters like , [] and (), last it does not jeopardize the citingspace - function structure of MFP. For example, assume user wants to use an array as a parameter, but the array includes both value and string elements, then MFP only needs user to declare a statment like

```
@execution_entry ::my_cs::my_func ([#, 3, "Hello", [@, 2.41, #]])
```

, then in a console box, user simply types like

```
Mfplang.cmd myscript.mfps 77+38.44i, [aabbcc] [5.49]
```

, MFP interpreter will know to execute

```
::my_cs::my_func ([77+38.44i, 3, "Hello", ["[aabbcc]", 2.41, [5.49]]])
```

. One thing user has to keep in mind. In a console command, parameters are separated by space. So in the above example, no space is allowed in between 77+38.44i. Otherwise, MFP interpreter will try to replace the first placeholder char by the part before space and the second placeholder char by the part after space.

@execution_entry supports optional parameters. @... and ... means all the optional parameters are treated as strings while #... means all the optional parameters are treated as numeric values. Note that an @execution_entry can only have one optional parameter declaration and it must be the last piece in parameter declaration (i.e. exactly before the close bracket). For example

```
@execution_entry f1 (#, @, @...)
```

or

```
@execution_entry f1 (#, @, ...)
```

means when user call the script file, the script file needs at least two parameters, first one is treated as a numeric value, second one is treated as a string. If there are more than two parameters, the third parameter and on-wards are all looked on as strings. Comparatively, @execution_entry f1 (#, @, #...) means from the first parameter all the parameters are treated as numeric values.

@execution_entry can also be used without any parameter declaration, for example

```
@execution_entry func1
```

. This means @execution_entry will try to match any number of parameters transferred from console. And all these parameters are treated as strings.

If an mfps script file declares @execution_entry, it can be configured self-executable. In Android, user only needs to open the File Manager tool and long click the file to run it. A dialog box will be populated to ask for file parameters if any parameters are required. In Windows with JAVA support, user needs to associate .mfps file type with mfplang.cmd file (by right clicking an mfps file and selecting “open with”).But please note that Windows file association does not support additional file parameters so user cannot pass any parameters to the mfps script.

In Unix/MacOSX/Linux/Cygwin, making mfps self-executable is not that straightforward. First, user needs to call chmod 777 mfps_file_name to make the file executable. Second, user needs to create a soft link in /usr/bin folder named mfplang linking to the mfplang.sh file in AnMath folder. Third, user needs to add a shebang line on top of the mfps file as below:

```
#!/usr/bin/mfplang
```

. Then the mfps script is self-executable and it is able to accept file parameters.

Nevertheless, the right way to declare shebang should be

```
#!/usr/bin/env mfplang
```

. But this is related to the use of Unix/Linux operating system and not easy to config. So it is not covered in this manual.

Section 19 Deploying User Defined Functions

In order to implement and run user defined function(s), the following steps are required:

1. Start Scientific Calculator Plus;

2. Open Script Editor, type your program and save.

Clearly, it is not an easy thing to type in a mobile phone. So what user can do is

1. Connect your mobile device to a PC via an USB cord;

2. Enable write-access to your mobile device's internal storage and/or SD card;

3. Find your mobile storage (either internal storage or SD card) in PC, find AnMath folder, and navigate to AnMath/scripts. In this folder, create a new .mfps file, say my_prog.mfps.

4. Edit my_prog.mfps using any text editor in PC. An example function (This example can be found in the examples.mfps file in MFP fundamental sub-folder in the manual's sample code folder) would be like:

```
function myFunc (value1, value2, value3, value4)

    Variable avg_value

    avg_value = (value3 - value1) - (value4 - value2)

    Return avg_value

Endf
```

. Then save the file and disconnect your mobile from PC.

5. Open Scientific Calculator Plus;

6. Start command line, type

```
::mfpexample::myfunc(1,2,3,4)
```

, then type enter, result which is 0 will be shown.

Please note that

1. If input user defined function from PC, user may need to restart Scientific Calculator Plus after disconnected from PC. Otherwise, Scientific Calculator Plus may not load your new function.

2. Function title must be declared, i.e.

```
Function XXXX(...)

Endf
```

is needed. A script without function declaration is not supported.

3. Ensure that different functions do not share the same name. Ensure that user defined function name is different from predefined or build-in functions. We suggest user always declares function name with initial MyF*****.

4. Some mobiles, e.g. Sumsang Galaxy Express, do not allow user to create a new file from SD card or phone's internal storage after the phone is connected to PC. This prevents virus to transfer from phone to PC or vice versa and ensures security of both phone and PC. However, this feature makes some small troubles when user wants to create his/her own functions. A work around is copying the whole AnMath folder to a read-write enabled folder in PC, adding new .mfps files there and debugging them using JAVA based Scientific Calculator Plus. Then user can copy the whole AnMath folder back and replace the old AnMath folder in SD Card.

5. The .mfps files created by user must be located in AnMath\scripts folder or its subfolders. The name of a subfolder may include Unicode characters. For example, user may create an abc.mfps and save it in the following folder:

```
AnMath\scripts\mylib\文件库1
```

The folder name includes Chinese characters but it is no problem for Scientific Calculator Plus. When the software starts, file AnMath\scripts\mylib\文件库1\abc.mfps will be found and all the functions in it will be loaded.

Summary

This chapter introduces the semantics of MFP keywords. MFP programming language is intuitive and straightforward. Its syntax is similar to Basic so that user with some programming experience can easily understand. User without any programming experience can read this chapter as an initiating programming tutorial.

When programming MFP, user needs to provide a function for each activity. Inside the function, user needs to create variable(s) using variable statement. The conditional statements of MFP are if …… elseif …… else …… endif and select …… case …… default …… ends. The looping statements include while, for and do. If want to jump to the beginning of the loop or jump out of the loop, user may use continue and break respectively.

MFP supports handling exceptions and recovering after exception (using try …… throw …… catch …… endtry). MFP also supports solving math expressions which can help user verify calculation results.

From version 1.7, citingspace is introduced into MFP. This is a milestone in the evolution of MFP programming language. Citingspace ensures MFP's expanding potential and is an important part of object-orientated programming. However, citingspace might be hard to understand for beginners. Fortunately, MFP's backward compatibility allows user to ignore it and program in the old way.

From version 1.7.1, annotation @execution_entry is introduced. With this new feature, mfps script files can run like other scripting languages, e.g. Python and Perl. In Android, user can also execute an mfps script file from the File Manager tool by long-clicking the file.

Chapter 3 Numbers, Strings and Arrays

MFP provides language level supports to different types of numbers, strings, and arrays. Also, a number of built-in functions have been included to handle these data types.

Section 1 Functions to Operate Numbers

1. Integer Functions

MFP programming language provides functions to round a fractional value and calculate modulus of integers.

Three functions, round, ceil and floor, are able to round a fractional value. Function round half-adjusts a value. For example, round(1.6) returns 2, round(-1.6) returns -2, round(1.4) returns 1, round(-1.4) returns -1.Function ceil returns the minimum integer which is no less than its parameter. For example, ceil(1.6) returns 2, ceil(-1.6) returns -1, ceil(-5.0) returns -5. Function floor returns the maximum integer which is no greater than its parameter. For example, floor(1.6) returns 1, floor(-1.4) returns -2 and floor(-5.0) returns -5.

Round, ceil and floor also support rounding a value to selected decimal places. In this case, all the three functions need two parameters. The first parameter is the value to round. The second parameter is how many decimal places to round to. For example, round(-1.82347, 4) half-adjusts -1.82347 to 4 decimal places so that it returns -1.8235; ceil(-1.82347, 4) returns the minimum value which is no less than -1.82347 and includes 4 decimal places, so the value is -1.8234; floor(-1.82347, 1) returns the maximum value which is no larger than -1.82347 and includes 1 decimal place, so that value is -1.9.

The function to calculate modulus of integers in MFP is mod. Mod(x,y) returns modulus of its two parameters. If x or y is not an integer, it will be converted to integer first. The converting approach is, if the value is positive, then it is converted to the maximum integer which is no larger than it; if the value is negative, then it is converted to minimum integer which is no smaller than it. For example, mod(-17.8214, 4.665) equals mod(-17, 4) and its result is 3, while mod(17.8214, 4.665) equals mod(17, 4) and its result is 1.

The following example is for the above four functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual's sample code folder.

```
Help

@language:

  test round, ceil, floor and mod functions
```

```
@end
@language:simplified_chinese
  测试round，ceil，floor和mod等几个函数
@end
endh
function testRoundsMod()
 print("\n round(1.6) == " + round(1.6))
 print("\n round(1.4) == " + round(1.4))
 print("\n round(-1.6) == " + round(-1.6))
 print("\n round(-1.4) == " + round(-1.4))
 print("\n ceil(1.6) == " + ceil(1.6))
 print("\n ceil(-1.6) == " + ceil(-1.6))
 print("\n ceil(-5.0) == " + ceil(-5.0))
 print("\n floor(1.6) == " + floor(1.6))
 print("\n floor(-1.4) == " + floor(-1.4))
 print("\n floor(-5.0) == " + floor(-5.0))
 print("\n round(-1.82347, 4) == " + round(-1.82347, 4))
 print("\n ceil(-1.82347, 4) == " + ceil(-1.82347, 4))
 print("\n floor(-1.82347, 1) == " + floor(-1.82347, 1))
 print("\n mod(-17.8214, 4.665) == " + Mod(-17.8214, 4.665))
 print("\n mod(17.8214, 4.665) == " + Mod(17.8214, 4.665))
endf
```

The output of the above example should be as follows:

```
round(1.6) == 2
 round(1.4) == 1
 round(-1.6) == -2
 round(-1.4) == -1
```

```
ceil(1.6) == 2

ceil(-1.6) == -1

ceil(-5.0) == -5

floor(1.6) == 1

floor(-1.4) == -2

floor(-5.0) == -5

round(-1.82347, 4) == -1.8235

ceil(-1.82347, 4) == -1.8234

floor(-1.82347, 1) == -1.9

mod(-17.8214, 4.665) == 3

mod(17.8214, 4.665) == 1
```

2. MFP Base Number Converters

MFP programming language supports binary, octal, decimal and hexadecimal numbers, either integer or floating. A binary number starts with 0b, e.g. 0b0011100; an octal number starts with 0, e.g. 0371.242 or 00.362; a hexadecimal number starts with 0x, e.g. 0xAF46BC.0DD3E. Though MFP is able to recognize these numbers, they will be converted to decimal numbers before calculation. For example, the following code snippet:

```
Variable a = 0b1101 //equal to Variable a = 13

Print("a+1 = " + (a+1))
```

will give out

```
a+1 = 14
```

. The result is decimal although a’s initial value in the source code is a binary number.

The following functions are able to convert parameter from one base number system to another base number system. However, user may keep in mind that, if the destination base number system is not decimal, the returned result is a string. Otherwise, the returned result is a decimal value:

Conv_bin_to_dec(x) converts a non-negative binary value or a string representing a non-negative binary value to a decimal number.

Conv_bin_to_hex(x) converts a non-negative binary value or a string representing a non-negative binary value to a string representing a hexadecimal value.

Conv_bin_to_oct(x) converts a non-negative binary value or a string representing a non-negative binary value to a string representing an octal value.

Conv_dec_to_bin(x) converts a non-negative decimal value or a string representing a non-negative decimal value to a string representing a binary value.

Conv_dec_to_hex(x) converts a non-negative decimal value or a string representing a non-negative decimal value to a string representing a hexadecimal value.

Conv_dec_to_oct(x) converts a non-negative decimal value or a string representing a non-negative decimal value to a string representing an octal value.

Conv_oct_to_bin(x) converts a non-negative octal value or a string representing a non-negative octal value to a string representing a binary value.

Conv_oct_to_dec(x) converts a non-negative octal value or a string representing a non-negative octal value to a decimal number.

Conv_oct_to_hex(x) converts a non-negative octal value or a string representing a non-negative octal value to a string representing a hexadecimal value.

Conv_hex_to_bin(x) converts a non-negative hexadecimal value or a string representing a non-negative hexadecimal value to a string representing a binary value.

Conv_hex_to_dec(x) converts a non-negative hexadecimal value or a string representing a non-negative hexadecimal value to a decimal number.

Conv_hex_to_oct(x) converts a non-negative hexadecimal value or a string representing a non-negative hexadecimal value to a string representing an octal value.

Also note that, when using the above functions, a string representing a binary or octal or hexadecimal value should not include the prefix of base number system. For example, the string for binary number 0b1101 is not "0b1101 " but "1101 ".

The following example is for the above base number converters. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual's sample code folder.

Help

```
@language:
 test conversion functions between bin, oct, dec and hex
@end
@language:simplified_chinese
  测试进制转换函数
@end
endh
function testBinOctDecHex()
 print("\n\nconv_bin_to_dec(\"00111001.000110\") = " _
   + conv_bin_to_dec("00111001.000110"))
 print("\n\nconv_bin_to_hex(\".1000110001\") = " _
   + conv_bin_to_hex(".1000110001"))
 print("\n\nconv_bin_to_oct(\"1000110001\") = " _
   + conv_bin_to_oct("1000110001"))
 print("\n\nconv_dec_to_bin(\".487960\") = " _
   + conv_dec_to_bin(".487960"))
 print("\n\nconv_dec_to_bin(.487960) = " _
   + conv_dec_to_bin(.487960))
 print("\n\nconv_dec_to_bin(0.48700) = " _
   + conv_dec_to_bin(0.48700))
 print("\n\nconv_dec_to_hex(\"153439.000\") = " _
   + conv_dec_to_hex("153439.000"))
 print("\n\nconv_dec_to_hex(153439.000) = " _
   + conv_dec_to_hex(153439.000))
 print("\n\nconv_dec_to_hex(153) = " _
   + conv_dec_to_hex(153))
 print("\n\nconv_dec_to_oct(\"1356.2341\") = " _
  + conv_dec_to_oct("1356.2341"))

 print("\n\nconv_dec_to_oct(1356.2341) = " _

  + conv_dec_to_oct(1356.2341))

 print("\n\nconv_dec_to_oct(1356) = " _

  + conv_dec_to_oct(1356))

 print("\n\nconv_hex_to_bin(\"0AB0039BA.FFE01BBC64\") = " _

  + conv_hex_to_bin("0AB0039BA.FFE01BBC64"))

 print("\n\nconv_hex_to_dec(\"0AB0039BA.FFE01BBC64\") = " _

  + conv_hex_to_dec("0AB0039BA.FFE01BBC64"))

 print("\n\nconv_hex_to_oct(\"0AB0039BA\") = " + conv_hex_to_oct("0AB0039BA"))

 print("\n\nconv_oct_to_bin(\"027400330.017764\")        =        "        +
conv_oct_to_bin("027400330.017764"))

 print("\n\nconv_oct_to_dec(\"027400330.017764\")        =        "        +
conv_oct_to_dec("027400330.017764"))

 print("\n\nconv_oct_to_hex(\"027400330\") = " + conv_oct_to_hex("027400330"))

endf
```

Running the example, the output is:

```
conv_bin_to_dec("00111001.000110") = 57.09375

conv_bin_to_hex(".1000110001") = 0.8c4

conv_bin_to_oct("1000110001") = 1061

conv_dec_to_bin(".487960") =
0.0111110011101010111100100101000111000000110010011101100111010011010
00101100011001101001000001010111110100010111100000101101001110000100
0111011011110010101001
```

111101111100111011011001000101101000011100101011000000010000011000100
10011011101001011110001

conv_dec_to_hex("153439.000") = 2575f

conv_dec_to_hex(153439.000) = 2575f

conv_dec_to_hex(153) = 99

conv_dec_to_oct("1356.2341") =
2514.1676677220777134443505161674646552054171173545773053

conv_dec_to_oct(1356.2341) =
2514.1676677220777134443505161674646552054171173545773053

conv_dec_to_oct(1356) = 2514

conv_hex_to_bin("0AB0039BA.FFE01BBC64") =
10101011000000000011100110111010.11111111111000000001101110111100011
001

conv_hex_to_dec("0AB0039BA.FFE01BBC64") =
2868918714.99951337193851941265165805816650390625

conv_hex_to_oct("0AB0039BA") = 25300034672

conv_oct_to_bin("027400330.017764") =
10111100000000011011000.0000011111111101

conv_oct_to_dec("027400330.017764") = 6160600.0312042236328125

conv_oct_to_hex("027400330") = 5e00d8

3. Logic Functions in MFP

Logic functions accept Boolean parameters and return a Boolean value. They are usually used in if or elseif statement, or conditional function iff.

MFP provides three logic functions, i.e. and, or and xor.

And function accepts at least one parameter and returns logic and of the parameter(s). If any parameter is not an Boolean, it will be converted to Boolean first. If conversion is unsuccessful, an exception will be thrown.

For example, and(True, 3>2, 1-1) returns false because in the three parameters, Boolean values of True and 3>2 are both true, while 1-1 is 0 which means false in Boolean. Because one parameter is false, function and returns false. However, if the third parameter is changed to 1-2, which is -1 and Boolean value is true, function and will return true.

Function or accepts at least one parameter and returns logic or value of the parameter(s). If any parameter is not Boolean, it will be converted to Boolean. If conversion is unsuccessful, an exception will be thrown.

For example, or(True, 3>2, 1-1) returns true because Boolean values of True, 3>2 and 1-1 include true. However, if the first parameter is False, and the second parameter is 3<2, their Boolean values are changed to false so that function or is going to return false.

Function xor calculates logic xor of its two parameters, i.e. if the two parameters are equal, it returns true, otherwise, returns false.

The logic functions listed above are different from bitwise operators including bitwise and &, bitwise or | and bitwise xor ^. First of all, operands of bitwise operators must be positive integer or can be converted to positive integer. Comparatively, logic functions, except xor, only accept Boolean parameters or at least the parameter can be converted to Boolean.

Second, bitwise operator only has two operands, while logic functions and and or can accept an arbitrary number (>0) of parameters.

Last, bitwise operators perform calculation on each bit of parameters. For example, to calculate 7&8, MFP needs to figure out that 7's binary value is 111 and 8's binary value is 1000. As such 7&8 gets 0000 which is 0. Comparatively, logic function and converts 7 and 8 to Boolean and then calculate. Therefore it returns true.

Function iff is also related to logic calculation. Function iff is the function version of if statement. It needs at least three parameters. Its usage is:

iff(condition1, result_if_condition1_holds, condition2, result_if_condition2_holds, condition3, result_if_condition3_holds, ……, result_if_no_condition_is_true)

. For example, iff(true, 3, 2) returns 3, iff(3 < 2, 3, 2) returns 2 because 3 < 2 is false, iff(3<2, 3, 5>4, 5, 6==9, 6, 9) returns 5, and iff(3<2, 3, 5<4, 5, 6==9, 6, 9) returns 9.

The following example is for the above functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual’s sample code folder.

```
Help
@language:
 test logic operation functions
@end
@language:simplified_chinese
   测试逻辑函数
@end
Endh
function testLogic()
 print("\n and(True, 3>2, 1-1) = " + and(True, 3>2, 1-1))
 print("\n and(True, 3>2, 1-2) = " + and(True, 3>2, 1-2))
 print("\n or(True, 3>2, 1-1) = " + or(True, 3>2, 1-1))
 print("\n or(False, 3<2, 1-1) = " + or(False, 3<2, 1-1))
 print("\n 7&8 = " + (7&8)) // result is 0（结果为 0）
 print("\n and(7, 8) = " + and(7, 8)) // result is true（结果为 true）
 print("\n iff(true, 3, 2) = " + iff(true, 3, 2))
 print("\n iff(3 < 2, 3, 2) = " + iff(3 < 2, 3, 2))
 print("\n iff(3 < 2, 3, 5 > 4, 5, 6 == 9, 6, 9) = " _
   + iff(3 < 2, 3, 5 > 4, 5, 6 == 9, 6, 9))
 print("\n iff(3 < 2, 3, 5 < 4, 5, 6 == 9, 6, 9) = " _
   + iff(3 < 2, 3, 5 < 4, 5, 6 == 9, 6, 9))
endf
```

Output of the above example is:

```
and(True, 3>2, 1-1) = FALSE
```

and(True, 3>2, 1-2) = TRUE

or(True, 3>2, 1-1) = TRUE

or(False, 3<2, 1-1) = FALSE

7&8 = 0

and(7, 8) = TRUE

iff(true, 3, 2) = 3

iff(3 < 2, 3, 2) = 2

iff(3 < 2, 3, 5 > 4, 5, 6 == 9, 6, 9) = 5

iff(3 < 2, 3, 5 < 4, 5, 6 == 9, 6, 9) = 9

4. Functions for Complex Numbers in MFP

A complex number includes real part and image part, and can be also represented as a radius with an angle. Correspondingly MFP provides 4 functions: real, image, abs and angle to return a complex number's real part, image part, absolute value (i.e. radius) and angle. Note that function abs can also return a real number's absolute value.

For example, real(-3+2i) returns -3, image(-3+2*i) returns 2, and image(-3+2*i, true) returns 2*i. Note that function image has an optional second parameter. If this parameter is true, it returns image part's image value, otherwise, it returns image part's real value. By default, it is false. Abs(-3+2*i) returns 3.60555128, angle(-3+2*i) returns 2.55359006 which is based on arc.

The following example is for the above functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual's sample code folder.

```
Help
@language:
 test complex functions
@end
@language:simplified_chinese
  测试复数操作函数
@end
```

```
Endh

function testComplexFuncs()

  print("\nreal(-3+2*i) = " + real(-3+2i))

  print("\nimage(-3+2*i) = " + image(-3+2*i))

  print("\nabs(-3+2*i) = " + abs(-3+2 * i))

  print("\nangle(-3+2*i) = " + angle(-3+2i))

endf
```

The result output would be:

```
real(-3+2*i) = -3

image(-3+2*i) = 2

abs(-3+2*i) = 3.6055512754639893469033040673821233212947845458984375

angle(-3+2*i) =
2.5535900500422257345460612287910790449857054711524495709749445923
```

Section 2 String Functions in MFP

MFP is able to determine size of a string, append a string, separate a string, etc.

If user wants to know how many characters are in a string, strlen function should be called. For instance, strlen("abcdefg!") returns 8, which means 8 characters are in the parameter string, i.e. "abcdefg!".

Then user may want to know the third and fourth characters in the parameter string. Function strsub is able to undertake this task. Strsub(str, start, end) returns parameter str (which is a string)’s sub-string. The sub-string starts from character start (the first character is character 0) and end at character end-1. For example, strsub("abcdefg!", 2, 4) returns a new string "cd", where character “c” is the third character (index is 2) and character “d” is the fourth character (index is 4-1 = 3).

Different from other programming languages like C++, MFP does not have char (character) data type. Even with only one character, it is still a string. For example strsub("abcdefg!", 2, 3) returns a single char string "c".

The third parameter of function strsub is optional. If it does not exist, the returned sub-string ends at the end of parameter string. For example, strsub("abcdefg!", 2) returns "cdefg!".

If the second or the third parameter falls outside the string's character indexing range, an exception will be thrown. For example, if user runs strsub("abcdefg!", 2, 10), an error message "Invalid parameter range!" will be printed.

If user wants to link several strings end to end and construct a new big string, function strcat can be used. For example, strcat("abc","hello", " 1,3,4") returns "abchello 1,3,4". Alternatively, user may use operator + to connect strings. For example, "abc"+"hello"+ " 1,3,4" also gives "abchello 1,3,4".

If user wants to cut a string into several parts, function split should be selected. Split(string_input, string_regex) cuts parameter string_input into several sub-strings following the rule defined by regular expression string_regex, and it returns an array whose elements are the sub-strings. Regular expression can be very complicated. User may read JAVA documentation about Pattern class and String.split function for more details. In this section, examples are provided to show user how to split a string by blank, by colon, by an English letter or by comma.

For example, split(" ab kkk\t6\nd", "\\s+") returns ["", "ab", "kkk", "6", "d"] because "\\s+" means any kind of blank characters in regular expression. This includes space, \t (tab character) and \n (change line character). Moreover, in this example, more than one blank character locates between "ab" and "kkk". But because the "+" in the regular expression parameter, i.e. "\\s+", adjacent blank characters are recognized as a single separating sign.

For other examples, split("boo:and:foo", ":") returns ["boo", "and", "foo"] and split("boo:and:foo", "o") returns ["b", "", ":and:f"]. Here the regular expression parameter does not include a "+" so that adjacent "o"s are treated as independent cutting signs. As such "boo: …" are splitted into "b", "" and ":…".

Split((",Hello,world,", ",") uses "," as splitting sign and returns ["","Hello", "world"]. Since nothing is before the first ",", the first sub-string is empty. Comparatively, nothing is after the last "," but no sub-string is returned after the last ",". This is a common feature of split function, no matter what the regular expression is.

Besides the above functions, trim_left, trim_right and trim remove all the blank characters (including space, "\t" and "\n") from left side, right side and both left and right sides of parameter string respectively and returns the new string after processing. For example, Trim("\t \tabc   def \n ") returns "abc   def".

Function to_lowercase_string converts all the capitalized letters in the parameter string to corresponding lower case letters. Function to_uppercase_string converts all the lower case letters in the parameter string to corresponding upper case letters. Function to_string returns string value of its parameter which can be any data type. For example, to_lowercase_string("abEfg") returns "abefg" and to_string(123) returns "123".

MFP is also able to compare strings. Function strcmp compares any parts of two strings. Its usage is strcmp(src, dest, src_start, src_end, dest_start, dest_end), which means comparing part of src string (from src_start to src_end) with part of dest string (from dest_start to dest_end). If src is larger than dest, returns a positive value, if src is smaller than dest, returns a negative value, otherwise, returns zero. Note that src_start, src_end, dest_start and dest_end are all character indices starting from 0, and src_end and dest_end should be the index of last selected character plus one. Also note that the last four parameters are optional with default value for src_start and dest_start is 0 and default value for src_end and dest_end is corresponding string length.

Function stricmp is similar to strcmp. Its usage is also stricmp(src, dest, src_start, src_end, dest_start, dest_end). However, stricmp is case insensitive. In other words, stricmp converts every big letter in the comparing parts of the two strings to corresponding lower case letter before comparison.

Here two examples are presented to demonstrate how to use strcmp and stricmp.

Stricmp("abc","ABc") returns 0 because “abc” equals “ABc” if ignore case.

Strcmp("abcdefgk", "defik", 5, 8, 2, 5) compares the 6th, 7th and 8th characters of string "abcdefgk" with the 3rd, 4th and 5th characters of string "defik", i.e. compares "fgk" with "fik". It returns -2 which means "fgk" is less than "fik".

MFP provides two functions to transform between a string and an integer array. Function conv_ints_to_str converts an integer array to a Unicode string. Each integer is converted to a character in the string. This function also accepts a single integer as parameter which will be converted to a single character string. Function conv_str_to_ints converts a Unicode string to an integer array. In general, a Unicode character should be mapped to an integer (Very rarely a Unicode character is mapped to two integers if it is out of the scope of UTF-16 character set). Because many characters, e.g. ¥∑⑨ and Chinese characters, are Unicode, these two functions may help user input and output some special characters. For example:

```
conv_str_to_ints(″中文汉字¥Σ⑨″)
```

returns [20013, 25991, 27721, 23383, 165, 8721, 9320]. Then user may use the return of conv_str_to_ints as the parameter of conv_ints_to_str to get the special characters, i.e.

conv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320])

returns ″中文汉字¥Σ⑨″.

The following example is for the above functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual’s sample code folder.

```
Help
@language:
 test string functions
@end
@language:simplified_chinese
  测试字符串操作函数
@end
Endh
function testString()
 print("\nstrlen(\"abcdefg!\") = " + strlen("abcdefg!"))
 print("\nstrsub(\"abcdefg!\", 2, 4) = " + strsub("abcdefg!", 2, 4))
 print("\nstrsub(\"abcdefg!\", 2, 3) = " + strsub("abcdefg!", 2, 3))
 print("\nstrsub(\"abcdefg!\", 2) = " + strsub("abcdefg!", 2))
 print("\nstrcat(\"abc\",\"hello\", \" 1,3,4\") = " _
   + strcat("abc","hello", " 1,3,4"))
 print("\nsplit(\" ab  kkk\\t6\\nd\", \"\\\\s+\") = " _
   + split(" ab  kkk\t6\nd", "\\s+"))
 print("\nsplit(\"boo:and:foo\", \":\") = " _
   + split("boo:and:foo", ":"))
 print("\nsplit(\"boo:and:foo\", \"o\") = " _
   + split("boo:and:foo", "o"))
 print("\nsplit(\",Hello,world,\", \",\") = " _
   + split(",Hello,world,", ","))
 print("\nTrim(\"\\t \\tabc   def  \\n  \") = " _
   + Trim("\t \tabc   def  \n  "))
 print("\nto_lowercase_string(\"abEfg\") = " _
   + to_lowercase_string("abEfg"))
 print("\nto_string(123) = " + to_string(123))
 print("\nstricmp(\"abc\",\"ABc\") = " + stricmp("abc","ABc"))
 print("\nstrcmp(\"abcdefgk\", \"defik\", 5, 8, 2, 5) = " _
   + strcmp("abcdefgk", "defik", 5, 8, 2, 5))
  print(″\nconv_str_to_ints(\″中文汉字¥Σ⑨\″) = ″ _
     + conv_str_to_ints(″中文汉字¥Σ⑨″))
 print("\nconv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320]) = " _
+ conv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320]))
endf
```

The output of the above example is:

```
strlen("abcdefg!") = 8
strsub("abcdefg!", 2, 4) = cd
strsub("abcdefg!", 2, 3) = c
strsub("abcdefg!", 2) = cdefg!
strcat("abc","hello", " 1,3,4") = abchello 1,3,4
split(" ab  kkk\t6\nd", "\\s+") = ["", "ab", "kkk", "6", "d"]
split("boo:and:foo", ":") = ["boo", "and", "foo"]
split("boo:and:foo", "o") = ["b", "", ":and:f"]
split(",Hello,world,", ",") = ["", "Hello", "world"]
Trim("\t \tabc   def  \n  ") = abc   def
to_lowercase_string("abEfg") = abefg
to_string(123) = 123
stricmp("abc","ABc") = 0
strcmp("abcdefgk", "defik", 5, 8, 2, 5) = -2
```

```
conv_str_to_ints(″中文汉字¥Σ⑨″) = [20013, 25991, 27721, 23383,
165, 8721, 9320]
```

```
conv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320])
= 中文汉字¥Σ⑨
```

Section 3 Functions to Operate Arrays (Matrices)

Array, or its mathematical name matrix, is one of the data types supported by MFP programming language. Array includes several elements. Each element can be a number, a string or an array. Besides the arithmetic operators like +, -, * and /, MFP also provides functions to process arrays.

1. Functions to Create Array

MFP provides several functions to create array. The first one is alloc_array function. This function accepts one or several parameters. If parameter number is more than 1, each of them must be a positive integer, which means the size of the array at each dimension. If only one parameter is used, the parameter must be an array, and each element of the array must be a positive integer which stores the size of one dimension of the array. The elements in the created array are all initialized as zero. For example, alloc_array(3) returns [0, 0, 0], while alloc_array(2,3,4) and alloc_array([2,3,4]) both return [[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]].

If, however, user wants alloc_array to create an array whose elements are not initialized as zero, but some other values, an additional parameter is required. The usage of alloc_array in this case is alloc_array(x,y), where x is an array whose elements are positive integer which means size of each dimension of created array, y is the initial value of each element of created array. For example, alloc_array([2,1],"hello") returns [["hello"], ["hello"]].

Besides alloc_array, functions eye, ones and zeros also create and return a new array. Eye(x) has one parameter, i.e. x, which must be a positive integer, and returns a x times x square matrix I. Note that eye(0) returns 1 as a special case.

Function ones returns a matrix whose elements are all 1. Similar to alloc_array, ones's parameter(s), either a single positive integer array parameter or several positive integer parameters, determine size of the returned matrix.

Function zeros returns a matrix whose elements are all 0. Similar to alloc_array, zeros's parameter(s), either a single positive integer array parameter or several positive integer parameters, determine size of the returned matrix.

The following example is for the above functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual's sample code folder.

```
Help

@language:

  test array construction functions

@end

@language:simplified_chinese

  测试创建数组的函数

@end

Endh

function createArray()

  print("\nalloc_array(3) = " + alloc_array(3))

  print("\nalloc_array(2,3,4) = " + alloc_array(2,3,4))

  print("\nalloc_array([2,3,4]) = " + alloc_array([2,3,4]))

  print("\nalloc_array([2,1],\"hello\") = " + alloc_array([2,1],"hello"))

  print("\neye(3) = " + eye(3))

  print("\nones(2,3) = " + ones(2,3))

  print("\nzeros([2,3]) = " + zeros([2,3]))

endf
```

The output of the above example is:

```
alloc_array(3) = [0, 0, 0]

alloc_array(2,3,4) = [[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]]

alloc_array([2,3,4]) = [[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]]

alloc_array([2,1],"hello") = [["hello"], ["hello"]]

eye(3) = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

ones(2,3) = [[1, 1, 1], [1, 1, 1]]

zeros([2,3]) = [[0, 0, 0], [0, 0, 0]]

2. Size and Other Properties of Array

In order to acquire size of an array, user needs to call size function. Size function can accept one parameter, i.e. size(x) where x is an array and returns x's size vector for all the dimensions. If x is not an array, size(x) returns []. Note that elements in size vector should be the maximum size in each dimension. For example, array [1, 2+3i, [5, "hello", [9, 10], 6], 11, 12] includes 5 elements which are 1, 2+3i, [5, "hello", [9, 10], 6], 11 and 12. As such, the size of the first dimension is 5. For the second dimension, size of 1, 2+3i, 11 and 12 is always [] (here a complex number is treated as a non-matrix single value). However, element [5, "hello", [9, 10], 6] is an array with 4 elements so that the second dimension's size is 4. For the third dimension, 5, "hello" and 6 are all simple values (string is also non-matrix simple value) so that their size is []. But element [9, 10] has two elements so that the third dimension's size is 2. As a final result, we get size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12])==[5, 4, 2].

The other usage of function size is taking two parameters, i.e. size(x, y) and returns matrix's first y dimensions' sizes. Here x should be an array and y is a positive integer. If x is not an array, size function returns []. For example,

size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12], 2)

returns the first two dimension sizes of parameter [1, 2+3i, [5, "hello", [9, 10], 6], 11, 12] which is [5,4].

User may use the following functions to check property of a matrix.

Function is_eye(x) identifies if x is an I matrix;

Function is_zeros(x) identifies if x is an all-zero matrix;

Functions includes_inf(x), includes_nan(x), includes_null(x), includes_nan_or_inf(x) and includes_nan_or_inf_or_null(x) are able to identify parameter x is a matrix including at least an element whose value is inf or –inf, nan, null, either nan or inf or –inf, or either nan or inf or –inf or null.

The following example is for the above functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual's sample code folder.

```
Help

@language:

 acquire array's properties functions
```

```
@end
@language:simplified_chinese
  获取数组特性的函数
@end
Endh
function getArrayProperty()
 print("\nsize([1, 2+3i, [5, \"hello\", [9, 10], 6], 11, 12]) = " _
   + size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12]))
 print("\nsize([1, 2+3i, [5, \"hello\", [9, 10], 6], 11, 12], 2) = " _
   + size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12], 2))
 print("\nis_eye([[1,1],[0,1]]) = " + is_eye([[1,1],[0,1]]))
 print("\nis_zeros([[0,0],0]) = " + is_zeros([[0,0],0]))
 print("\nincludes_nan_or_inf([5, [3, -inf], \"hello\"]) = " _
   + includes_nan_or_inf([5, [3, -inf], "hello"]))
 print("\nincludes_null([5, [3, -inf], \"hello\"]) = " _
   + includes_null([5, [3, -inf], "hello"]))
Endf
```

The output of the example is:

```
size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12]) = [5, 4, 2]

size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12], 2) = [5, 4]

is_eye([[1,1],[0,1]]) = FALSE

is_zeros([[0,0],0]) = TRUE

includes_nan_or_inf([5, [3, -inf], "hello"]) = TRUE

includes_null([5, [3, -inf], "hello"]) = FALSE
```

3. Assign Value to Array

Clearly, the traditional assigning value statement still works for array, e.g.

```
Variable a = [1,2,3]

a[1] = [7,9]
```

. However, array differs from other data type in the indexing range. If the index is not in the valid range, i.e. from zero to array length – 1, traditional assigning statement will lead to a run-time error. In the above example, a does not have the fourth or fifth element. If user wants to assign 3+6i to the fifth element of a, i.e. a[4], statement a[4] = 3+6i cannot be used.

Function set_array_elem is the right approach to assign value to any element, whether exists or not, in an array. Set_array_elem(x, y, z) assigns z to x[y] and returns new x. Please keep in mind that x is not necessarily to be an array, while y must be a positive integer vector, e.g. [1,2,3]. Y, however, can be beyond array x's valid indexing range. For example, if x is 3, y is [1,2] and z is 2+3i, set_array_elem(x,y,z) returns [3, [0, 0, 2+3i]]. Also note that, inside set_array_elem function, x may or may not be changed to a new value. In order to ensure x is updated, the right way to call set_array_elem is:

```
x = set_array_elem(x, y, z)
```

. So to assign 3+6i to a[4], the right way is:

```
a = set_array_elem(a, 4, 3+6i)
```

. After the above statement runs, a's value becomes [1, [7, 9], 3, 0, 3 + 6 * i]. a[3] did not exist before the statement, but because a[4] is created, a[3] has to be created as well. a[3]'s value is set to be default which is 0.

User may need to use array as parameter when creating new functions. Inside the function, user assigns new value using the traditional assigning statement to some elements of the array. Please note that, if elements of array change value in a sub-function, main function can also see the change. For example, user may define a sub-function like:

```
function subfunc1(array_value)

…

Variable my_array = Array_value

my_array[2] = 7

…

Endf
```

In the main function user calls ::mfpexample::subfunc1 in the following way:

```
…

variable array_val = [1,2,3]

::mfpexample::subfunc1(array_val)

print(array_val)

…
```

, after running print(array_val), user will see array_val turns into [1, 2, 7].

The reason is, when MFP transfers an array parameter to a sub-function, the parameter is not copied into sub-function's stack in whole. Only the array's reference is sent to the sub-function. Therefore, both sub-function and main function see the same array. In the above code, sub-function assigns a value to an element in the parameter array without changing array's reference so that main function can see the value change.

If user wants to change array value in a sub-function, but hopes main function still see the value before change, function clone can be used. This function only has one parameter which can be any data type. The value of the parameter will be copied and returned. The returned value is saved in a different place in memory from the parameter so that any change on it will not affect original value. For the above example, if we change sub-function to:

```
function subfunc2(array_value)

…

Variable my_array = clone(Array_value)

my_array[2] = 7

…

Endf
```

, and call ::mfpexample::subfunc2 in main function:

```
…

variable array_val = [1,2,3]

::mfpexample::subfunc1(array_val)

print(array_val)
```

…

, after running print(array_val) we will see array_val is still [1, 2, 3].

The following example is for the above functions. It can be found in the examples.mfps file in numbers, strings and arrays sub-folder in the manual's sample code folder.

```
function subfunc1(array_value)
 Variable my_array = Array_value
 my_array[2] = 7
Endf

function subfunc2(array_value)
 Variable my_array = clone(Array_value)
 my_array[2] = 7
Endf

function assignValue2Array()
 variable array_val = [1,2,3]
 print("\narray_val's initial value is " + array_val)
 ::mfpexample::subfunc2(array_val)
 // clone function called in ::mfpexample::subfunc2, any change inside
 // will not affect main function.
 print("\nWith clone, after calling sub function array_val is " + _
   array_val)
 ::mfpexample::subfunc1(array_val)
 // clone function not called in ::mfpexample::subfunc2, value changes
 // of array_val inside will affect main function.
 print("\nWithout clone, after calling sub function array_val is " _
   + array_val)
```

```
  array_val = set_array_elem(array_val, [4], -5.44-6.78i)

  // array_val now has 5 elements after calling set_array_elem

  print("\nAfter set_array_elem array_val is " + array_val)

endf
```

. User runs ::mfpexample::assignValue2Array() and sees the following output:

```
array_val's initial value is [1, 2, 3]

With clone, after calling sub function array_val is [1, 2, 3]

Without clone, after calling sub function array_val is [1, 2, 7]

After set_array_elem array_val is [1, 2, 7, 0, -5.44 - 6.78i]
```

Summary

MFP programming language has built-in support to real numbers, complex numbers, arrays and strings. Moreover, MFP also provides a complete set of functions to handle these data types, including rounding functions, base number converters, logic calculation functions, complex value readers/writers, string functions (including acquiring size, getting sub-string and comparing strings etc.) and array functions (including creation and acquiring properties)

User may keep in mind that matrix and array in MFP programming language are two related but not the same terms. Matrix in MFP normally means 2D square matrix, while an array can possess arbitrary number of dimensions, and the sizes of the dimensions may not necessarily be the same. Matrix is a special kind of array. This manual will introduce many matrix calculation functions later on.

Chapter 4 Mathematic Analysis and Scientific Calculation Functions

A big advantage of MFP programming language is its capability of performing mathematical analysis and scientific calculation. This chapter will introduce the related variables and functions.

Section 1 Built-in Constant Variables

In the previous sections user has been demonstrated how to use some of MFP built-in constant variables like i (unit of image value), null, nan and nani. To fully support scientific calculation, MFP also includes the following constant values:

1. Inf, which means real positive infinite. Clearly, -inf means real negative infinite.

2. Infi, which means positive infinite image value. Clearly, -infi means negative infinite image value.

   MFP provides this constant variable because infi cannot be obtained from multiplication of inf and i (inf * i == nan + infi), so that a specific variable is required.

3. pi , whose value in MFP is 3.1415926535897932384626433832795028841971693993751058209749445923.

4. e , whose value in MFP is 2.7182818284590452353602874713526624977572470936999595749669676277.

Section 2 Unit Conversion Functions and Constant Value Functions

MFP programming language provides a unit conversion function, i.e. convert_unit. Convert_unit(value, from_unit, to_unit) converts a value based on from_unit to a new value based on to_unit. Note that both from_unit and to_unit are case sensitive. For example, convert_unit(23.71,"m","km") returns 0.2371.

Convert_unit supports the following units:

1. Length units: "um" (micrometers), "mm" (millimeters), "cm" (centimeters), "m" (meters), "km" (kilometers), "in" (inches), "ft" (feet), "yd" (yards), "mi" (miles), "nmi" (nautical miles), "AU" (astronomical units), "ly" (light years), "pc" (parsec);

2. Area units: "mm2" (square millimeters), "cm2" (square centimeters), "m2" (square meters), "ha" (hectares), "km2" (square kilometers), "sq in" (square inches), "sq ft" (square feet), "sq yd" (square yards), "ac" (acres), "sq mi" (square miles);

3. Volume units: "mL" (milliliters (cc)), "L" (litres), "m3" (cubic meters), "cu in" (cubic inches), "cu ft" (cubic feet), "cu yd" (cubic yards), "km3" (cubic kilometers), "fl oz(Imp)" (fluid ounces (Imp)), "pt(Imp)" (pints (Imp)), "gal(Imp)" (gallons (Imp)), "fl oz(US)" (fluid ounces (US)), "pt(US)" (pints (US)), "gal(US)" (gallon s(US));

4. Mass units: "ug" (micrograms), "mg" (milligrams), "g" (grams), "kg" (kilograms), "t" (tonnes), "oz" (ounces), "lb" (pounds), "jin" (market catties), "jin(HK)" (catties (HK)), "jin(TW)" (catties (TW));

5. Speed units: "m/s" (meters per second), "km/h" (kilometers per hour), "ft/s" (feet per second), "mph" (miles per hour), "knot" (knots);

6. Time units: "ns" (nanoseconds), "us" (microseconds), "ms" (milliseconds), "s" (seconds), "min" (minutes), "h" (hours), "d" (days), "wk" (weeks), "yr" (years);

7. Force units: "N" (newtons), "kgf" (kilogram-forces), "lbF" (pound-forces);

8. Pressure units: "Pa" (pascals), "hPa" (hectopascals), "kPa" (kilopascals), "MPa" (megapascals), "atm" (atomspheres), "psi" (pounds per square inch), "Torr" (torrs (millimeters of mercury);

9. Energy units: "J" (joules), "kJ" (kilojoules), "MJ" (megajoules), "kWh" (kilowatt-hours), "cal" (calories), "kcal" (kilocalories), "BTU" (British Thermal Units);

10. Power units: "W" (Watts), "kW" (kilowatts), "MW" (megawatts), "cal/s" (calories per second), "BTU/h" (BTUs per hour), "hp" (horse powers);

11. Temperature units: "0C" (celsius), "0F" (fahrenheit), "K" (Kelvin);

MFP is also capable of returning some constant values in science. Function get_constant(const_name, n) returns the constant value corresponding to the case-sensitive string const_name. The returned value will be rounded to n significant digits after decimal point. Here n must be non-negative and is optional. If n is neglected, the returned value will not be rounded. This function supports the following constants:

1. Ratio of circumference of a circle to its diameter (const_name == "pi");

2. Natural logarithm (const_name == "e");

3. Light speed in vacuum [m/s] (const_name == "light_speed_in_vacuum");
4. Gravitational constant [m**3/kg/(s**2)] (const_name == "gravitational_constant");
5. Planck constant [J*s] (const_name == "planck_constant");
6. Magnetic constant [N/(A**2)] (const_name == "magnetic_constant");
7. Electric constant [F/m] (const_name == "electric_constant");
8. Elementary charge [c] (const_name == "elementary_charge_constant");
9. Avogadro constant [1/mol] (const_name == "avogadro_constant");
10. Faraday constant [C/mol] (const_name == "faraday_constant");
11. Molar gas constant [J/mol/K] (const_name == "molar_gas_constant");
12. Boltzman constant [J/K] (const_name == "boltzman_constant");
13. Standard gravity [m/(s**2)] (const_name == "standard_gravity");

For example, if user inputs get_constant("pi", 4), the result will be 3.1416; if user inputs get_constant("pi", 8), the result will be 3.14159265; if user inputs get_constant("pi", 0), s\he will get 3; if user inputs get_constant("pi"), the result will be 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679 (with 100 digits after decimal point), which is the pi value internally used by the software.

The following example is for the above two functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder.

```
Help
@language:
  test convert_unit and get_constant functions
@end
@language:simplified_chinese
  测试 convert_unit 函数和 get_constant 函数
@end
endh
```

```
function getConstCvtUnit()
 print("\nconvert_unit(23.71,\"m3\",\"fl oz(US)\") = " _
   + convert_unit(23.71,"m3","fl oz(US)"))
 print("\nget_constant(\"pi\", 4) = " + get_constant("pi", 4))
 print("\nget_constant(\"pi\", 8) = " + get_constant("pi", 8))
 print("\nget_constant(\"pi\", 0) = " + get_constant("pi", 0))
 print("\nget_constant(\"pi\") = " + get_constant("pi"))
endf
```

The result of the above example is:

```
convert_unit(23.71,"m3","fl oz(US)") =
801730.4782606974628681506740932151121554177525643799285007984748218
874

get_constant("pi", 4) = 3.1416

get_constant("pi", 8) = 3.14159265

get_constant("pi", 0) = 3

get_constant("pi") =
3.1415926535897932384626433832795028841971693993751058209749445923
```

Section 3 Trigonometrical Functions and Hyperbolic Functions

Trigonometrical and hyperbolic functions in MFP programming language have the same name and usage as in math. The only thing to keep in mind is, if no d in the end of the function name, calculation result is based on arc, otherwise, it is based on degree. For example, cos(pi/3) returns 0.5, while asind(0.5) is 30, which means 30 degrees. Also note that all these functions support complex calculation, e.g. asind(8) returns 90 – 158.63249757 * i.

All the trigonometrical functions and inverse trigonometrical functions are listed below:

Function name	Function info
acos	acos(1) :

	acos(x), where x can be a complex number, returns arccos value of x.
acosd	acosd(1) : Function acosd(x) calculates degree based arccos value of x.
asin	asin(1) : asin(x), where x can be a complex number, returns arcsin value of x.
asind	asind(1) : Function asind(x) calculates degree based arcsin value of x.
atan	atan(1) : atan(x) returns arctan value of x, where x can be a complex number.
atand	atand(1) : Function atand(x) calculates degree based arctan value of x.
cos	cos(1) : cos(x) returns cos value of x, where x can be a complex number.
cosd	cosd(1) : Function cosd(x) calculates cos x where x is a degree.
sin	sin(1) : sin(x) returns sin value of x, where x can be a complex number.

sind	sind(1) : Function sind(x) calculates sin x where x is a degree.
tan	tan(1) : tan(x) returns tan value of x, where x can be a complex number.
tand	tand(1) : Function tand(x) calculates tan x where x is a degree.

All the hyperbolic functions are listed below:

Function name	Function info
acosh	acosh(1) : Function acosh(x) calculates inverse hyperbolic cos of x.
asinh	asinh(1) : Function asinh(x) calculates inverse hyperbolic sin of x.
atanh	atanh(1) : Function atanh(x) calculates inverse hyperbolic tan of x.
cosh	cosh(1) : Function cosh(x) calculates hyperbolic cos of x.
sinh	sinh(1) : Function sinh(x) calculates hyperbolic sin of x.
tanh	tanh(1) : Function tanh(x) calculates hyperbolic tan of x.

The following example is for the above two functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder.

```
Help
@language:
  test trigonometric and hyperbolic trigonometric functions
@end
@language:simplified_chinese
    测试三角函数和双曲三角函数
@end
Endh
function testTrigHTrig()
  print("\ncos(pi/3) = " + cos(pi/3))
  print("\ntand(45) = " + tand(45))
  print("\nsin(1 + 2*i) = " + sin(1 + 2*i))
  print("\nasind(0.5) = " + asind(0.5))
  print("\nacos(8) = " + acos(8))
  print("\nacosh(4.71 + 6.44i) = " + acosh(4.71 + 6.44i))
  print("\nsinh(e) = " + sinh(e))
  print("\natanh(e) = " + atanh(e))
endf
```

Running the above example user will see:

```
cos(pi/3) = 0.5

tand(45) = 1

sin(1 + 2*i) =
3.1657785132161682200525265356615738370974142362979081716694416027 +
1.959601041421606115476765045922954107994611047413801140191859959i

asind(0.5) =
30.0000000000000030712880255288762424340850900194236602185899203 76
```

acos(8) = -2.7686593833135737519057784083997830748558044433593 75i

acosh(4.71 + 6.44i) = 2.7711160843983257962008792674168944358825683593 75 + 0.9430568597413974130105884796648751944303512573242 1875i

sinh(e) = 7.5441371028169749712866121171828126534819602966308 59375

atanh(e) = 0.3859684164526523963978377196326619014143943786621 09375 + 1.5707963267948966192313216916397514420985846996875529104874722962i

. User may notice that, the return value of asind(0.5) is 30.000000000000003071288025528876242434085090019423660218589920376 instead of 30 (degrees). This is because asind first converts the parameter, i.e. 0.5 to a complex value and then does the calculation and calculating asind of a complex value leads to a tiny error.

Section 4 Exponential, Logarithmic and Power Functions

MFP supports a number of exponential, logarithmic and power functions as below:

Function name	Function info
exp	exp(1) : exp(x), where x is a real or complex number, returns x powers of e.
lg	lg(1) : Function lg(x) returns e based log value of x.
ln	ln(1) : Function ln(x) returns e based log value of x.
log	log(1) : log(x), where x can be a complex number, returns e based logarithm value of x.
log10	log10(1) : Function log10(x) returns 10 based log value of x.
log2	log2(1) : Function log2(x) returns 2 based log value of x.

loge	loge(1) : Function loge(x) returns e based log value of x.

pow	pow(2) :
	pow(x,y) returns y powers of x. Note that both x and y can be either a real or a complex number. If there are more than one results of pow(x,y), return the first result.
	pow(3) :
	pow(x,y,z) returns a list including first z values of y powers of x. If y powers of x has less than z values, returns all the values. Note that y must be a real number while x can be either a real or a complex number. Z must be a positive integer.

sqrt	sqrt(1) : Function sqrt(x) returns square root of real number x.

Please note that:

1. Lg(x), log(x), ln(x) and loge(x) all returns natural logarithm of x. Log2(x) returns 2 based logarithm of x. Log10(x) is 10 based. All these functions accept complex value as parameter. Any other base can be obtained from division of these functions. For example, user simply uses log(x)/log(3) to get 3 based logarithm of x.

2. There are two ways to call pow function. First is pow(x, y), which returns x^y. Note that both x and y can be either real or complex value. If the result have many values, this function returns the first value it sees when anti-clockwisely rotating from 0 degree in the complex domain. This usage equals using x**y. For example, pow(32, 0.2) returns 2. However, pow(-32, 0.2) will not returns -2 although -2 is one of its roots. Instead, it returns 1.61803399 + 1.1755705 * i.

   The second way to call pow function is pow(x, y, z), which returns the first z results of x^y. Here y must be a real value while x can be either real or complex. Z should be a positive integer. The returned values are stored in a vector. If the number of results is less than z, all the results are returned. Otherwise, it returns the first z results it sees when anti-clockwisely rotating from 0 degree in the complex domain. For example, if user wants to see all the results of $-32^{0.2}$, pow(-32, 0.2, 5) can be called and a 5-element vector, i.e. [1.61803399 + 1.1755705 * i,

-0.61803399 + 1.90211303 * i, -2, -0.61803399 - 1.90211303 * i, 1.61803399 - 1.1755705 * i], is returned.

3. Sqrt(x) is equal to x**0.5 or pow(x, 0.5). It returns the first square root seen when rotating anti-clockwisely from 0 degree in the complex domain. For example, sqrt(4) == 2, sqrt(-2) == 1.41421356 * i and sqrt(-2+3i) == 0.89597748 + 1.67414923 * i.

The following example is for the above functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder.

```
Help
@language:
 test log, exp and pow and related functions
@end
@language:simplified_chinese
  测试对数，指数和次方函数
@end
endh
function testLogExpPow()
 print("\nlg(e) == " + lg(e))
 print("\nlog(9, 3) == log(9)/log(3) == " + log(9)/log(3))
 print("\nlog2(3 + 4i) == " + log2(3 + 4i))
 print("\npow(32, 0.2) == " + pow(32, 0.2))
 print("\npow(-32, 0.2) == " + pow(-32, 0.2))
 print("\npow(-32, 0.2, 5) == " + pow(-32, 0.2, 5))
 print("\nsqrt(4) == " + sqrt(4))
 print("\nsqrt(-2) == " + sqrt(-2))
 print("\nsqrt(-2+3i) == " + sqrt(-2+3i))
endf
```

The result of the above example is shown below:

```
lg(e) == 1
```

log(9, 3) == log(9)/log(3) == 2

log2(3 + 4i) == 2.3219280948873622916712631553180615794157506196217129274315603707 + 1.337804212450976175615004492526409565791145361743891813677556325i

pow(32, 0.2) == 2

pow(-32, 0.2) == 1.6180339887498949025257388711906969547271728515625 + 1.175570504584946274206913585658185184001922607421875i

pow(-32, 0.2, 5) == [1.6180339887498949025257388711906969547271728515625 + 1.175570504584946274206913585658185184001922607421875i, - 0.6180339887498946804811339461593888700008392333984375 + 1.90211303259030728440848179161548614501953125i, -2, - 0.6180339887498951245703437962220050394535064697265625 - 1.9021130325903070623638768665841780602931976318359375i, 1.6180339887498946804811339461593888700008392333984375 - 1.17557050458494671829612343572080135345458984375i]

sqrt(4) == 2

sqrt(-2) == 1.4142135623730951454746218587388284504413604736328125i

sqrt(-2+3i) == 0.8959774761298379706375607865525069497958199765590683867889064147 + 1.6741492280355400682758136732173307274213575287387175311747860088i

Section 5 Matrix Related Functions

Chapter 4 demonstrates user how to use functions to operate MFP arrays. In this section, more functions for matrix calculation are introduced.

In MFP, array is different from matrix. MFP matrix is a mathematical concept. It must be an array. It also supports many mathematical calculations, e.g. +, -, *, /, etc. Moreover, it has to be two dimensional, and in many cases it is a square matrix. The functions for matrix calculation generally cannot be used for common MFP arrays.

MFP matrix calculation functions are listed below:

Function name	Function info
adj	adj(1) : adj(x), where x is 2D square matrix, returns the adjugate matrix of x.

cofactor	cofactor(1) : cofactor(x), where x is 2D square matrix, returns the cofactor matrix of x.
det	det(1) : Function det(x) calculates determinant of square matrix x.
deter	deter(1) : Function deter(x) calculates determinant of square matrix x.
dprod	dprod(2) : Function dprod calculates dot product of two vectors [x1, x2, ... xn] and [y1, y2, ... yn].

eig	eig(1) :
	eig(A) calculates 2D square matrix A's eigen vectors and eigen values. This function returns a two element list. First element is the eigen vector matrix, each column is an eigen vector. Second element is a diagonal matrix. Each diagonal element is an eigen value. Note that this function needs big memory to run and consumes significant CPU time. If it runs in mobile device, size of A and B should be no greater than 6*6. If it runs in PC, A and B should be no greater than 8*8. Otherwise, it may fail because lack of memory or run for very long time.
	eig(2) :
	eig(A, B) calculates 2D square matrix A's eigen vectors and eigen values against same size matrix B, i.e. Av = lambda * Bv, where lambda is an eigen value and v is an eigen vector. The second parameter, B, is optional. By default, B is an I matrix. This function returns a two element list. First element is the eigen vector

	matrix, each column is an eigen vector. Second element is a diagonal matrix. Each diagonal element is an eigen value. Note that this function needs big memory to run and consumes significant CPU time. If it runs in mobile device, size of A and B should be no greater than 6*6. If it runs in PC, A and B should be no greater than 8*8. Otherwise, it may fail because lack of memory or run for very long time.

get_eigen_values	get_eigen_values(1) :
	get_eigen_values(A) calculates 2D square matrix A’s eigen values. This function returns an eigen value list which includes all the eigen values including duplicated ones. Note that this function needs big memory to run and consumes significant CPU time. If it runs in mobile device, size of A and B should be no greater than 6*6. If it runs in PC, A and B should be no greater than 8*8. Otherwise, it may fail because lack of memory or run for very long time.
	get_eigen_values(2) :
	get_eigen_values(A, B) calculates 2D square matrix A’s eigen values against same size matrix B, i.e. Av = lambda * Bv, where lambda is an eigen value and v is an eigen vector. The second parameter, B, is optional. By default, B is an I matrix. This function returns an eigen value list which includes all the eigen values including duplicated ones. Note that this function needs big memory to run and consumes significant CPU time. If it runs in mobile device, size of A and B should be no greater than 6*6. If it runs in PC, A and B should be no greater than 8*8. Otherwise, it may fail because lack of memory or run for very long time.

invert	invert(1) : invert(x) inverts 2D matrix x. Note that the elements of x can be complex numbers but x must be a square matrix (i.e. number of rows equals number of columns).
left_recip	left_recip(1) : left_recip(x) calculates left-division reciprocal of x. Note that so far x can only be a number or a 2D matrix.
rank	rank(1) : rank(matrix) returns the rank of a matrix. For example rank([[1,2],[2,4]]) returns 1.
recip	recip(1) : recip(x) calculates reciprocal of x. Note that so far x can only be a number or a 2D matrix.

User may note that:

1. Functions recip, left_recip and invert are actually the same thing for a 2-D square matrix x, which are equal to calculating 1/x.

2. Det and deter are two names for the same function. Their usage and calculation are exactly the same.

3. All the above functions support complex matrix.

The following example is for the above functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder:

```
Help

@language:

test matrix functions

@end

@language:simplified_chinese

测试矩阵相关函数

@end
```

```
endh

function testMatrix()

print("\ncofactor([[1,3,-4.81-0.66i],[-0.91i,5.774,3.81+2.03i],[0,-6,-7.66-3i]])=" _

+ cofactor([[1,3,-4.81-0.66i],[-0.91i, 5.774, 3.81+2.03i],[0, -6, -7.66-3i]]))

print("\nadj([[1,-7],[-4, 6]]) = " + adj([[1,-7],[-4, 6]]))

print("\ndet([[2.7-0.4i, 5.11i],[-1.49, -3.87+4.41i]]) = " _

+ det([[2.7-0.4i, 5.11i],[-1.49, -3.87+4.41i]]))

print("\ndprod([1,2,3],[4,5,6]) = " + dprod([1,2,3],[4,5,6]))

print("\neig([[1,0],[0,1]]) = " + eig([[1,0],[0,1]]) )

print("\neig([[1+3.7i,-0.41-2.93i,5.33+0.52i],[0.33+2.71i,-3.81i,0.41+3.37i],[2.88,0,-9.4i]])=" _

+ eig([[1+3.7i,-0.41-2.93i,5.33+0.52i],[0.33+2.71i,-3.81i,0.41+3.37i],[2.88,0,-9.4i]]))

print("\nget_eigen_values([[1+3.7i,-0.41-2.93i,5.33+0.52i],[0.33+2.71i,-3.81i,0.41+3.37i],[2.88,0,-
9.4i]])=" _

+get_eigen_values([[1+3.7i,-0.41-2.93i,5.33+0.52i],[0.33+2.71i,-3.81i,0.41+3.37i],[2.88,0,-9.4i]]))

print("\nrank([[1,2,3],[4,5,8]]) = " _

+ rank([[1,2,3],[4,5,8]]))

Endf
```

The result of the above example is:

```
cofactor([[1,3,-4.81-0.66i],[-0.91i,5.774,3.81+2.03i],[0,-6,-7.66-3i]])=[[-
21.368839999999999999999999999999999999999999999999999999999999999986 -
5.1419999999999999999999999999999999999999999999999999999999999999994i,
2.73 - 6.9706i, 5.46i], [51.84 + 12.96i, -7.66 - 3i, 6],
[39.20293999999999999999999999999999999999999999999999999999999999999
+ 9.9008400000000000000000000000000000000000000000000000000000000000001i,
-4.4106 + 2.3471i, 5.774 + 2.73i]]

adj([[1,-7],[-4, 6]]) = [[6, 7], [4, 1]]

det([[2.7-0.4i, 5.11i],[-1.49, -3.87+4.41i]]) = -8.685 + 21.0689i

dprod([1,2,3],[4,5,6]) = 32

eig([[1,0],[0,1]]) = [[[0, 0], [0, 0]], [[1, 0], [0, 1]]]
```

eig([[1+3.7i,-0.41-2.93i,5.33+0.52i],[0.33+2.71i,-3.81i,0.41+3.37i],[2.88,0,-
9.4i]])=[[[0.5823305472444549220819340004695808799844908575313355771049
880239 +
2.680907257548215607437849572543481298140621315251586887558556 7468i,
0.011413582046639550205485295448825655181747113900781052921011 9208 +
3.439433004887303245979382425236703427006029598977321488720684 2872i,
0.138864930860651023204222876956198023375926286538081560664657 5143 -
0.686935557720650199105514310087515898239342900052438522907454 5558i],
[0.170665027141073449368678271683842020455634872094878445912311 4335 +
3.261037951768426552950056683279530107408830215850611810769286 1942i,
1.035284639904452008084417539506301745719299218190824661787210 7954 +
2.071831537068021572646832656758065839434895757605464351178020 4727i,
1], [1, 1, -
1.460475554240340345462756071856744238231396298888757335354400 7743 +
0.598993739978272354579739449816818018945142764442616707911228 0372i]],
[[1.677111976064030323413256252757154481170550722353327570186943 7736 -
1.678987098261140618719141177185817783052387721675118168569465 9201i, 0,
0], [0,
0.032871116294318688788558286089080324142554159411463556807376 1979 +
0.505567054075434790415122110012536396973519737194896438541856 4962i,
0], [0, 0, -
0.709983092358349012201814538846234805313104881764791126994319 9715 -
8.336579955814294171695980932826718613921132015519778269972390 5761i]]]

get_eigen_values([[1+3.7i,-0.41-2.93i,5.33+0.52i],[0.33+2.71i,-
3.81i,0.41+3.37i],[2.88,0,-
9.4i]])=[1.6771119760640303234132562527571544811705507223533275701869 43
7736 -
1.678987098261140618719141177185817783052387721675118168569465 9201i,
0.032871116294318688788558286089080324142554159411463556807376 1979 +
0.505567054075434790415122110012536396973519737194896438541856 4962i, -
0.709983092358349012201814538846234805313104881764791126994319 9715 -
8.336579955814294171695980932826718613921132015519778269972390 5761i]

rank([[1,2,3],[4,5,8]]) = 2

Section 6 Functions for Expression Calculation and Calculus

Functions for expression calculation and calculus accept string based MFP expression(s) and carry out corresponding calculations. They are:

Function name	Function info
deri_ridders	deri_ridders(4) : deri_ridders(expr, var, val, ord) calculates ord-order derivative

	value of expression expr which is based on variable var when the variable's value is equal to val. This function always uses Ridders method. For example, deri_ridders("x**2+x","x",3,2) returns 2.

derivative	derivative(2) :
	derivative(expression, variable) calculates derivative of expression which is based on variable. Note that both expression and variable must be strings. For example, derivative("x**2+x","x") returns a string based expression which is "2*x+1".
	derivative(4) :
	derivative(expr, var, val, method) calculates derivative value of expression expr which is based on variable var when the variable's value is equal to val. The parameter method selects the method to use. True means using Ridders method while false means simply calculating derivative expression value at val. For example, derivative("x**2+x","x",2,true) returns 5.

evaluate	evaluate(1...) : evaluate(expr_string,var_string1,var_value1,var_string2,var_value2, ...) returns the value of string based expression expr_string when the string based variable var_string1 equals var_value1, variable var_string2 equals var_value2, ... respectively. Note that var_value1, var_value2, ... can be any type and the number of variables can be zero, i.e. evaluate("3+2") is valid.

integrate	integrate(2) :
	integrate(x,y) returns the indefinite integrated expression of expression x with respect to variable y where x and y are both strings. Note that if x cannot be indefinitely integrated or x is too complicated to solve, this function throws an exception.
	integrate(4) :
	integrate(x,y,z,w) returns the integrated value of expression x with respect to variable y changing from z to w. Note that x and y are string typed and z and w can be real numbers, complex numbers or strings. The integrating algorithm selected is adaptive Gauss-Kronrod method.

	integrate(5) : integrate(x,y,z,w,v) returns integrated value given a string expression x of a variable y (also a string) and an interval [z, w]. In calculation, one step length is (w - z)/v, note that v must be a positive integer while w and z can be real numbers, complex values or strings. If v is zero, this function is the same as integrate(x,y,z,w).
lim	lim(3) : lim(expr, var, dest_value) calculates the limit value of expression expr when variable var is closing to dest_value. expr and var should be strings and dest_value can be expression or value, whether string based or not. For example, lim("1/x", "x", 0) or lim("(x+2)/(x+3)","x","3+0"). Note that this function is still under development.
product_over	product_over(3) : product_over(x, y, z) calculates the product of string based expression x over integer value y to z. Note that y and z are also string based values, y should be written like "a=10" (where a is the variable) and z should be like "20". For example, product_over("x+1", "x=1", "10").
sum_over	sum_over(3) : sum_over(x, y, z) calculates the sum of string based expression x over integer value y to z. Note that y and z are also string based values, y should be written like "a=10" (where a is the variable) and z should be like "20". For example, sum_over("x+1", "x=1", "10").

Function deri_ridders calculates first, second or third derivative value of a given expression at a given point using Ridders method. The example, deri_ridders("x**2+x", "x", 3, 2), means calculating second derivative value of x**2+x when x equals 3.

Derivative calculates first derivative value of a given expression at a given point, or first derivative expression of a given expression. The first example, derivative("x**2+x","x"), means calculating first derivative expression of x**2+x. The second example, derivative("x**2+x","x", 2, true), means calculating first derivative value of x**2+x when x is 2. Note that the last parameter tells the function whether to use Ridders method or not. If it is true, Ridders method is used. If false, this function first calculates derivative expression, and then gives out the derivative value from the derivative expression. The last parameter is optional. By default, it is true.

Derivative function can also calculate higher order derivative expression. This needs nested calls. For example, derivative(derivative("x**2+x","x"),"x") gives out second derivative expression of x**2+x.

Function sum_over is the sum operator ∑ in math, and its usage is the same as ∑. For example, sum_over("x+1", "x=1", "10") means calculating sum of x + 1s where x is an integer varying from 1 to 10. Using a mathematical expression it is ∑x=110(x+1).

Similar to sum_over, product_over means ∏ in math, and its usage is the same as ∏. For example, product_over("x+1", "x=1", "10") means calculating product of all x + 1s where x is an integer varying from 1 to 10. Using a mathematical expression it is ∏x=110(x+1).

Function evaluate evaluates a string based MFP expression. If this expression has one or more variables, evaluate function takes additional parameters to assign values to the variables so that it is able to give out the calculated result. For example, evaluate("x+y+1","x",3,"y",4) calculates the value of x + y + 1 where x is 3 and y is 4. Clearly, the final result is 8. Of course, if no variable is used in the expression, no additional parameters are required.

Function evaluate can also evaluate functions, whether the to-be-evaluated function is user-defined or system provided. For example, evaluate("sind(30)") returns 0.5.

Function integrate calculates definite or indefinite integrals. In fact, the integration tool in Scientific Calculator Plus is based on this function. It is quite simple to calculate indefinite integrals. Only two parameters are needed. The first one is the expression to be integrated. The second one is variable name. Both of them are based on string. For example,

integrate("cos(x)","x")

calculates integration of cos(x), and the returned result is also a string which is "sin(x)".

There are two numerical methods to calculate definite integrals. First is using Gauss-Kronrod method. This method is able to handle the case where from and/or to value(s) are infinite. It is also able to process high-frequently oscillating functions. It is even capable of integrating functions with singular points. For example,

integrate("exp(x)","x",-inf,0)

returns the integrated value of exp(x) where x is from negative infinite to 0, and the result is 1. For another example,

```
integrate("log(x)","x",0,1)
```

calculates the integrated value of log(x) where x is from 0 to 1. The returned result is 1.00000018 (which includes a small error). Note that here 0 is a singular point.

Although Gauss-Kronrod method is powerful, it is very slow. Most of the functions, however, can adopt the traditional Riemann Sum integration algorithm which is much faster with reasonable accuracy. User, as such, needs to indicate the number of steps. Nevertheless, if the number of steps is zero, or from and/or to value(s) are infinite, integrate function still uses Gauss-Kronrod method. An example using Riemann Sum integration algorithm is:

```
integrate("x**2+1","x", -3+4i, 7-9i, 100)
```

. This example integrates x**2 + 1 in the complex domain. The number of steps is 100. The final result is -481.7345 - 225.69505 * i. Theoretical result is -481.66666667 - 225.66666667 * i which means the error will be very small if the number of steps is big enough.

User can also calculate higher-order integrals using nested integrate functions. For example,

```
integrate("integrate(\"x*y\",\"x\",1,6,100)","y",-4,3,100)
```

integrates x*y where x is from 1 to 6 and y is from -4 to 3. The result is -61.25.

Because in general higher-order integrals need much more calculation steps than first-order integral, Riemann Sum method is strongly recommended as it is fast. This is why the number of steps is not optional in higher-order integration.

Nevertheless, some integrals cannot be calculated by integrate function, e.g. very complicated integrals or integrals which cannot converge. In this case, an exception will be thrown. For example, running the following statements (by copying them and pasting them into GUI based Scientific Calculator Plus for JAVA, and then pressing ENTER key):

```
try

print("integrate(\"e**(x**2)\",\"x\")", integrate("e**(x**2)","x"))

catch

print("e**(x**2) cannot be integrated")

endtry
```

, user will see an exception printed:

e**(x**2) cannot be integrated

, which means is not capable of being integrated.

Also note that the functions introduced in this section are able to read values of declared variables. For instance, assume that in a program a variable b has been declared and its value is 3. If user calls evaluate("x+b","x",9), 12 will be returned. Because of this feature, it is recommended to use different variable name(s) in the maths expression parameter(s) to avoid potential conflict(s). This requirement is particularly important when calculating higher-order integrals.

The following example is for the above functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder:

```
Help
@language:
 test expression and calculus functions
@end
@language:simplified_chinese
  测试表达式和微积分相关函数
@end
endh
function exprcalculus()
 print("\nderivative(\"1/x**2*log(x) + 9\", \"x\") = " _
  + derivative("1/x**2*log(x) + 9", "x"))
 print("\nderivative(\"tanh(x)**-1\", \"x\") = " _
  + derivative("tanh(x)**-1", "x"))
 // test high order derivative
 print("\nderivative(derivative(\"x*sin(x)\", \"x\"), \"x\") = " _
  + derivative(derivative("x*sin(x)", "x"), "x"))
 // test derivative value
 print("\nderi_ridders(\"x**0.5+x+9\", \"x\", 0.3, 1) = " _
   + deri_ridders("x**0.5+x+9", "x", 0.3, 1))
 print("\nderivative(\"x**0.5+x+9\", \"x\", 0.3) = " _
   + derivative("x**0.5+x+9", "x", 0.3))
 print("\nderi_ridders(\"x**0.5+sqrt(sin(x**2))\", \"x\", 0.3, 3) = " _
   + deri_ridders("x**0.5+sqrt(sin(x**2))", "x", 0.3, 3))
 print("\nsum_over(\"1/(x-10)\",\"x=1\",\"9\") = " _
   + sum_over("1/(x - 10)", "x = 1", "9"))
 print("\nproduct_over(\"1/(x-10)\",\"x=9\",\"1\") = " _
   + product_over("1/(x-10)", "x = 9", "1"))
 print("\nevaluate(\"x+y+1\",\"x\",5,\"y\",7) = " _
   + evaluate("x+y+1","x",5,"y",7))
 print("\nevaluate(\"sind(30)\") = " + evaluate("sind(30)"))
 print("\nintegrate(\"tanh(x)**-1\",\"x\") = ")
 print(integrate("tanh(x)**-1","x"))
 print("\nintegrate(\"sinh(x)*cosh(x)**-1\",\"x\") = ")
 print(integrate("sinh(x)*cosh(x)**-1","x"))
 print("\nintegrate(\"1/x**2\",\"x\",2,inf) = ")
 print(integrate("1/x**2","x",2,inf))
 print("\nintegrate(\"1/x**2\",\"x\",2,50,100) = ")
 print(integrate("1/x**2","x",2,50,100))
 // test unintegratable.
 try
 print("integrate(\"e**(x**2)\",\"x\")", integrate("e**(x**2)","x"))
 catch
 print("e**(x**2) cannot be integrated")
 endtry
 // test high order integration
```

```
  print("\nintegrate(\"integrate(\\\"x*y\\\",\\\"x\\\",1,6,100)\",\"y\",-4,3,100) = ")

  print(integrate("integrate(\"x*y\",\"x\",1,6,100)","y",-4,3,100))

endf
```

Output of the above example is:

```
derivative("1/x**2*log(x) + 9", "x") = (-2)*log(x)*x**(-3)+x**(-3)

derivative("tanh(x)**-1", "x") = -(-
0.5)*2.7182818284590452353602874713526624977572470936999595749669676277**x*(sinh(x)/cosh(x))**(-2)*sinh(x)*cosh(x)**(-2)+(-
0.5)*2.7182818284590452353602874713526624977572470936999595749669676277**(-x)*(sinh(x)/cosh(x))**(-2)*sinh(x)*cosh(x)**(-2)+(-
0.5)*2.7182818284590452353602874713526624977572470936999595749669676277**x*(sinh(x)/cosh(x))**(-2)/cosh(x)+(-
0.5)*2.7182818284590452353602874713526624977572470936999595749669676277**(-x)*(sinh(x)/cosh(x))**(-2)/cosh(x)

derivative(derivative("x*sin(x)", "x"), "x") = (-1)*x*sin(x)+2*cos(x)

deri_ridders("x**0.5+x+9", "x", 0.3, 1) =
1.9128709291772078606011099231055019184972226816921057762830380748

derivative("x**0.5+x+9", "x", 0.3) =
1.912870929175276901723634637164650484919548034667968 75

deri_ridders("x**0.5+sqrt(sin(x**2))", "x", 0.3, 3) =
7.1575232288636571107632429280365926329437264758531027037489228909

sum_over("1/(x-10)","x=1","9") = -
2.8289682539682539682539682539682539682539682539682539682539682540

product_over("1/(x-10)","x=9","1") = -
0.0000027557319223985890652557319223985890652557319223985890652557

evaluate("x+y+1","x",5,"y",7) = 13

evaluate("sind(30)") = 0.5

integrate("tanh(x)**-1","x") = log(sinh(x))

integrate("sinh(x)*cosh(x)**-1","x") = log(cosh(x))

integrate("1/x**2","x",2,inf) =
0.4999999999999999800759152415811779636542182411236778807349767932
```

integrate("1/x**2","x",2,50,100) = 0.48474650870065751246583179175056732567588197853949780307108217481e**(x**2) cannot be integrated

integrate("integrate(\"x*y\",\"x\",1,6,100)","y",-4,3,100) = -61.25

. Note that in version 1.7, citingspace was introduced into MFP programming language. As such, when calculating indefinite integrals or derivative expressions in this version, each function the answer includes complete citingspace path. For example, log(x) in older versions turns to ::mfp::math::log_exp::log(x) in version 1.7. This change makes the answer longer and hard to read. So from version 1.7.1 it is improved to only show the minimum citingspace path in the result. In other words, ::mfp::math::log_exp::log(x) is returned back to log(x) gain.

Section 7 Statistic, Stochastic and Sorting Functions

Scientific Calculator Plus provides a number of statistic, stochastic and sorting functions as below:

Function name	Function info
avg	avg(1...) : Function avg(...) returns average value of an arbitrary number of parameters.
beta	beta(2) : Function beta(z1, z2) returns beta function value of complexes z1 and z2, note that real part of z1 and z2 must be positive.
gamma	gamma(1) : Function gamma(z) returns gamma function value of complex z, note that real part of z must be positive.
gavg	gavg(1...) : Function gavg(...) returns geometric mean value of an arbitrary number of parameters.
havg	havg(1...) : Function havg(...) returns harmonic mean value of an arbitrary

	number of parameters.
max	max(1...) : Function max(...) returns maximum value of an arbitrary number of parameters.
med	med(1...) : Function med(...) returns medium value of an arbitrary number of parameters. If the number of parameters is even, returns average of the middle two parameters.
min	min(1...) : Function min(...) returns minimum value of an arbitrary number of parameters.
quick_sort	quick_sort(2) : Function quick_sort(desc, original_list) returns a sorted list of an arbitrary number of parameters. If desc is true (or 1), list elements are from largest to smallest, otherwise (desc is false or 0), from smallest to largest. For example, quick_sort(1, [5,6,7,9,4])'s result is [9,7,6,5,4] while quick_sort(0, [5,6,7,9,4]) is [4,5,6,7,9].
ncr	ncr(2) : Function nCr(x, y) calculates the number of y-combination of a set S which has x elements. Note that x, y are non-negative integers, x >= y.
npr	npr(2) : Function nPr(x, y) calculates the number of y-permutation of a set S which has x elements. Note that x, y are non-negative integers, x >= y.
rand	rand(0) : rand() function returns a random float number between 0 (inclusive) and 1 (exclusive).
stdev	stdev(1...) : Function stdev(...) returns standard deviation of an arbitrary number

	of parameters. Note that the parameters are a sample of a larger set.
stdevp	stdevp(1...) : Function stdevp(...) returns standard deviation of an arbitrary number of parameters.
sum	sum(1...) : Function sum(...) returns sum value of an arbitrary number of parameters.

Note that functions stdev and stdevp are different. If the two functions have same parameters which are x_1， x_2， x_3， …， x_N , stdev returns

$$\sqrt{\frac{1}{N-1}((x_1-u)^2+(x_2-u)^2+(x_3-u)^2+...+(x_N-u)^2)}$$

, while stdevp returns

$$\sqrt{\frac{1}{N}((x_1-u)^2+(x_2-u)^2+(x_3-u)^2+...+(x_N-u)^2)}$$

. u here is the average value of x_1， x_2， x_3， …， x_N .

The following example is for the above functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder:

```
Help
@language:
  test statistics and sorting functions
@end
@language:simplified_chinese
  测试统计、随机和排序相关函数
@end
endh
function testStatSort()
```

```
 print("\navg(1,5,9,-6,3,-18,7) = " + avg(1,5,9,-6,3,-18,7))
 print("\nbeta(3.71, 23.55) = " + beta(3.71, 23.55))
 print("\ngamma(5.44 - 10.31i) = " + gamma(5.44 - 10.31i))
 print("\ngavg(1,5,9,-6,3,-18,7) = " + gavg(1,5,9,-6,3,-18,7))
 print("\nhavg(1,5,9,-6,3,-18,7) = " + havg(1,5,9,-6,3,-18,7))
 print("\nmax(1,5,9,-6,3,-18,7) = " + max(1,5,9,-6,3,-18,7))
 print("\nmed(1,5,9,-6,3,-18,7) = " + med(1,5,9,-6,3,-18,7))
 print("\nmin(1,5,9,-6,3,-18,7) = " + min(1,5,9,-6,3,-18,7))
 print("\nquick_sort(1,[1,5,9,-6,3,-18,7]) = " _
   + quick_sort(1,[1,5,9,-6,3,-18,7]))
 print("\nquick_sort(0,[1,5,9,-6,3,-18,7]) = " _
   + quick_sort(0,[1,5,9,-6,3,-18,7]))
 print("\nstdev(1,5,9,-6,3,-18,7) = " + stdev(1,5,9,-6,3,-18,7))
 print("\nstdevp(1,5,9,-6,3,-18,7) = " + stdevp(1,5,9,-6,3,-18,7))
 print("\nsum(1,5,9,-6,3,-18,7) = " + sum(1,5,9,-6,3,-18,7))
 print("\nncr(8,3) = " + ncr(8,3))
 print("\nnpr(8,3) = " + npr(8,3))
 print("\nrand() = " + rand())
endf
```

The above example outputs the follows:

```
avg(1,5,9,-6,3,-18,7) =
0.1428571428571428571428571428571428571428571428571428571429

beta(3.71, 23.55) =
0.0000279537392314725872716390423881975646941670888511331711318296

gamma(5.44 - 10.31i) =
0.00153606217320356956205529368949431837178203181366617456390043119 -
0.0279816213196075726360710743099268272949427989554500480691282345i

gavg(1,5,9,-6,3,-18,7) =
5.194584255413065676521000568754971027374267578125
```

havg(1,5,9,-6,3,-18,7) =
4.472616632860040567951318458417849898580121703853955375253549696

max(1,5,9,-6,3,-18,7) = 9

med(1,5,9,-6,3,-18,7) = 3

min(1,5,9,-6,3,-18,7) = -18

quick_sort(1,[1,5,9,-6,3,-18,7]) = [9, 7, 5, 3, 1, -6, -18]

quick_sort(0,[1,5,9,-6,3,-18,7]) = [-18, -6, 1, 3, 5, 7, 9]

stdev(1,5,9,-6,3,-18,7) = 9.3528707077661721314143505878746509552001953125

stdevp(1,5,9,-6,3,-18,7) =
8.65907569182385117301237187348306179046630859375

sum(1,5,9,-6,3,-18,7) = 1

ncr(8,3) = 56

npr(8,3) = 336

rand() = 0.67638281271680666950629756684065796434879302978515625

Section 8 Signal Processing Functions

Signal processing functions are included in Scientific Calculator Plus for electrical engineers. There are three functions, conv (convolution), FFT (Fast Fourier Transform), iFF (Inverse Fast Fourier Transform):

Function name	Function info
conv	conv(2) : conv(input_a, inputb) returns convolution of input_a and input_b. input_a and input_b can either be two 1-D lists or two 2-D arrays. So far conv function only support 1-D and 2-D convolution. For example, conv([4,8,2,9],[5,3,8,9,6,7,8]) = [20, 52, 66, 151, 139, 166, 181, 132, 79, 72] conv([[4,8,2,9],[8,6,7,9],[2,2,8,-4]],[[-5,i,7],[0.6,8,4]]) = [[-20, -40 + 4 * i, 18 + 8 * i, 11 + 2 * i, 14 + 9 * i, 63], [-37.6, 6.8 + 8 * i, 102.2 + 6 * i, 50.4 + 7 * i, 129 + 9 * i, 99], [-5.2, 57.6 + 2 * i, 58.2 + 2 * i, 119.4 + 8 * i, 156 - 4 * i, 8], [1.2, 17.2, 28.8, 69.6, 0, -16]]

FFT	FFT(1...) : Function FFT(a, ...) returns Fast Fourier Transform of a series of values, note that the number of values in the series should always be 2 to a positive integer. If a is a list of real or complex numbers, this function should only have one parameter and return Fast Fourier Transform of a[0], a[1], ... a[N-1] where N is the number of elements in a. If a is a single value (real or complex), this function should have at least two parameters and return Fast Fourier Transform of a, optional_params[0], optional_params[1], ..., optional_params[number_of_optional_params - 1]. The returned value is always an array. Examples of this function: FFT(1, 2, 3, 4) returns [10, -2 + 2i, -2, -2 - 2i]; FFT([1, 2, 3, 4]) also returns [10, -2 + 2i, -2, -2 - 2i].

IFFT	IFFT(1...) :
	Function IFFT(a, ...) returns Inverse Fast Fourier Transform of a series of values, note that the number of values in the series should always be 2 to a positive integer. If a is a list of real or complex numbers, this function should only have one parameter and return Inverse Fast Fourier Transform of a[0], a[1], ... a[N-1] where N is the number of elements in a. If a is a single value (real or complex), this function should have at least two parameters and return Inverse Fast Fourier Transform of a, optional_params[0], optional_params[1], ..., optional_params[number_of_optional_params - 1]. The returned value is always an array.
	Examples of this function:
	IFFT(10, -2 + 2i, -2, -2 - 2i) returns [1, 2, 3, 4];
	IFFT([10, -2 + 2i, -2, -2 - 2i]) returns [1, 2, 3, 4];

The following example is for the above functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder:

```
Help
@language:
  test sign processing functions
```

```
@end

@language:simplified_chinese

  测试信号处理相关函数

@end

endh

function testSignalProc()

 print("\nconv([4,8,2,9],[5,3,8,9,6,7,8]) = " _

   + conv([4,8,2,9],[5,3,8,9,6,7,8]))

 print("\nconv([[4,8,2,9],[8,6,7,9],[2,2,8,-4]],[[-5,i,7],[0.6,8,4]]) = " _

   + conv([[4,8,2,9],[8,6,7,9],[2,2,8,-4]],[[-5,i,7],[0.6,8,4]]))

 print("\nFFT(1, 2, 3, 4) = " + FFT(1, 2, 3, 4))

 print("\nFFT([1,2,3,4]) = " + FFT([1,2,3,4]))

 print("\niFFT(10, -2 + 2i, -2, -2 - 2i) = " _

   + IFFT(10, -2 + 2i, -2, -2 - 2i))

 print("\niFFT([10, -2 + 2i, -2, -2 - 2i]) = " _

   + IFFT([10, -2 + 2i, -2, -2 - 2i]))

Endf
```

The above example gives the following results:

```
conv([4,8,2,9],[5,3,8,9,6,7,8]) = [20, 52, 66, 151, 139, 166, 181, 132, 79, 72]

conv([[4,8,2,9],[8,6,7,9],[2,2,8,-4]],[[-5,i,7],[0.6,8,4]]) = [[-20, -40 + 4i, 18 + 8i, 11 + 2i, 14 + 9i, 63], [-37.6, 6.8 + 8i, 102.2 + 6i, 50.4 + 7i, 129 + 9i, 99], [-5.2, 57.6 + 2i, 58.2 + 2i, 119.4 + 8i, 156 - 4i, 8], [1.2, 17.2, 28.8, 69.6, 0, -16]]

FFT(1, 2, 3, 4) = [10, -2 + 2i, -2, -2 - 2i]

FFT([1,2,3,4]) = [10, -2 + 2i, -2, -2 - 2i]

iFFT(10, -2 + 2i, -2, -2 - 2i) = [1, 2, 3, 4]

iFFT([10, -2 + 2i, -2, -2 - 2i]) = [1, 2, 3, 4]
```

Section 9 Factor, Is_prime and Roots Functions

Factor function calculates factorial of a non-negative integer. For example, factor(3) returns 6.

Factor function has only one parameter. If the parameter is not an integer, it will be truncated to an integer first. If the parameter is negative or cannot be converted to an integer, an error will be reported.

Is_prime function determines whether a real value is a prime or not. For example, is_prime(3.3) returns false while is_prime(97) returns true. Note that this function has only one parameter which must be a real value. If it is not a real value, an error will be reported.

Function roots returns all the roots of a polynomial. Its usage is roots(a, ...). If a is a vector includes N elements (whether real or complex), it returns all the roots (saved in an array) of equation

a[0] * x**(N-1) + a[1] * x**(N-2) + ... + a[N-2] * x + a[N-1] == 0

. If a is single value, then this function needs at least two parameters and returns all the roots (saved in an array) of equation

a * x**(number of parameters - 1) + (first parameter after a) * x**(number of parameters - 2) + … + (second last parameter) * x + (last parameter) == 0

. Note that if the degree of the polynomial is no smaller than 4, Newton Raphson method is used in this function to give out approximation of polynomial roots. This generally takes long time depending on the performance of hardware.

For example, to calculate roots of 3 * x**2 - 4 * x + 1 == 0, user inputs roots([3, -4, 1]) in Command Line and the result is [1, 0.33333333].

To calculate roots of (1+2i) * x**3 + (7-6i) * x**2 + 0.54 * x - 4.31 - 9i == 0, user inputs roots(1+2i, 7-6i, 0.54, -4.31-9i) and gets [0.79288607 + 3.9247084 * i, -0.56361748 - 0.78399569 * i, 0.7707314 + 0.85928729 * i].

Function roots gives the same result as a solve block. However, it doesn't analyze syntax of statement block so that much faster.

The following example is for the above functions. It can be found in the examples.mfps file in math libs sub-folder in the manual's sample code folder:

Help

@language:

```
 test prime, factor and roots functions
@end
@language:simplified_chinese
  测试质数、阶乘和一元多项式求根的相关函数
@end
endh
function PrimeFactRoots()
 print("\nis_prime(3.3) = " + is_prime(3.3))
 print("\nis_prime(97) = " + is_prime(97))
 print("\nis_prime(-97) = " + is_prime(-97))
 print("\nis_prime(1) = " + is_prime(1))
 print("\nis_prime(2) = " + is_prime(2))
 print("\nis_prime(0) = " + is_prime(0))
 print("\nis_prime(8633) = " + is_prime(8633))
 print("\nfact(3) = " + fact(3))
 print("\nfact(63) = " + fact(63))
 print("\nfact(0) = " + fact(0))
 print("\nroots([3, -4, 1]) = " + roots([3, -4, 1]))
 print("\nroots(1+2i, 7-6i, 0.54, -4.31-9i) = " _
   + roots(1+2i, 7-6i, 0.54, -4.31-9i))
Endf
```

Output of the above example is:

```
is_prime(3.3) = FALSE
is_prime(97) = TRUE
is_prime(-97) = FALSE
is_prime(1) = FALSE
```

is_prime(2) = TRUE

is_prime(0) = FALSE

is_prime(8633) = FALSE

fact(3) = 6

fact(63) =
19826083154044400641161467083618981375447736902272686281062795996127
29753600000000000000

fact(0) = 1

roots([3, -4, 1]) = [1,
0.3333333333333333333333333333333333333333333333333333333333333333]

roots(1+2i, 7-6i, 0.54, -4.31-9i) =
[0.7928860730571022099839052581254713102006395451929364570881977409 +
3.9247083954445877597678511535081536006180195929229657166716945562i, -
0.5636174763329694374664831861988500484145796260620965968434731734 -
0.7839956883798520996055087388165005568364007706209239481847346781i,
0.7707314032758672274825779280733787382139400808691601397552754324 +
0.8592872929352643398376575853083469562183811776979582315130401219i]

Summary

MFP programming language provides a complete list of functions for mathematical analysis and scientific calculation. These functions are handy and easy to use.

Among them, functions for expression calculation and calculus accept string based MFP expressions as parameters. Different from graphing functions to be introduced later, these functions are able to recognize predefined variables in an MFP expression. In this case, a predefined variable will be replaced by its value before any calculation.

Chapter 5 MFP Graphing Functions

In Chapter 1, this manual has demonstrated how to plot 2D/3D/polar graphs using Smart Calculator or Chart Plotter. Smart Calculator and Chart Plotter are handy and useful for users without programming experience because details of underlying graphing functions called by these tools are hidden from users. However, more complicated graphs can be flexibly plotted if user is able to program the graphing functions directly.

Like other functions, MFP graphing functions can be called both in Scientific Calculator for JAVA and the Command Line module in Scientific Calculator Plus for Android. And the plotted graphs will be exactly the same except that in a PC platform the graph would be wider. In order to save time, in this chapter all sample graphs are plotted by Scientific Calculator Plus for JAVA. However, user will get same graphs if copy the codes and run them in Scientific Calculator Plus for Android.

This chapter provides many graphing samples with only one MFP statement, i.e. calling an MFP graphing function. As such, only the statement of MFP graphing call instead of the whole function definition is included in the manual. And all these statements are part of none parameter function plotGraphs in the sample code. The sample code is located in the examples.mfps file in graph libs sub-folder in the manual's sample code folder.

Section 1 Plotting Graphs for Expressions

Plotting graph for expressions is quick and easy because user needs not to worry about the graph type (2D, 3D or polar), plotting range (which can be adjusted after graph is generated), colour, point and line style, and even the title and legends. All of these will be automatically determined by software.

The function to plot graph for expressions is plot_exprs whose usage is as below:

Function plot_exprs analyses at least one expression at most 8 expressions and draws 2D, polar or 3D curves based on the number of variables in the expression. The expression should be an equation, e.g. "4*x+9 == y +z**2" and "log(x*y) == x", or an assignment with an unknown variable on the left side, e.g. "k= 3+ 7 * sin(z)", or an expression which can be recognized as an assignment, e.g. "9*log(y)" can be looked on as "x = 9 * log(y)". Note that the total number of unknown variables in the expressions should be no more than 3 and each expression should include at most one unknown variable less than the total number of unknown variables. The initial range of each unknown variable is configurable, by default it is from -5 to 5 but user can adjust the range after the chart is plotted. If there are two unknown variables and one of the unknown variables is Greek letter α, β, γ or θ, instead of plotting a 2D chart, a polar graph is drawn. One example of this function is plot_exprs("4*x+sin(y)", "4-y**2==(x**2 + z**2)", "x*lg(x)/log2(z)==y"). Also note that when plotting a 2D expression which is actually an implicit function, this function can plot at most 4 root expressions; when plotting a 3D implicit function, it could be very slow because this function may solve all of the three variables and plot at most 2 root expressions for each of the variables. Because plot_exprs function is able to plot at most 8 curves, while a 3D implicit function may use up to 6 of them, user may see a too many curves to plot error if more than one implicit functions are included in the parameter list.

For example, user may want to draw the following two curves in one chart:

$$f(x) = \ln x + 2 x - 6$$

and

$$g(x) = x3 + 0.9 x$$

. Note that the two expressions are not implicit functions so that there is no need to input f(x) or g(x), only the right parts of the two expressions are required. However, MFP strictly requires (and) for a function call, as such has to be written as ln(x) + 2*x – 6, and has to be x**3 + 0.9*x because power operator is not ^ but ** in MFP. Finally, keep in mind that plot_exprs function only accepts string based parameters. And the whole function call statement should be:

```
plot_exprs("ln(x)+2*x-6", "x**3+0.9*x")
```

. Alternatively, user may call the function in the following way:

```
plot_exprs("y==ln(x)+2*x-6", "x**3+0.9*x==y")
```

, or

```
plot_exprs("y=ln(x)+2*x-6", "y=x**3+0.9*x")
```

. Both the second and the third calling approaches set a y variable which means the value of the expressions, i.e. f(x) and g(x). Using a unique y variable for f(x) and g(x) instead of two different variables (e.g. yf and yg) because user wants to draw curves for the two expressions in one 2D chart. If two variables are declared for f(x) and g(x) respectively, a 3D chart instead of a 2D chart will be drawn.

User may also notice that, in the second calling approach where equation (==) is used, y can be placed on both left side and right side of equation. Comparatively, if assignment is used, y can only appear on the left side of assignment. This limit matches the rule of mathematics where value can be assigned to a variable but not to an expression.

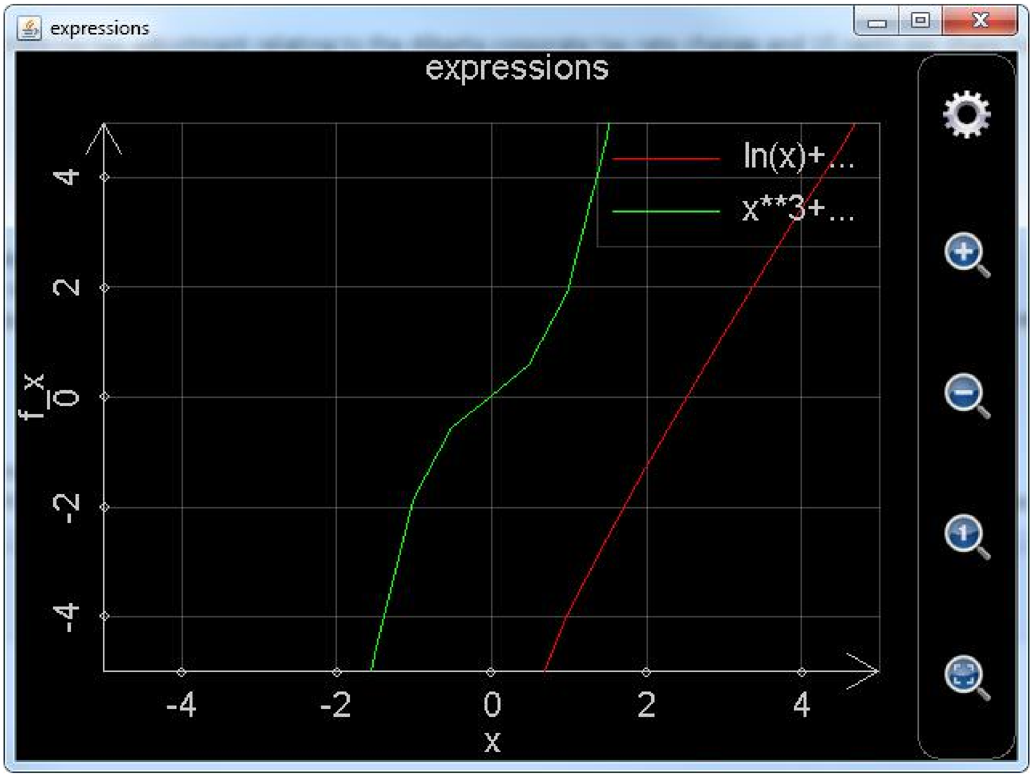
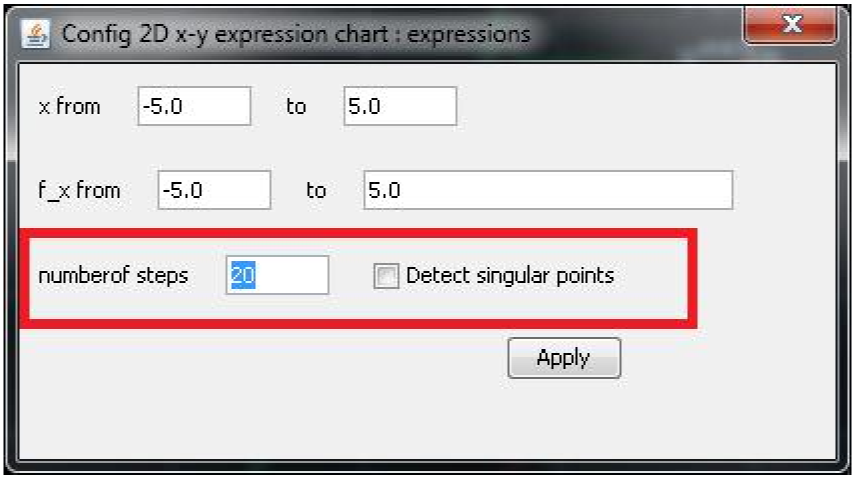


Figure 5.1: Plot_exprs function draws a 2D chart for expressions.

The plotted chart is shown above. User can drag the curves to shift, and click the buttons to zoom-in and zoom-out. All of these actions lead to change of plotting range and recalculation of some values. If the curves are sophisticated or include many points, response to user's operations in an Android device could be sluggish, depending on the hardware performance. But in a PC this will not be an issue.

Some users may complain that the green curve is not very smooth. This is simply because the number of steps when drawing the curve is too small. User can click the gear button and adjust the settings. In the following chart, left part of the red rectangle is the number of steps. More steps result in smoother curve and longer calculation time. By default, the number of steps is 20. Right part of the red rectangle is for singular points. Clearly, detecting singular points is time-expensive. By default, plot_exprs function does not detect singular points. However, when user uses Smart Calculator to plot curves for expressions, the default number of steps is 100 and singular points are detected. This ensures fine and accurate charts. This is also the reason that some Android users complain the slow calculation when using Smart Calculator to plot charts.



Config 2D x-y expression chart : expressions

x from -5.0 to 5.0

f_x from -5.0 to 5.0

numberof steps 20 □ Detect singular points

Figure 5.2: Settings of a 2D chart.

For another example, assume user wants to draw a chart for the following expressions:

$$r^2 = \alpha^2 + 9$$

$$r = \cos \alpha$$

$$\alpha = \sin r$$

, plot_exprs can be called in the following ways:

```
plot_exprs("r**2==α**2+9","r==cos(α)","α==sin(r)")
```

or

```
plot_exprs("r**2==α**2+9","cos(α)"," sin(r)")
```

. Here the left parts of r==cos(α) and α==sin(r) (i.e. variable and ==) can be neglected. MFP is able to analyze the three expressions and automatically complete an expression based on the total number of variables in the expressions. Comparatively any part of r**2==α**2+9 cannot be neglected because it is an implicit function.

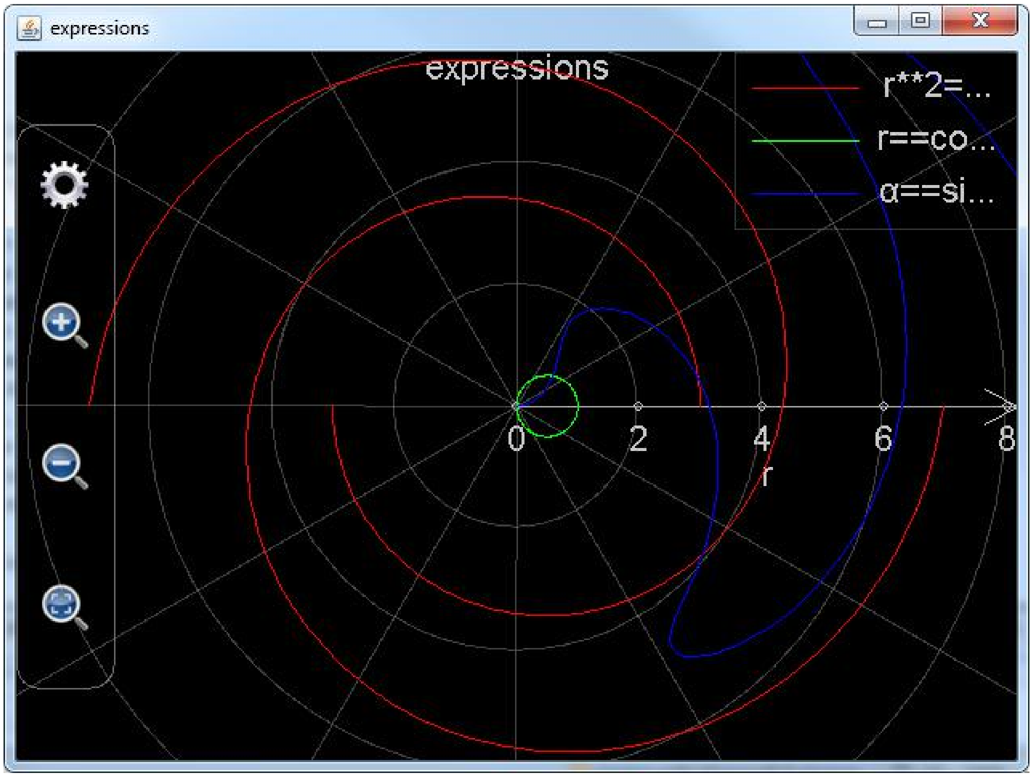


Figure 5.3: Plot_exprs function draws a polar chart for expressions.

The plotted graph is shown above. Because the total number of variables is two and one variable's name is Greek letter α, a polar chart is drawn. Note that the following chart has been zoomed in and the number of steps has been increased to 200. Otherwise the chart will be quite ugly.

Please also note that, when adjusting settings of a polar chart, range of radius (r) is adjustable, but angle (α)'s range is always from -2*pi to 2*pi because zooming a polar chart does not change the range of angle.

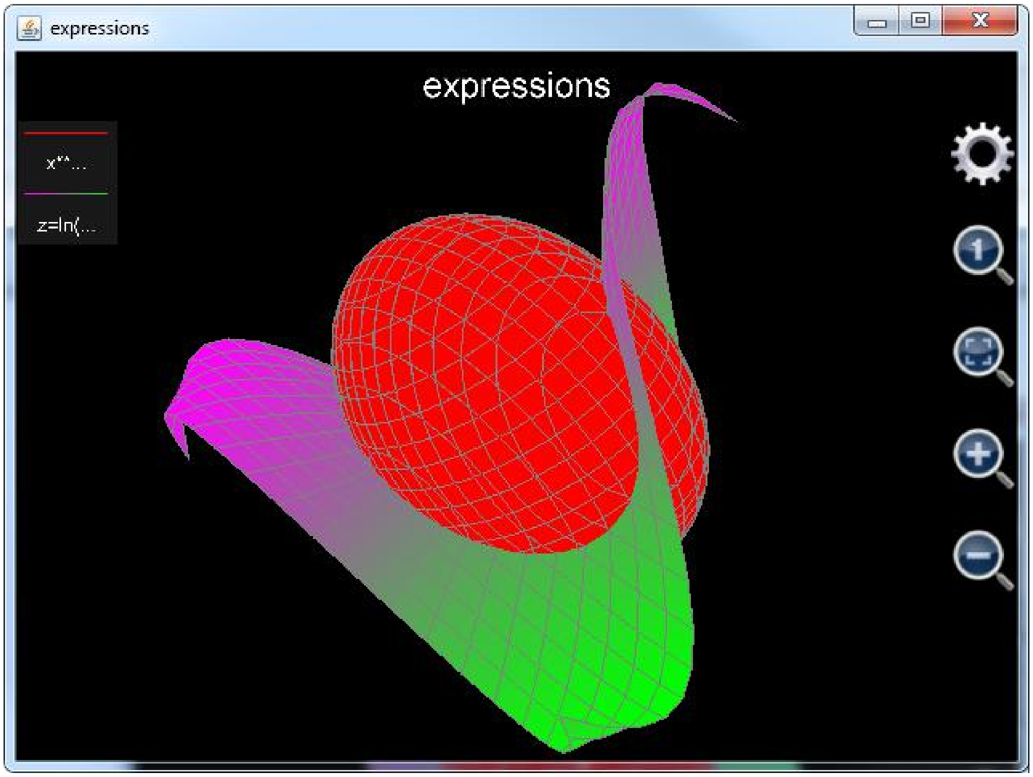


Figure 5.4: Plot_exprs function draws a 3D chart for expressions.

Assume user wants to draw a 3D chart including an ellipsoid:

$$x^2 + 2y^2 + z^2 = 20$$

, and a surface cutting the ellipsoid:

$$z = \ln(3x^2 + y^2 + 2y + 2)\sin(\frac{xy}{10})$$

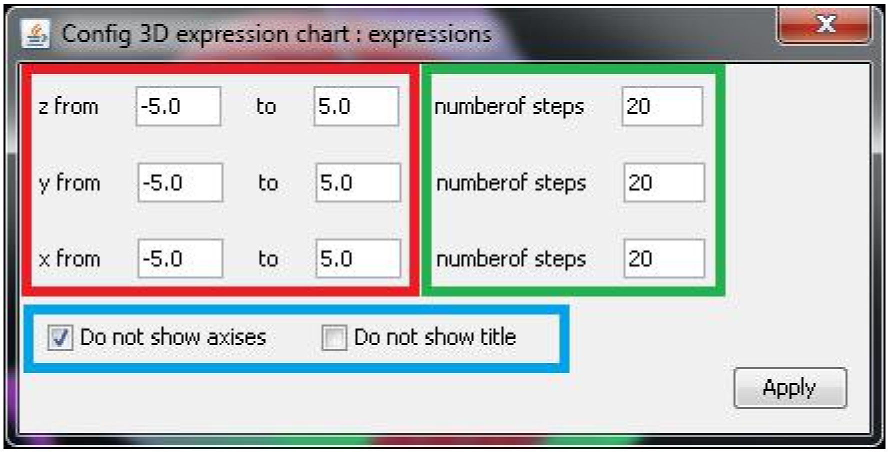
, the statement could be:

```
plot_exprs("x**2+2*y**2+z**2==20","z=ln(3*x**2+y**2+2*y+2)*sin(x*y/10)")
```

, or ignore the variable part in the explicit function:

```
plot_exprs("x**2+2*y**2+z**2==20","ln(3*x**2+y**2+2*y+2)*sin(x*y/10)")
```

. Similar to a 2D graph, user can zoom in and out the chart. However, dragging the chart will not shift but rotate it. Therefore, if user wants to set plotting range, the only approach is clicking the gear button to launch the configuration dialog. The configuration dialog is shown as below:



Config 3D expression chart : expressions

z from -5.0 to 5.0 numberof steps 20

y from -5.0 to 5.0 numberof steps 20

x from -5.0 to 5.0 numberof steps 20

☑ Do not show axises ☐ Do not show title

Figure 5.5: Settings of a 3D chart.

In the dialog, the red rectangle encloses the settings of plotting range along each axis; the green rectangle is the number of steps, i.e. number of points to draw, along each axis; the blue rectangle includes the switches to show/hide axes and title. Note that before revision 1.6.7 user is only able to show or hide axes and title together. From revision 1.6.7 axes and title can be shown or hidden independently. By default, title is shown but axes are hidden.

Also note that, plotted surface can only have one colour for implicit function. Otherwise, the colour of the plotted surface gradually changes from the smallest z value to the biggest z value. Nevertheless, user is not able to choose the colour in any case.

In the end of this section another 3D chart is provided to user:

```
plot_exprs("x**2-z**2==20","x**2-y**2==6")
```

. And the plotted chart would be (both axes and title have been hidden):

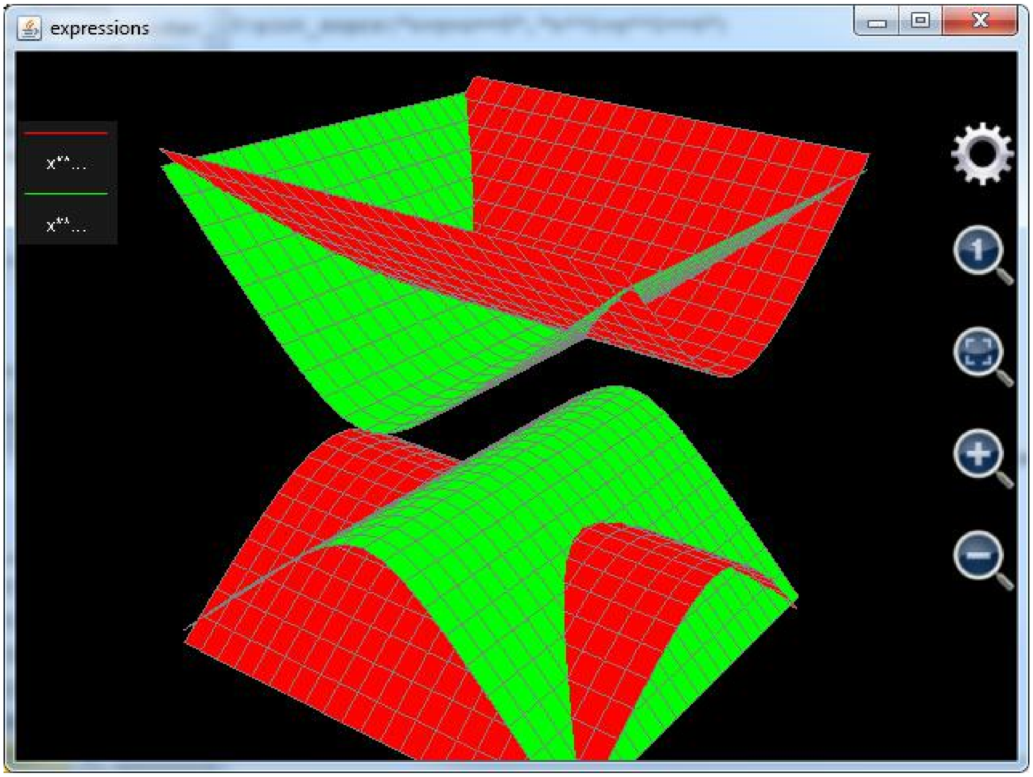


Figure 5.6: A 3D graph is plotted based on the number of variables.

. Note that if plot x**2-z**2==20 or x**2-y**2==6 separately, user will see a 2D curve. However, if plot the two expressions together in the same graph, the total number of variables is 3 (x, y and z) not 2 so that a 3D graph is generated.

Also note that function plot_exprs is written by MFP programming language. Parameters of plot_exprs are simply transferred to the underlying functions. Because of this reason, plot_exprs is not able to evaluate variables declared before calling string based parameters. This is different from the expression calculation and calculus functions introduced in Section 6 of Chapter 4. For example, assume user has declared a variable a and its value is 3. If user runs plot_exprs("x+a"), MFP will not look on "x+a" as "x+3". Instead, a will be treated as an unknown variable similar to x. Therefore, a 3D surface instead of a 2D line will be drawn. If user does hope to use variable's value in the string based parameter, s\he has to call function plot_exprs in the following way:

plot_exprs("x+"+a)

, and a's value is automatically appended to the string "x+" so that what is actually plotted is expression "x+3". Similarly, inside function plot_exprs only MFP default citingspaces are visible. Addtional visible citingspaces when plot_exprs is called cannot be transferred into the body of plot_exprs. This may lead to undefined function error if a function parameter of plot_exprs does not include absolute citingspace path. Assume, for example, user declares a citingspace named ::aaaaa. Inside it user implements a function named aaaaaF(). Then user adds using citingspace ::aaaaa before calling plot_exprs. This using statement cannot simplify the calling plot_exprs statement as aaaaaF()'s absolute citingspace path is still required. So,

Plot_exprs("::aaaaa::aaaaaF(x)")

will work while plot_exprs("aaaaaF(x)") will fail as ::aaaaa is not in the searching list of function plot_exprs.

Like function plot_exprs, many other graphing functions in MFP are implemented by MFP itself and call lower level JAVA implemented functions. As such the above approach (i.e. using "x+"+a instead of "x+3" and using absolute citingspace path) is strongly recommended.

Section 2 Plotting 2D Graphs in Normal Coordinates

MFP provides the following functions to draw 2D graphs in normal coordinates:

Function Name	Function Info
plot2dex	plot2dex(6...) : Function plot2DEX calls plot_multi_xy function to plot at most eight 2D-curves in one chart. It has the following parameters: 1. chart name (i.e. chart file name); 2. chart title; 3. X axis title; 4. Y axis title; 5. chart's background colour; 6. show grid or not; 7. curve title; 8. curve point colour; 9. curve point shape; 10. curve point size; 11. curve line colour; 12. curve line pattern; 13. curve line size; 14. t values start from; 15. t values end at; 16. t values' interval; 17. X's expression (with respect to variable t); 18. Y's expression (with respect to variable t)... Note that every new curve needs additional 12 parameters (i.e. parameters 7 to 18). At most 8 curves can be included. Also note that at this moment chart's background colour, curve point size, curve line colour and curve line pattern are not realized yet. And curve line size only has two values,

	i.e. zero means no connection line and non-zero means with connection line. An example of this function is plot2DEX(″chart 3″, ″3rd chart″, ″x″, ″y″, ″black″, true, ″cv1″, ″blue″, ″x″, 2, ″blue″, ″solid″, 1, -5, 5, 0.1, ″t″, ″t**2/2.5 - 4*t + 6″, ″cv2″, ″red″, ″square″, 4, ″square″, ″solid″, 1, -10, 10, 0.1, ″5*sin(t)″, ″10*cos(t)″) .
plot_2d_curves	plot_2d_curves(6...) : Function plot_2d_curves plots at most 1024 2D-curves in one chart. It has the following parameters: 1. chart name (i.e. chart file name); 2. chart title; 3. X axis title; 4. Y axis title; 5. chart’s background colour; 6. show grid or not (string ″true″ or string ″false″); 7. curve title; 8. curve point colour; 9. curve point shape; 10. curve point size; 11. curve line colour; 12. curve line pattern; 13. curve line size; 14. internal variable’s name (generally it is ″t″); 15. internal variable’s value starts from; 16. internal variable’s value ends at; 17. internal variable’s value changing interval; 18. X’s expression (with respect to the internal variable); 19. Y’s expression (with respect to the internal variable)... Note that every new curve needs additional 13 parameters (i.e. parameters 7 to 19). At most 1024 curves can be included. Also note that at this moment chart’s background colour, curve point size, curve line colour and curve line pattern are not realized yet. And curve line size only has two values, i.e. zero means no connection line and non-zero means with connection line. An example of this function is plot_2d_curves(″chart 3″, ″3rd chart″, ″x″, ″y″, ″black″, ″true″, ″cv1″, ″blue″, ″x″, 2, ″blue″, ″solid″, 1, ″t″, -5, 5, 0.1, ″t″, ″t**2/2.5 - 4*t + 6″, ″cv2″, ″red″, ″square″, 4, ″square″, ″solid″, 1, ″t″, -10, 10,

	0.1, ″5*sin(t)″, ″10*cos(t)″) .
plot_2d_data	plot_2d_data(16) : Function plot_2d_data analyses at least one at most eight groups of data lists and each data group will be plotted as one curve. The number of parameters in these function can be 1 (one curve), 2 (one curve), 4 (two curves), 6 (three curves), 8 (four curves), 10 (five curves), 12 (six curves), 14 (seven curves) and 16 (eight curves). Each parameter is a data list (i.e. 1-D data array). If only one parameter, each element value in the parameter will be the y value of a point in the curve, x value of the point starts from 1 and added by 1 at each point, otherwise, the odd number of parameters are the x values of the points and the even number of parameters are the y values. Note that the size of x value parameter must match the size of y value parameter. For example, plot_2d_data([5.5, -7, 8.993, 2.788]) or plot_2d_data([2.47, 3.53, 4.88, 9.42], [8.49, 6.76, 5.31, 0.88], [-9, -7, -5, -3, -1], [28, 42, 33, 16, 7]).
plot_multi_xy	plot_multi_xyz(2...) : plot_multi_xyz(at least 2 parameters) plots a 3D chart which includes at most 1024 surface curves. Parameters 1 and 2 are chart name and settings respectively. The chart settings parameter is a string like ″chart_type:multiXYZ;chart_title:This is a graph;x_title:x axis;x_min:-24.43739154366772;x_max:24.712391543667717;x_labels:10;y_title:Y axis;y_min:-251.3514430737091;y_max:268.95144307370913;y_labels:10;z_title:Z axis;z_min:-1.6873277335234405;z_max:1.7896774628184482;z_labels:10″. Note that chart_type session should

	always be multiXYZ, and x_labels, y_labels and z_labels means how many scale marks are in the x, y and z axises respectively. From parameter 3, every four parameters define a curve. Among the four parameters, the first describes curve settings, the second is an array of x values, the third is an array of y values and the fourth is an array of z values. An example of curve settings parameter is ″curve_label:cv2;is_grid:true;min_color:blue;min_color_1:cyan;min_color_value:-2.0;max_color:white;max_color_1:yellow;max_color_value:2.0″. Note that the dimensions of x, y and z arrays should be equal and they should only include real value elements. This function returns nothing. An example of this function is plot_multi_xyz(″chartII″, ″chart_type:multiXYZ;chart_title:This is a graph;x_title:x;x_min:-5;x_max:5;x_labels:6;y_title:Y;y_min:-6;y_max:6;y_labels:3;z_title:Z;z_min:-3;z_max:1;z_labels:4″, ″curve_label:cv1;min_color:blue;min_color_1:green;max_color:yellow;max_color_1:red″, [[-4, -2, 0, 2, 4],[-4, -2, 0, 2, 4],[-4, -2, 0, 2, 4]], [[-5, -5, -5, -5, -5], [0, 0, 0, 0, 0], [-5, -5, -5, -5, -5]], [[-2.71, -2.65, -2.08, -1.82, -1.77], [-2.29, -2.36, -1.88, -1.45, -1.01], [-1.74, -1.49, -0.83, -0.17, 0.44]]) .

Functions plot2dex and plot_2d_curves draw 2D curves for expressions in the selected plotting range. Function plot_2d_data plots 2D data graph. Function plot_multi_xy is very low-level and is called by plot2dex and plot_2d_data.

Note that the API plot_2d_curves is exposed to user from revision 1.6.7. This function did exist before revision 1.6.7. However, its old version was only capable of drawing at most 8 curves. Since revision 1.6.7, plot_2d_curves is able to draw as many as 1024 curves. Because plot_2d_curves is implemented by JAVA, it is much faster than MFP implemented function plot2dEx. Thus it is strongly recommended to use plot_2d_curves instead of plot2dEx.

The above four functions, as well as all the other functions to be introduced in this Chapter, share a big difference from the function plot_exprs. User needs not to set plotting range when calling plot_exprs. Plotting range is dynamically adjusted after graph is generated when user shifts or zooms the graph. Comparatively, the above four functions, as well as all the other functions to be introduced in this Chapter, need user's input of plotting range before drawing the graph. Plotting range is fixed after the graph is generated. Shifting or zooming the graph will not lead to any plotting range change and no recalculation is required. As such, operations on this kind of graphs are very smooth and swift.

Plot_2d_curves function is actually called by the independent 2D Chart Plotter in Scientific Calculator Plus for Android. Its first six parameters respectively set the chart file name (.mfpc extension will be appended automatically), chart title, x-axis name, y-axis name, background colour and drawing grid or not. All these parameters are strings. In particular, drawing grid or not is a string based Boolean value ("true" or "false"), and background colour should be "white ", "black", "red", "green", "blue", "yellow", "cyan", "magenta", "dkgray" (dark gray) and "ltgray" (light gray). By default the background colour is "black".

From the 7th parameter, a curve is defined by every 13 parameters which are

1. Curve name (string based value);

2. Point colour (string based value, available choices are the same as background colour);

3. Point shape (string based value, available choices are "point", "circle", "triangle", "square", "diamond" and "x" (cross));

4. Point size (a positive integer. This parameter hasn't been implemented so that user can assign any positive integer to it at this moment);

5. Colour of connecting lines between points (string based value, available choices are the same as background colour);

6. Style of connecting lines between points (string based value. This parameter hasn't been implemented so that user just set it "solid" at this moment);

7. Thickness of connecting lines between points (a non-negative integer. If it is zero, the connecting lines will not be drawn);

8. Internal variable name. Generally it is "t";

9. Starting point of varying range of variable t (if internal variable name is "t", otherwise it is a different name). Here t is the same t used by the independent Chart Plotter in Scientific Calculator Plus for Android. User

may refer to Section 4 of Chapter 1 for usage details. Also note that starting point of varying range must be a real value.

10. Ending point of varying range of variable t. It also must be a real value;

11. Varying step length of t. It must be a real value and it equals varying range / number of steps. User may set it zero which means the step length is automatically determined by software;

12. Function of X value with respect to t (string based expression. User may refer to Section 4 of Chapter 1 for details of X(t));

13. Function of Y value with respect to t (string based expression. User may refer to Section 4 of Chapter 1 for details of Y(t));

Because plot_2d_curves function can draw at most 1024 curves, its parameters can be as many as 6+13*1024 = 13318. In fact, user can program the calling statement of plot_2d_curves function to draw very complicated graphs.

For example, assume user wants to draw an ellipse and a parabola. The function of ellipse is:

$$4\ x^2 + y^2 = 16$$

, the function of parabola is:

$$y = \frac{x^2}{2.5} - 4x + 6$$

. To draw the ellipse, user may let x equal 2*cos(t), y equal 4*sin(t), and t is from 0 to 2*pi with a step length equal to 0.02*pi. To draw the parabola, let x be t, y be t**2/2.5-4*t+6, t is from -5 to 5 with a step length equal to 0.3. The whole statement is:

```
Plot_2d_curves("chart 1", "plo2dEx chart", "x", "y", "black", "true", "cv1", "red", "diamond", 3,
"blue", "solid", 1, "t", -5, 5, 0.3, "t", "t**2/2.5 - 4*t + 6 ", "cv2", "green", "point", 2, "green",
"solid", 2, "t", 0, 2*pi, 0.02*pi, "2*cos(t)", "4*sin(t)")
```

. The plotted graph is shown below. User can drag and zoom the chart without any sluggishness. However, there is no point outside the plotting range, i.e. x <-5 or x > 5, though the parabola should extend from –inf to inf, far beyond the plotting range.

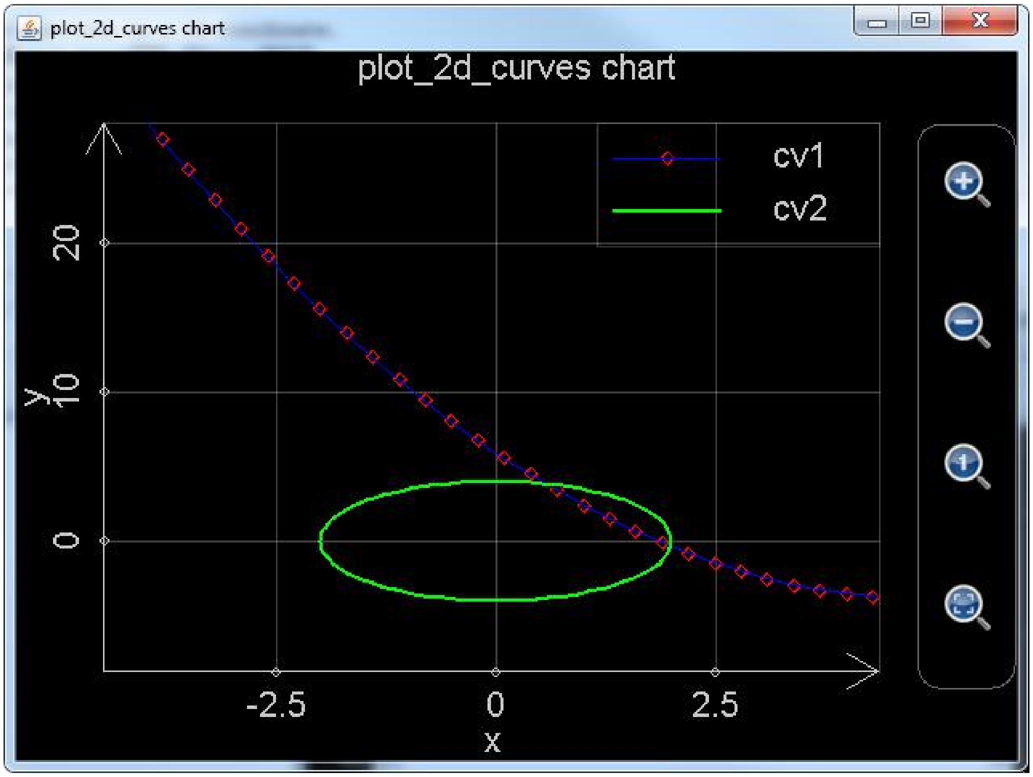


Figure 5.7: Function plot_2d_curves draws 2D curves in the selected plotting range.

Also note that the generated graph is automatically saved by Scientific Calculator Plus so that user can reopen it later on. If this graph is created in Android, user can start Scientific Calculator Plus for Android, tap the "Chart Manager" icon and go into the chart folder. Because the chart file name in the above example is "chart 1", the created chart file should be chart 1.mfpc. User needs to locate this file, and long press it to open. If the graph is created in a PC, it should be saved in the AnMath/charts folder where AnMath folder is the location of Scientific Calculator Plus for JAVA. To open the graph, user needs to start GUI based Scientific Calculator Plus for JAVA, then select "Tools" menu, then click "View Chart" sub-menu or simply press Ctrl-O key, and select the file in the Open dialog box.

For another example, assume that user wants to draw a regular triangle. The functions of the three legs of a regular triangle are:

$$y = \sqrt{3}x + 2 \text{...........} (-\sqrt{3} \le x \le 0)$$

$$y = -\sqrt{3}x + 2 \text{..........} (0 \le x \le \sqrt{3})$$

and

$$y = -1 \text{.................} (-\sqrt{3} \le x \le \sqrt{3})$$

. User can call plot_2d_curves to draw the three line segments. The first line segment settings are x=t and y=sqrt(3)*t+2, where t is from –sqrt(3) to 0 with a step length equal to 0.02; the second line segment settings are x=t and y=-sqrt(3)*t+2, where t is from 0 to sqrt(3) with a step length equal to 0.02; the third line segment settings are x=t and y=-1, where t is from -sqrt(3) to sqrt(3). The whole statement is:

```
plot_2d_curves("char 2", "plot_2d_curves chart", "x", "y", "black", "true", "cv1", "red", "point", 3, "red", "solid", 1, "t", -sqrt(3), 0, 0.02, "t", "sqrt(3)*t+2", "cv2", "green", "point", 3, "green", "solid", 1, "t", 0, sqrt(3), 0.02, "t", "-sqrt(3)*t+2", "cv3", "blue", "point", 3, "blue", "solid", 1, "t", -sqrt(3), sqrt(3), 0.02, "t", "-1")
```

. The plotted graph is shown below:

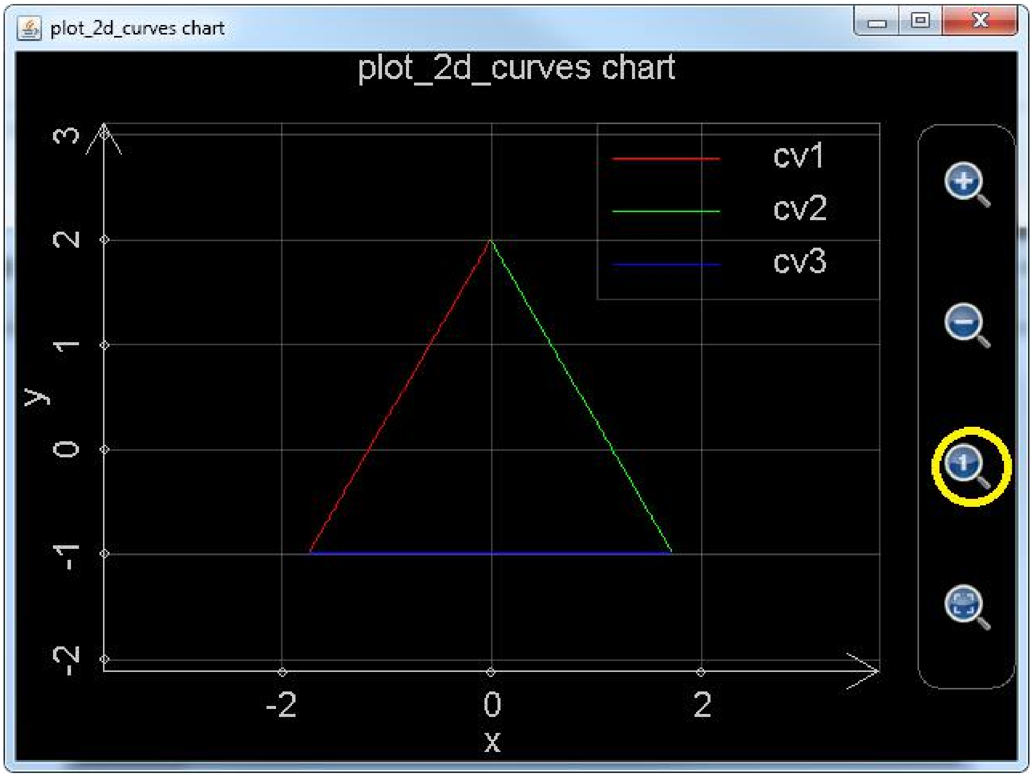


Figure 5.8: Using function plot_2d_curves to draw a regular triangle.

Note that, when the graph is created, it does not look like a regular triangle because the unit length of x axis is not equal to the unit length of y axis. User can click the button in the yellow circle and unit length ratio of the two axes will be automatically adjusted to 1:1. Then the shape of the triangle becomes regular.

Function plot_2d_data draws 2D data chart. Every parameter of this function must be a 1-D array and elements in the array must be real numbers. If there is only one parameter, the first element in the parameter is the point's y value when x is 1, the second element in the parameter is the point's y value when x is 2, etc.

The total number of points in this case equals the number of elements in the array. For example, the following statement draws a line chart connecting the points (1, 1), (2, 7), (3, 8) and (4, 6):

```
Plot_2d_data([1,7,8,6])
```

, the plotted graph is:

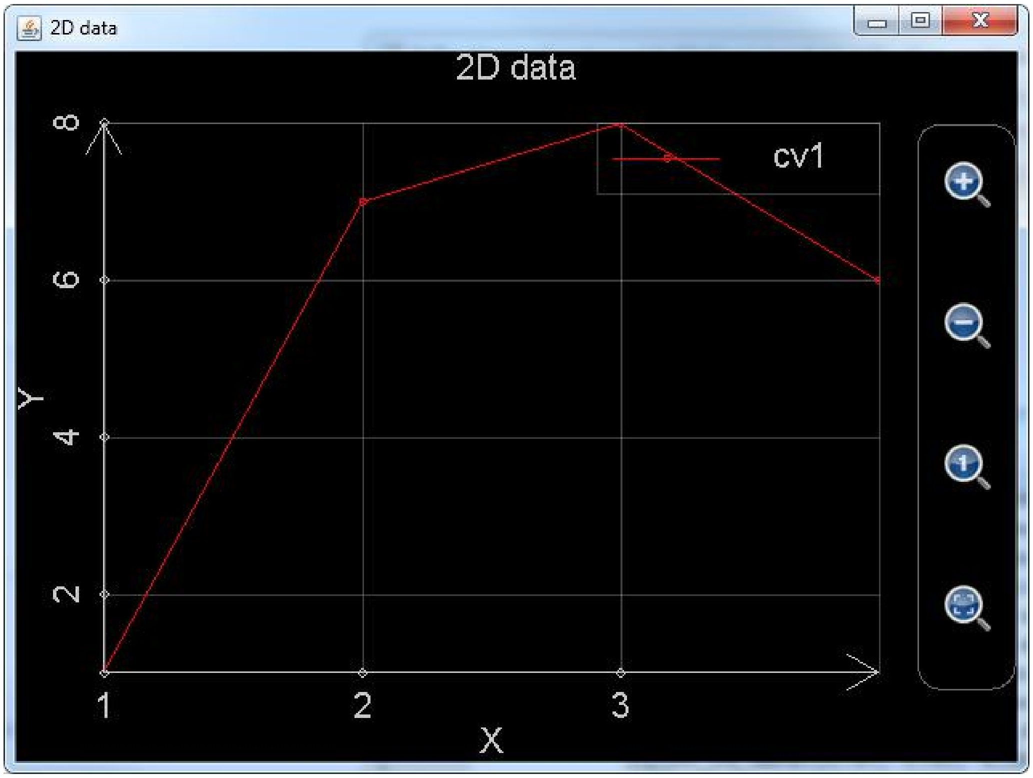


Figure 5.9: Using function plot_2d_data with single parameter to draw a 2D line chart.

If, however, there are more than one parameters, the number of parameters must be even, and grouped in pairs. The first parameter in each pair stores the x values of all the points in a curve and the second parameter stores all the y values. For example, if user wants to draw a curve with the points (-1.71, 6.24), (8.93, -7.08), (3.11, 5.85), (4.28, -5.76) and (5.99, -3.24), the two parameters should be [-1.71, 8.93, 3.11, 4.28, 5.99] and [6.24, -7.08, 5.85, -5.76, -3.24].

The following statement draws two curves (data sets) including different numbers of points. The second data set includes a point whose y value is Nan. This implies that the second curve is broken at this point because Nan cannot be plotted:

```
Plot_2d_data([-1.71, 8.93, 3.11, 4.28, 5.99], [6.24, -7.08, 5.85, -5.76, -3.24], [1.88, 2.41, 5.71, 7.66,
12.47, 15.19], [-3.69, 2.12, -1.74, Nan, 2.98,8.71])
```

. The plotted graph is shown below:

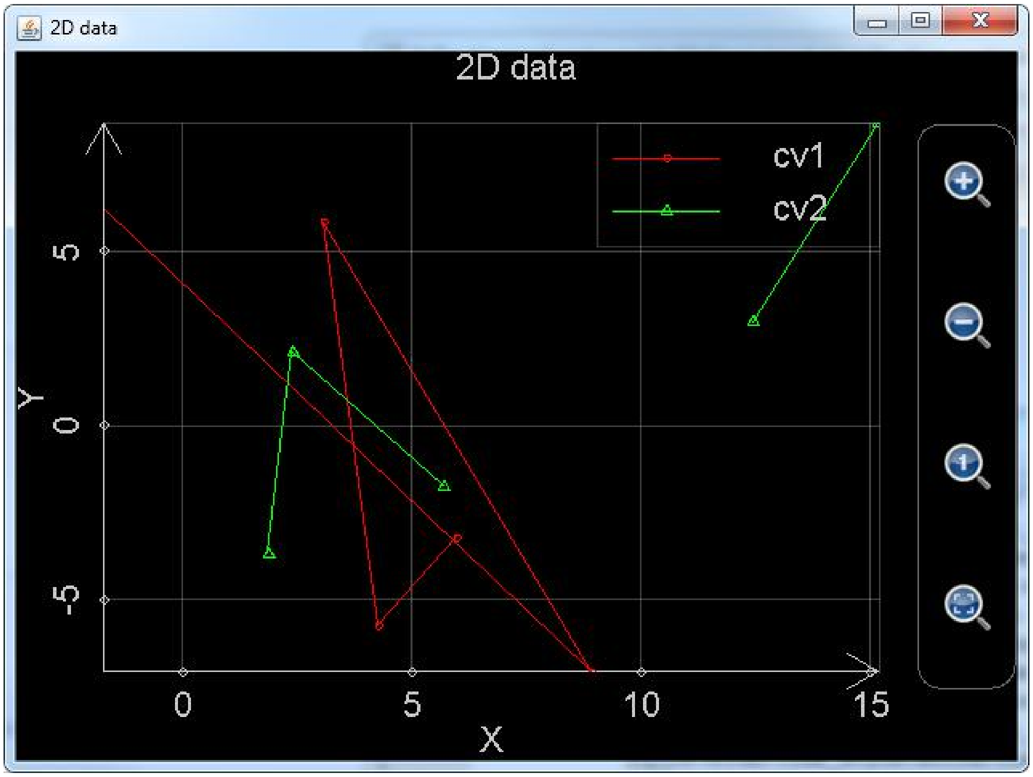


Figure 5.10: Using function plot_2d_data to draw two curves. Note that the green curve is broken at a point.

User can drag and zoom the charts created by plot_2d_data function. However, user cannot set the colour of the curve or the style or size of the points and connecting lines. And the chart will not be automatically saved.

Function plot_multi_xy is a low level function. Similar to plot_2d_data, this function accepts data values instead of expressions as parameter. Function plot_2d_data calls this function directly while function plot2dEx first calculates values of the to-be-plotted expressions at each point and then calls this function.

The first parameter of plot_multi_xy is string based chart name, i.e. chart file name without .mfpc extension.

The second parameter is chart level settings. All the configurations at chart level are included in the string based parameter. Each configuration follows the format of

Setting's name:setting's value;

. All the configurations are then concatenated into a string, like:

"chart_type:multiXY;chart_title:1 chart;x_title:x;x_min:-6.2796950076838645;x_max:6.918480857169536;x_labels:10;y_title:y;y_min:-

4.487378580559947;y_max:4.1268715788884345;y_labels:10;background_color:black;show_grid:true"

, here chart_type is the type of chart, which must be multiXY, chart_title is the title of the chart, x_title is the name of x-axis, x_min is the minimum value x-axis shows when the chart is plotted, x_max is the maximum value x-axis shows when the chart is plotted, x_label is the number of scale marks in x-axis, y_title is the name of y-axis, y_min is the minimum value y-axis shows when the chart is plotted, y_max is the maximum value y-axis shows when the chart is plotted, y_label is the number of scale marks in y-axis, background_color is the colour of background, and show_grid is a flag to determine showing grid in the chart or not.

The third parameter of plot_multi_xy is settings for a single curve. Like the previous parameter, all the configurations are included in a string and each of them follows the format of

Setting's name:setting's value;

. For example:

"curve_label:cv2;point_color:blue;point_style:circle;point_size:3;line_color:blue;line_style:solid;line_size:1"

, where curve_label is the title of the curve, point_color is the colour of the points, point_style is the shape of the points (square, circle etc.), point_size is the size of the points (this parameter hasn't been realized yet so that user can simply assign an arbitrary positive integer to it), line_color is the colour of the connecting lines, line_style is the style of the connecting lines (this parameter hasn't been realized yet so that user can simply set it solid), and line_size is the thickness of the connecting lines which must be a non-negative integer.

The fourth parameter of plot_multi_xy is the x values of all the points in this curve. Note that this parameter must be a 1-D array and each element in the array must be a real number. If an element is Nan, like function plot_2d_data, the curve is broken at the Nan point.

The fifth parameter of plot_multi_xy is the y values of all the points in this curve. Note that this parameter must be a 1-D array with the same size as parameter four, and each element in the array must be a real number. If an element is Nan, like function plot_2d_data, the curve is broken at the Nan point.

If user wants to draw more than one curves, the additional curves need their own parameter groups and each group includes parameters 3, 4 and 5. User can draw at most 1024 curves (or at most 8 curves if using revision 1.6.6 or earlier). Therefore the max number of parameters of this function is 2 + 1024*3 = 3074.

An example of this function is:

```
plot_multi_xy("chart2", "chart_type:multiXY;chart_title:1 chart;x_title:x;x_min:-6;x_max:6;x_labels:6;y_title:y;y_min:-4;y_max:4;y_labels:5;background_color:black;show_grid:true", "curve_label:cv2;point_color:blue;point_style:circle;point_size:3;line_color:blue;line_style:solid;line_size:1", [-5, -3, -1, 0, 1, 2, 3, 4, 5], [-3.778, -2.9793, -2.0323, -1.1132, 0.2323, 1.2348, 3.9865, 2.3450, 0.4356])
```

. The plotted curve is shown below. Note that because the chart file name is given as a parameter, the generated chart will be saved automatically into a .mfpc file so that user can reopen it sometime later on.

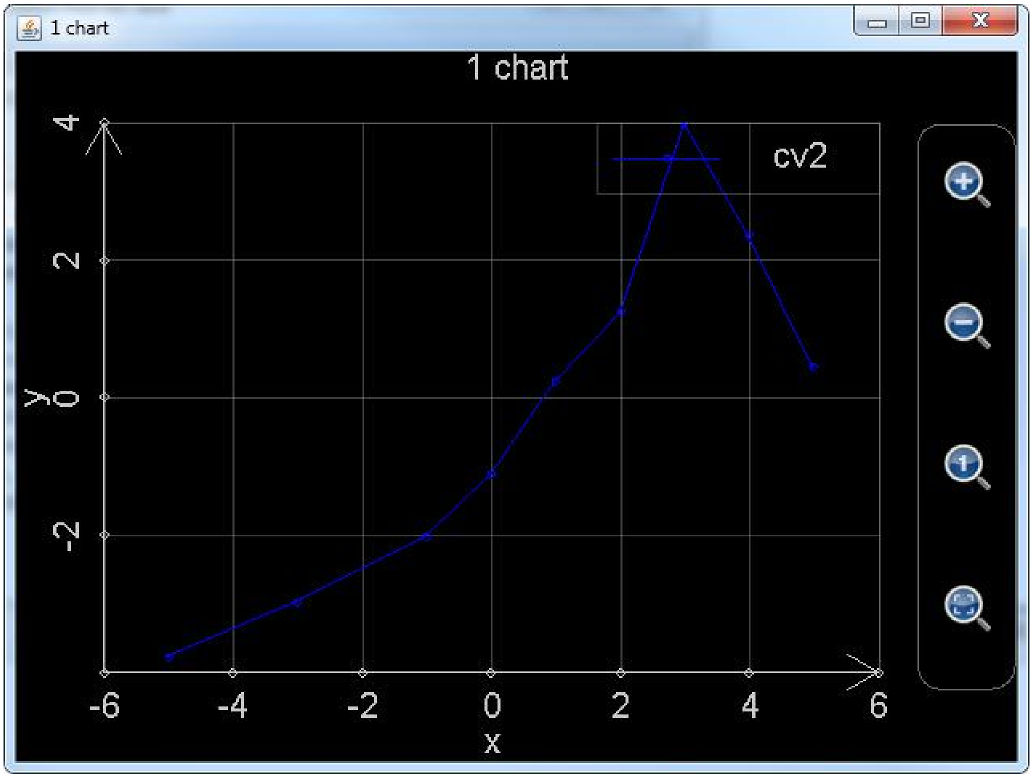


Figure 5.11: Using function plot_multi_xy to draw 2D chart.

Section 3 Plotting Polar Graphs

As shown below, MFP programming language provides user a list of functions to plot charts in a system of polar coordinates.

Function Name	Function Info
plot_polar	plot_polar(6...) : Function plot_polar calls plot_multi_xy function to plot at most eight polar-curves in one chart. It has the following parameters: 1. chart name (i.e. chart file name); 2. chart

	title; 3. R axis title; 4. Angle axis title; 5. chart’s background colour; 6. show grid or not; 7. curve title; 8. curve point colour; 9. curve point shape; 10. curve point size; 11. curve line colour; 12. curve line pattern; 13. curve line size; 14. t values start from; 15. t values end at; 16. t values’ interval; 17. R’s expression (with respect to variable t); 18. Angle’s expression (with respect to variable t)... Note that every new curve needs additional 12 parameters (i.e. parameters 7 to 18). At most 8 curves can be included. Also note that at this moment chart’s background colour, curve point size, curve line colour and curve line pattern are not realized yet. And curve line size only has two values, i.e. zero means no connecting line and non-zero means with connecting line. An example of this function is plot_polar("chart 3", "3rd chart", "R", "Angle", "black", true, "cv1", "blue", "point", 0, "yellow", "solid", 1, -5, 5, 0.1, "cos(t)", "t", "cv2", "red", "square", 4, "green", "solid", 1, 0, PI*2.23, PI/10, "5*sqrt(t)", "t + PI") .
plot_polar_curves	plot_polar_curves(6...) : Function plot_polar_curves plots at most 1024 polar-curves in one chart. It has the following parameters: 1. chart name (i.e. chart file name); 2. chart title; 3. R axis title; 4. angle title (actually angle title is never shown); 5. chart’s background colour; 6. show grid or not (string "true" or string "false"); 7. curve title; 8. curve point colour; 9. curve point shape; 10. curve point size; 11. curve line colour; 12. curve line pattern; 13. curve line size; 14. internal variable’s name (generally it is "t"); 15. internal variable’s value starts from; 16.

	internal variable's value ends at; 17. internal variable's value changing interval; 18. R's expression (with respect to the internal variable); 19. angle's expression (with respect to the internal variable)... Note that every new curve needs additional 13 parameters (i.e. parameters 7 to 19). At most 1024 curves can be included. Also note that at this moment chart's background colour, curve point size, curve line colour and curve line pattern are not realized yet. And curve line size only has two values, i.e. zero means no connecting line and non-zero means with connecting line. An example of this function is plot_polar_curves(″chart 3″, ″3rd chart″, ″R″, ″angle″, ″black″, ″false″, ″cv1″, ″blue″, ″x″, 2, ″blue″, ″solid″, 1, ″t″, -5, 5, 0.1, ″t″, ″t**2/2.5 - 4*t + 6″, ″cv2″, ″red″, ″square″, 4, ″square″, ″solid″, 1, ″t″, -10, 10, 0.1, ″5*sin(t)″, ″10*cos(t)″) .
plot_polar_data	plot_polar_data(16) : Function plot_polar_data analyses at least one at most eight groups of data lists and each data group will be plotted as one polar curve. The number of parameters in these function can be 2 (one curve), 4 (two curves), 6 (three curves), 8 (four curves), 10 (five curve), 12 (six curves), 14 (seven curves) and 16 (eight curves). Each parameter is a data list (i.e. 1-D data array). The odd number of parameters are the R values of the points and the even number of parameters are the angle values. Note that the size of R value parameter must match the size of angle value parameter. For example, plot_polar_data([2.47, 3.53, 4.88, 9.42], [8.49, 6.76, 5.31, 0.88], [-9, -7, -5, -3, -1], [28, 42, 33, 16, 7]).

Clearly, functions plot_polar, plot_polar_curves and plot_polar_data corresponds to the functions introduced in last section, i.e. plot2dEx, plot_2d_curve and plot_2d_data. Even their parameters exactly match. Similar to plot_2d_curves, plot_polar_curves is implemented by JAVA and can plot as many as 1024 curves. Its performance and plotting capability are much better than function plot_polar, which is implemented by MFP script, so that it is going to replace plot_polar gradually.

Difference between plot_polar_curves and plot_2d_curves is that the third parameter of plot_polar_curves is the name of radius axis while it is the name of x axis for plot_2d_curves; the fourth parameter of plot_polar_curves is the name of angle while it is the name of y axis for plot_2d_curves. Defining a curve in plot_polar_curves also needs 13 parameters. However, the 12th parameter is the function of radius with respect to internal variable (usually it is t), and the last parameter is the function of angle with respect to internal variable. Comparatively, in plot_2d_curves they are the function of x and the function of y respectively.

The following example calls plot_polar_curves to draw lotus and butterfly. The expression of lotus is

$$r = \sin\theta + \sin3\ 2.5\theta \ldots\ldots\ldots\ldots 0 \le \theta \le 4\pi$$

. User may set the function of θ (angle) to be t, the function of r, i.e. radius, to be r(t)==sin(t)+sin(2.5*t)**3, and let t vary from 0 to 4*pi with a 0.05 step length.

The expression of butterfly is

$$r = 0.6e^{\sin\theta} - 2\cos 4\theta + \sin^5 \frac{2\theta - \pi}{24}$$

. User may set the function of θ (angle) to be t, the function of r, i.e. radius, to be r(t)==0.6*exp(sin(t))-2*cos(4*t)+sin((2*t-pi)/24)**5, and let t vary from -pi to pi with a 0.02 step length.

The whole function call statement is:

```
plot_polar_curves("LotusAndButterfly", "Lotus & Butterfly", "R", "Angle", "black", "true",
"Lotus", "yellow", "point", 0, "red", "solid", 3, "t", 0, 4*pi, 0.05, "sin(t)+sin(2.5*t)**3", "t",
"Butterfly", "green", "circle", 4, "blue", "solid",1,"t",-pi, pi,0.02,"0.6*exp(sin(t))-
2*cos(4*t)+sin((2*t-pi)/24)**5","t")
```

. The plotted graph is:

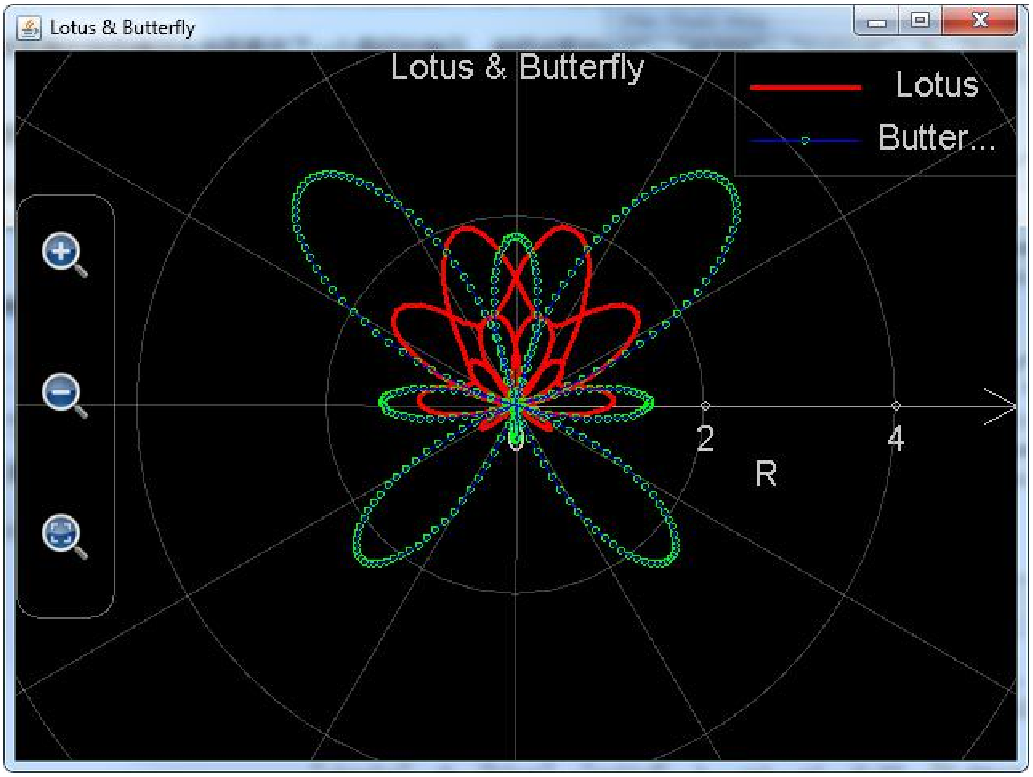


Figure 5.12: Using function plot_polar_curves to draw lotus and butterfly in the system of polar coordinates.

Similar to plot_2d_curves, the plotted chart will be saved as a file. The file name is the first parameter of plot_polar_curves with a .mfpc extension. In the above case it is LotusAndButterfly.mfpc. The location of the file is in AnMath\charts folder. User can open the file in Scientific Calculator Plus.

Function input of plot_polar_data is also very similar to plot_2d_data. The only difference is that plot_polar_data cannot accept only one parameter, i.e. plot_polar_data's parameters must be grouped in pairs. Each parameter pair defines a set of points for a curve. They are both 1D arrays. Elements in the arrays must be real value or Nan. And the two parameters in the pair must have same number of elements. The elements of the first parameter are radius of each point, and the elements of the second parameter are the angle.

The following example demonstrates user how to plot data chart in the system of polar coordinates. In this example, the first data set includes four points and the second data set includes six points. Please note that one point has Nan radius in data set two so that the second curve is broken at this point.

```
plot_polar_data([2.47, 3.53, 4.88, 9.42], [8.49, 6.76, 5.31, 0.88], [-9, -7, Nan, -3, -1, 1], [28, 42, 33, 16, 7, 0])
```

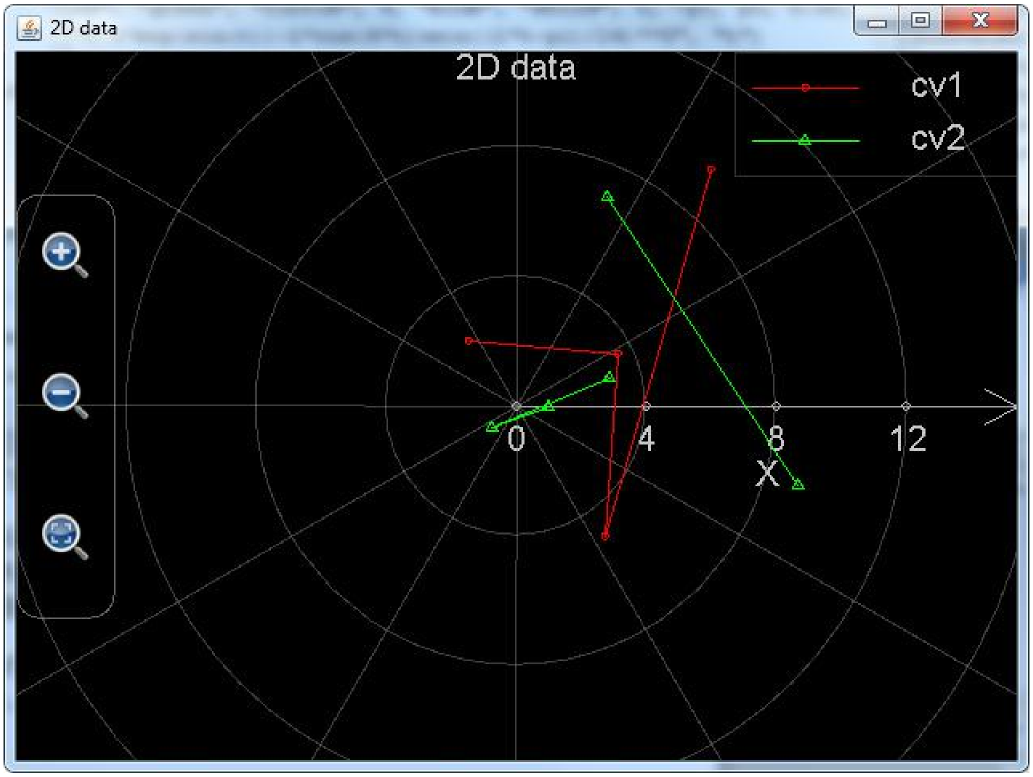


Figure 5.13: Using function plot_polar_data to plot data chart in the system of polar coordinates.

Functions plot_polar and and plot_polar_data are both implemented by MFP scripts. They both call the plot_multi_xy function introduced in last section. The usage of plot_multi_xy has been presented in detail. This function needs at least two parameters to draw 2D or polar chart. Each chart can include as many as 1024 curves. First parameter is chart name, second one is string-based chart level settings, e.g. "chart_type: multiRangle;chart_title:1 chart;x_title:x;x_min:-6.2796950076838645;x_max:6.918480857169536;x_labels:10;y_title:y;y_min:-4.487378580559947;y_max:4.1268715788884345;y_labels:10;background_color:black;show_grid:true". Note that chart_type here is no longer multiXY for 2D chart but multiRangle for polar chart. X_labels is the number of scale marks in r axis. Y_labels is meaningless for polar chart because angle's scale marks in a polar chart are always 8. As such y_labels can be any positive integer.

So what happens if user runs example of plot_multi_xy in last section but changes chart_type from multiXY to multiRangle? User may try the following statement:

```
plot_multi_xy("chart2", "chart_type:multiRangle;chart_title:1 chart;x_title:x;x_min:-6;x_max:6;x_labels:6;y_title:y;y_min:-4;y_max:4;y_labels:5;background_color:black;show_grid:true", "curve_label:cv2;point_color:blue;point_style:circle;point_size:3;line_color:blue;line_style:solid;line_size:1", [-5, -3, -1, 0, 1, 2, 3, 4, 5], [-3.778, -2.9793, -2.0323, -1.1132, 0.2323, 1.2348, 3.9865, 2.3450, 0.4356])
```

, and get the following graph:

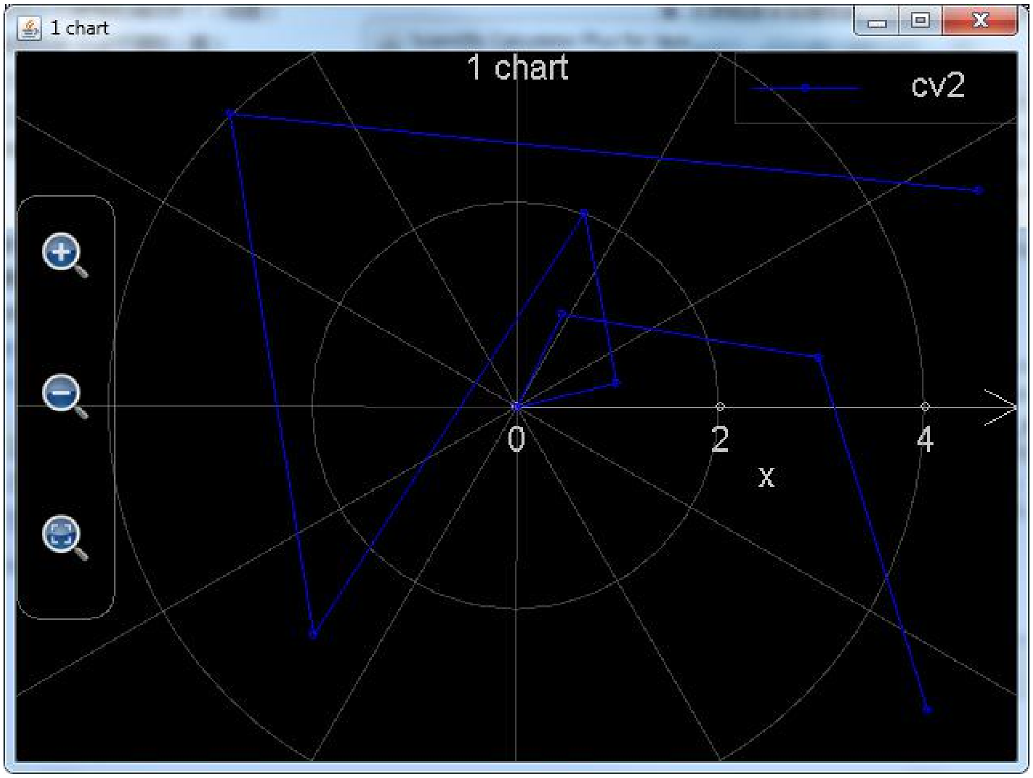


Figure 5.14: Using function plot_multi_xy to draw data chart in the system of polar coordinates.

. The above curve seems irregular compared to Figure 5.11: This is because normal x-y coordinate system is very different from polar coordinate system. If user cannot find any pattern from data in one system, trying another coordinate system may show very clear clues.

Section 4 Plotting 3D Graphs

MFP provides the following 3D graphing functions:

Function Name	Function Info
plot3d	plot3d(5...) : Function plot3D calls plot_multi_xyz function to plot at most eight 3D-surfaces in one chart. It has the following parameters: 1. chart name (i.e. chart file name); 2. chart title; 3. X axis title; 4. Y axis title; 5. Z axis title; 6. curve title; 7. grid or not (if

	false, a filled surface will be drawn); 8. colour at minimum z value; 9. minimum z value (null means automatically determined by software); 10. colour at maximum z value; 11. maximum z value (null means automatically determined by software); 12. u values start from; 13. u values end at; 14. u values’ interval; 15. v values start from; 16. v values end at; 17. v values’ interval; 18. X’s expression (with respect to variables u and v); 19. Y’s expression (with respect to variables u and v); 20. Z’s expression (with respect to variables u and v); ... Note that every new curve needs additional 15 parameters (i.e. parameters 6 to 20). At most 8 curves can be included. An example of this function is plot3D(″chartI″, ″first chart″, ″x″, ″y″, ″z″, ″Curve1″, true, ″red″, -0.5, ″green″, null, 0, pi, pi/8, -pi/2, pi/2, 0, ″sin(u)*cos(v)″, ″sin(u)*sin(v)″, ″cos(u)″) .
plot_3d_surfaces	plot_3d_surfaces(5...) : Function plot_3d_surfaces plots at most 1024 3D-surfaces in one chart. It has the following parameters: 1. chart name (i.e. chart file name); 2. chart title; 3. X axis title; 4. Y axis title; 5. Z axis title; 6. curve title; 7. grid or not (a boolean type. If false, a filled surface will be drawn); 8. front face colour at minimum z value; 9. back face colour at minimum z value; 10. minimum z value (null means automatically determined by software); 11. front face colour at maximum z value; 12. back face colour at maximum z value; 13. maximum z value (null means automatically determined by software); 14. first internal variable name (generally it is ″u″); 15. first internal variable's value starts from; 16. first internal variable's value ends at; 17.

	first internal variable's value changing interval; 18. second internal variable name (generally it is ″v″); 19. second internal variable's value starts from; 20. second internal variable's value ends at; 21. second internal variable's value changing interval; 22. X’s expression (with respect to the two internal variables); 23. Y’s expression (with respect to the two internal variables); 24. Z’s expression (with respect to the two internal variables); ... Note that every new curve needs additional 19 parameters (i.e. parameters 6 to 24). At most 1024 curves can be included. An example of this function is plot_3D_surfaces(″chartI″, ″first chart″, ″x″, ″y″, ″z″, ″Curve1″, true, ″red″, ″cyan″, -0.5, ″green″, ″yellow″, null, ″u″, 0, pi, pi/8, ″v″, -pi/2, pi/2, 0, ″sin(u)*cos(v)″, ″sin(u)*sin(v)″, ″cos(u)″) .
plot_3d_data	plot_3d_data(24) : Function plot_3d_data analyses at least one at most eight groups of data lists and each data group will be plotted as one surface in 3D chart. The number of parameters in these function can be 1 (one curve), 3 (one curve), 6 (two curves), 9 (three curves), 12 (four curves), 15 (five curves), 18 (six curves), 21 (seven curves) and 24 (eight curves). If only one parameter, the parameter must be a 2-D array each element’ s value in the parameter will be a point's z value in the surface, otherwise, every 3 parameters construct a group. In the group, the first parameter is a 1-D array whose elements are the x values of the points, the second parameter is a 1-D array whose elements are the y values, the third parameter is a 2-D array whose elements are the z values of the points in the surface.

	Note that the size of x value parameter and the size of y value parameter must match the size of z value parameter. Examples of this function are plot_3d_data([[2.47, 3.53, 4.88, 9.42], [8.49, 6.76, 5.31, 0.88], [-9, -7, -5, -3, -1]]) and plot_3d_data([1,2,3],[4,5,6,8],[[3,7,2],[5,8,9],[2,6,3],[7,4,4]],[8,7,4,8],[2,1],[[9,3,2,6],[4,5,3,7]]) .
plot_multi_xyz	plot_multi_xyz(2...) : plot_multi_xyz (at least 2 parameters) plots a 3-dim chart which includes at most 1024 surface curves. Parameters 1 and 2 are chart name and settings respectively. The chart settings parameter is a string like ″chart_type:multiXYZ;chart_title:This is a graph;x_title:x axis;x_min:-24.43739154366772;x_max:24.712391543667717;x_labels:10;y_title:Y axis;y_min:-251.3514430737091;y_max:268.95144307370913;y_labels:10;z_title:Z axis;z_min:-1.6873277335234405;z_max:1.7896774628184482;z_labels:10″. Note that chart_type session should always be multiXYZ, and x_labels, y_labels and z_labels means how many scale marks are in the x, y and z axes respectively. From parameter 3, every four parameters define a curve. Among the four parameters, the first describes curve settings, the second is an array of x values, the third is an array of y values and the fourth is an array of z values. An example of curve settings parameter is ″curve_label:cv2;is_grid:true;min_color:blue;min_color_1:cyan;min_color_value:-2.0;max_color:white;max_color_1:yellow;max_color_value:2.0″. Note that the dimension of x, y and z arrays should equal and they should only include real value elements. This function

	returns nothing. An example of this function is plot_multi_xyz(″chartII″, ″chart_type:multiXYZ;chart_title:This is a graph;x_title:x;x_min:-5;x_max:5;x_labels:6;y_title:Y;y_min:-6;y_max:6;y_labels:3;z_title:Z;z_min:-3;z_max:1;z_labels:4″, ″curve_label:cv1;min_color:blue;min_color_1:green;max_color:yellow;max_color_1:red″, [[-4, -2, 0, 2, 4],[-4, -2, 0, 2, 4],[-4, -2, 0, 2, 4]], [[-5, -5, -5, -5, -5], [0, 0, 0, 0, 0], [-5, -5, -5, -5, -5]], [[-2.71, -2.65, -2.08, -1.82, -1.77], [-2.29, -2.36, -1.88, -1.45, -1.01], [-1.74, -1.49, -0.83, -0.17, 0.44]]) .

. Functions plot3d and plot_3d_surfaces draw 3D curves or surfaces in the selected plotting range. Function plot_3d_data plots 3D data chart. Function plot_multi_xyz is very low-level and is called by plot3d and plot_3d_surfaces. Similar to plot2dEx and plot_polar, function plot3d is realized by MFP script and can only draw at most 8 curves (surfaces) in one graph. Its JAVA implemented counterparty, function plot_3d_surfaces, is much faster and can draw as many as 1024 curves (since revision 1.6.7, before revision 1.6.7 it was not exposed to user and can draw at most 8 curves). As such plot_3d_surfaces is recommended to use.

Plot_3d_surfaces function is actually called by the independent 3D Chart Plotter in Scientific Calculator Plus for Android. Its first five parameters respectively set the chart file name (.mfpc extension will be appended automatically), chart title, x-axis name, y-axis name and z-axis name. All these parameters are strings. The background colour is always black and not configurable.

Form the 6th parameter, every 19 parameters set a surface (or curve), they are:

1. Curve or surface name (string based value);

2. Grid or filled surface if it is a surface. Note that this parameter is a Boolean instead of a string. True means the surface is plotted as a grid. False means the surface is filled by selected colour(s). If a curve is plotted, this parameter is strongly recommended to set to true. Otherwise, the colour of the curve will be gray same as the colour of the axes so that it will be very unclear;

3. Colour corresponding to the minimum z value on the front side of the surface. This is a string based value with the following choices: "white ", "black", "red", "green", "blue", "yellow", "cyan", "magenta", "dkgray"

(dark gray) and "ltgray" (light gray). If it is not any of the colours, white colour will be selected by default;

4. Colour corresponding to the minimum z value on the back side of the surface. This is a string based value with the following choices: "white ", "black", "red", "green", "blue", "yellow", "cyan", "magenta", "dkgray" (dark gray) and "ltgray" (light gray). If it is not any of the colours, white colour will be selected by default;

5. Minimum z value. Note that here minimum z value is not necessarily the smallest z value of the plotted curve (or surface). This value is only defined for colour selection. The part of the surface (or curve) whose z value is smaller than or equal to minimum z value will be painted the colour corresponding to minimum z value. The part of the surface (or curve) whose z value is larger than minimum z value will have a transient colour between minimum z colour and maximum z colour. If this value is set to null, software will look for the minimum z value for user;

6. Colour corresponding to the maximum z value on the front side of the surface. This is a string based value with the following choices: "white ", "black", "red", "green", "blue", "yellow", "cyan", "magenta", "dkgray" (dark gray) and "ltgray" (light gray). If it is not any of the colours, white colour will be selected by default;

7. Colour corresponding to the maximum z value on the back side of the surface. This is a string based value with the following choices: "white ", "black", "red", "green", "blue", "yellow", "cyan", "magenta", "dkgray" (dark gray) and "ltgray" (light gray). If it is not any of the colours, white colour will be selected by default;

8. Maximum z value. Note that here maximum z value is not necessarily the biggest z value of the plotted curve (or surface). This value is only defined for colour selection. The part of the surface (or curve) whose z value is larger than or equal to maximum z value will be painted the colour corresponding to maximum z value. The part of the surface (or curve) whose z value is smaller than maximum z value will have a transient colour between maximum z colour and minimum z colour. If this value is set to null, software will look for the maximum z value for user;

9. First internal variable name. Generally it is "u";

10. Starting point of varying range of variable u (if internal variable name is "u", otherwise it is a different name). Here u is the same u used by the independent Chart Plotter in Scientific Calculator Plus for Android. User may refer to Section 4 of Chapter 1 for usage details. Also note that starting point of varying range must be a real value;

11. Ending point of varying range of variable u. It also must be a real value;

12. Varying step length of u. It must be a real value and it equals varying range / number of steps. User may set it zero which means the step length is automatically determined by software;

13. Second internal variable name. Generally it is "v";

14. Starting point of varying range of variable v (if internal variable name is "v", otherwise it is a different name). Here v is the same v used by the independent Chart Plotter in Scientific Calculator Plus for Android. User may refer to Section 4 of Chapter 1 for usage details. Also note that starting point of varying range must be a real value;

15. Ending point of varying range of variable v. It also must be a real value;

16. Varying step length of v. It must be a real value and it equals varying range / number of steps. User may set it zero which means the step length is automatically determined by software;

17. Function of X value with respect to u and v (string based expression. User may refer to Section 4 of Chapter 1 for details of X(u, v));

18. Function of Y value with respect to u and v (string based expression. User may refer to Section 4 of Chapter 1 for details of Y(u, v));

19. Function of Z value with respect to u and v (string based expression. User may refer to Section 4 of Chapter 1 for details of Z(u, v));

Because function plot_3d_surfaces can draw as many as 1024 curves in one chart, the number of its parameters can be at most 5+19*1024=19461.

The following example shows user how to draw a colorful cube. A cube includes 6 surfaces, which means function plot_3d_surfaces needs to draw six curves (surfaces) in one chart. Assume the height of the cube is 2, u is from -1 to 1 and step length is 2, and v is from -1 to 1 and its step length is also 2. Because both u and v’s step lengths equal their variation range, no grid will be drawn on the surfaces.

When drawing the surfaces, notice that each of the surfaces must be perpendicular to one of the x, y and z axes and in parallel with the other two. As such one of the x, y and z variables must be either -1 or 1, and the other two equal to u and v respectively. The whole statement is:

```
Plot_3d_surfaces("3dBox", "3D Box", "x", "y", "z", _

"",false,"red","red",null,"red","red",null,"u",-1,1,2,"v",-1,1,2,"u","v","1", _
"",false,"green","green",null,"green","green",null,"u",-1,1,2,"v",-1,1,2,"u","1","v", _
"",false,"blue","blue",null,"blue","blue",null,"u",-1,1,2,"v",-1,1,2,"1","u","v", _
"",false,"yellow","yellow",null,"yellow","yellow",null,"u",-1,1,2,"v",-1,1,2,"u","v","-1", _
"",false,"cyan","cyan",null,"cyan","cyan",null,"u",-1,1,2,"v",-1,1,2,"u","-1","v", _
"",false,"magenta","magenta",null,"magenta","magenta",null,"u",-1,1,2,"v",-1,1,2,"-1","u","v")
```

, and the plotted graph is:

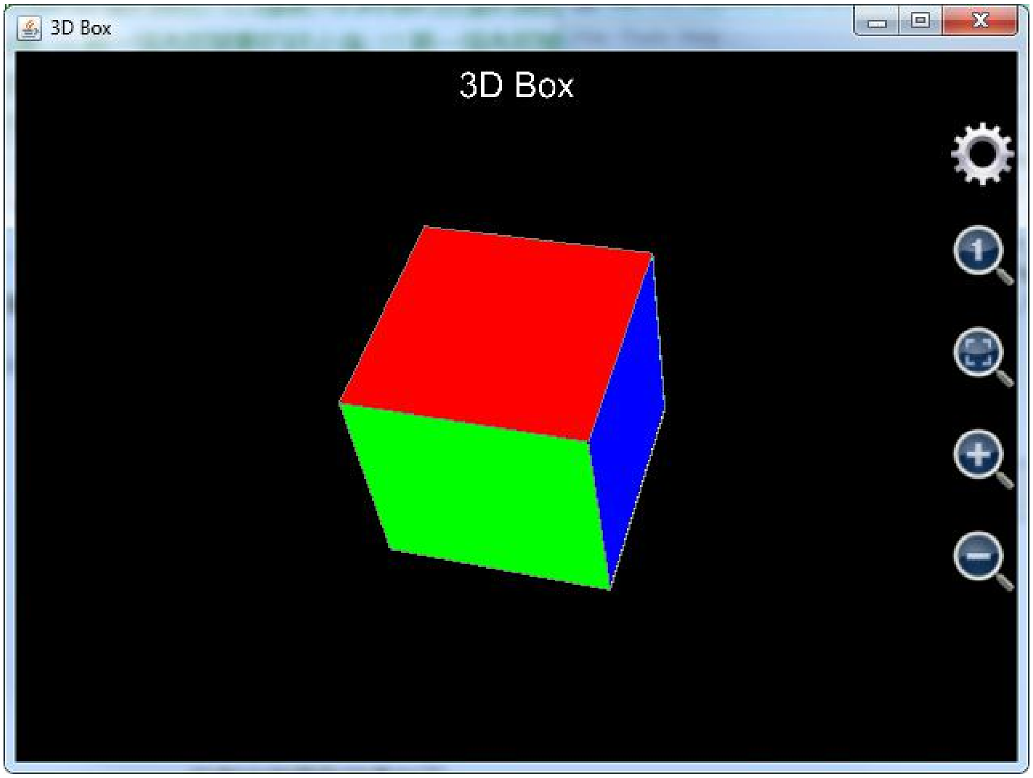
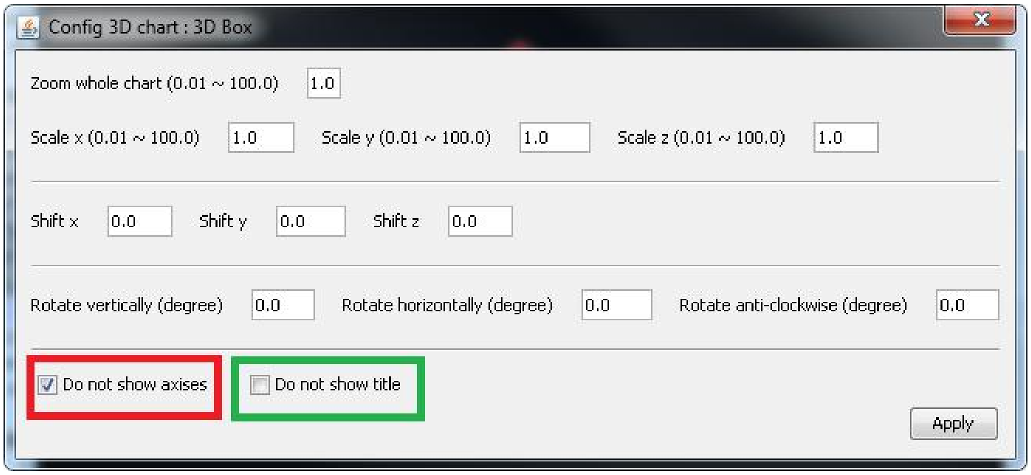


Figure 5.15: Using plot_3d_surfaces function to draw a cube.

Note that in the above example the title of each surface is an empty string so that they will not be shown in the chart.



Zoom whole chart (0.01 ~ 100.0) 1.0

Scale x (0.01 ~ 100.0) 1.0 Scale y (0.01 ~ 100.0) 1.0 Scale z (0.01 ~ 100.0) 1.0

Shift x 0.0 Shift y 0.0 Shift z 0.0

Rotate vertically (degree) 0.0 Rotate horizontally (degree) 0.0 Rotate anti-clockwise (degree) 0.0

☑ Do not show axises □ Do not show title

Figure 5.16: Settings of 3D graph, showing/hiding axes and title.

Like 3D charts created by function plot_exprs, user can drag the chart to rotate and zoom in and out by clicking corresponding buttons or pinching in Anroid. User can also show/hide axes and title by clicking the gear button and checking/unchecking corresponding items, as shown in the above chart.

If user is using Scientific Calculator Plus revision 1.6.6, axes and title can only be shown or hidden together, i.e. no way to show axes but hide title or hide axes but show title. Axes and title cannot be hidden in revisions older than 1.6.6.

If user shows axes but hide title, the plotted graph is shown below. Clearly, it doesn't look as nice as Figure 5.16:

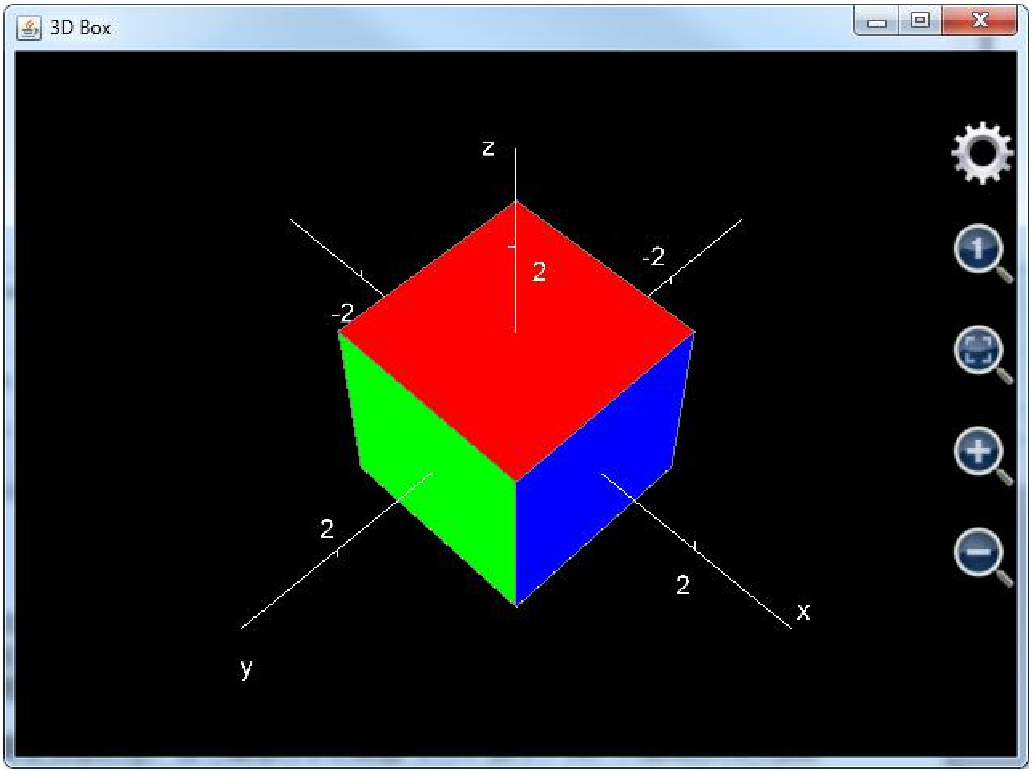


Figure 5.17: Cube chart with axes shown but title hidden.

Similar to function plot_2d_curves, charts created by plot_3d_surfaces will be saved in AnMath/charts folder and can be reopened by Scientific Calculator Plus.

Cube is the simplest 3D shape. Plot_3d_surfaces function is able to draw much more interesting and sophisticated graphs, like Egyptian pyramid.

Pyramid includes five surfaces. The bottom one is a square and the other four are tilted triangles. Assume the width of the square is 2 and it is placed on the x-y surface. Let u change from -1 to 1 with a step length equal to 2, and v change from -1 to 1 with a step length also equal to 2. Then x is u, y is v and z is 0 because the altitude of the bottom surface is zero.

The east side of pyramid is a tilted triangle. Assume its height is 1.67. Let u change from 0 to 1 with step length being 1. Let v change from -1 to 1 with step length equal to 2. Then x is 1-u, y is v*(1-u) and z is 1.67*u.

The west side of pyramid is also a leaning triangle. Let u change from 0 to 1 with step length being 1. Let v change from -1 to 1 with step length equal to 2. Then x is u-1, y is v*(1-u) and z is 1.67*u.

To draw north side of pyramid, let u change from 0 to 1 with step length equal to 1 and v change from -1 to 1 with step length equal to 2. Then x is v*(1-u), y is 1-u and z is 1.67*u.

To draw south side of pyramid, let u change from 0 to 1 with step length equal to 1 and v change from -1 to 1 with step length equal to 2. Then x is v*(1-u), y is u-1 and z is 1.67*u.

The statement to draw the pyramid is shown below.

```
Plot_3d_surfaces("pyramid", "Pyramid", "x", "y", "z", _

"",false,"red","red",null,"red","red",null,"u",0,1,1,"v",-1,1,2,"1-u","v*(1-u)","1.67*u", _          // east

"",false,"magenta","magenta",null,"magenta","magenta",null,"u",0,1,1,"v",-1,1,2,"u-1","v*(1-u)","1.67*u", _      // west

"",false,"blue","blue",null,"blue","blue",null,"u",0,1,1,"v",-1,1,2,"v*(1-u)","1-u","1.67*u", _    // north

"",false,"yellow","yellow",null,"yellow","yellow",null,"u",0,1,1,"v",-1,1,2,"v*(1-u)","u-1","1.67*u", _
            // south

"",false,"green","green",null,"green","green",null,"u",-1,1,2,"v",-1,1,2,"u","v","0") // bottom
```

The plotted graph looks like:

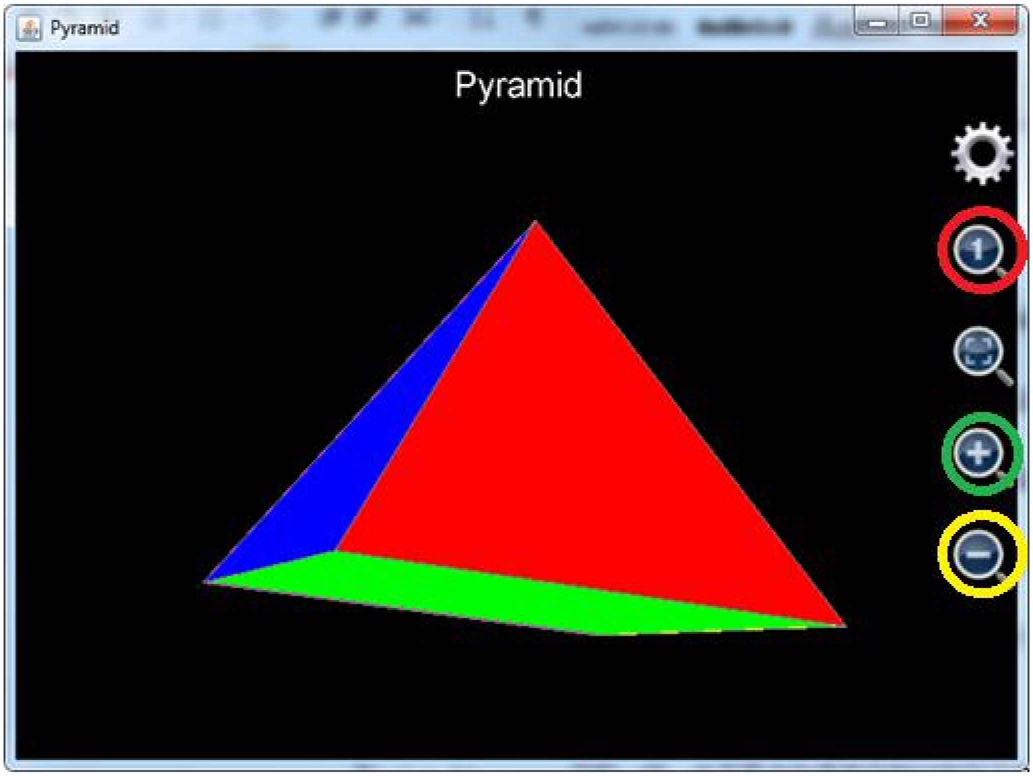


Figure 5.18: Using plot_3d_surfaces function to draw pyramid.

Note that when the graph is just plotted, the size of pyramid is a bit small compared to the graph size. User can click the zoom-in button in the green circle to enlarge the pyramid, then drag it to rotate to find the best viewing angle.

Also note that, revision 1.6.6 or earlier does not automatically set unit length ratio of x, y and z axes to be 1:1:1. Therefore, at the first glance the plotted shape may not look like a pyramid at all if user is using an old version Scientific Calculator Plus. In this case user can click the button in the red circle to adjust length ratio of x, y and z axes, and then click the zoom-out button in the yellow circle to see a true pyramid.

Although function plot_3d_surfaces exists in older Scientific Calculator Plus, this API hasn't been exposed to user until revision 1.6.7. As such, in most of the cases users of Scientific Calculator Plus 1.6.6 or earlier call function plot3d to draw 3D charts. Limit is plot3d only supports 8 groups of x, y and z expressions. In order to draw a sophisticated graph including more than 8 surfaces, user may use function iff to draw more than one surfaces when u (or v) is in different parts of its plotting range. To avoid connection between two surfaces, x or y or z should be set to Nan between the plotting range parts. Clearly, the same approach can be applied to functions plot_3d_surfaces, plot2dEx, plot_2d_curves, plot_polar and plot_polar_curves. Nevertheless, from revision 1.6.7, functions plot_3d_surfaces, plot_2d_curves and plot_polar_curves can draw at most 1024 curves (surfaces) in one graph so that this trick is meaningless to them.

In Section 4 of Chapter 1, user has been demonstrated how to draw Shanghai Oriental Pearl Radio & TV Tower using the independent Chart Plotter tool. Here the MFP code which does the same thing is provided. Function plot3d is used instead of plot_3d_surfaces so that old Scientific Calculator Plus users can also run it.

As mentioned above, plot3d uses one group of x, y and z expressions to draw multiple surfaces. For example, when drawing the three connecting columns between the big sphere and the small sphere, plot3d sets u changing from 0 to 8 which means angle changes from 0 to 8*pi, v changing from 20 to 70 which means points on the columns have an altitude from 20 to 70. Expression of x is iff(u<=2,1.5*cos(u*pi)-2,and(u>=3,u<=5), 1.5*cos(u*pi), u>=6, 1.5*cos(u*pi)+2, Nan). This means when u is between 0 and 2 (i.e. angle is between 0 to 2*pi), plot3d draws the first column; when u is between 3 and 5 (i.e. angle is between 3*pi to 5*pi); plot3d draws the second column, when u is between 6 and 8 (i.e. angle is between 6*pi to 8*pi), plot3d draws the last column; otherwise, x is Nan so that the three columns are not connected.

Similarly, expression of y is iff(u<=2,1.5*sin(u*pi)+2/sqrt(3),and(u>=3,u<=5), 1.5*sin(u*pi)- 4/sqrt(3), u>=6, 1.5*sin(u*pi)+2/sqrt(3), Nan). Expression of z, compared to x and y, is relatively simple because z is the height. So z should be v.

The whole statement is shown below:

```
plot3d("Oriental_Pearl_Tower","Oriental Pearl Tower","x","y","z", _

"",false,"red",null,"yellow",null,0,8,0.25,0,20,20,"iff(u<=2,3*cos(u*pi)-(20-
v)*sqrt(3)/2,and(u>=3,u<=5), 3*cos(u*pi), u>=6, 3*cos(u*pi)+(20-v)*sqrt(3)/2,
Nan)","iff(u<=2,3*sin(u*pi)+(20-v)/2,and(u>=3,u<=5), 3*sin(u*pi)-(20-v)*sqrt(3)/2, u>=6,
3*sin(u*pi)+(20-v)/2, Nan)","v", _ //Plot supporting leaning columns

"",false,"green",null,"yellow",null,-1,1,0.25,0,20,20,"cos(u*pi)*2","sin(u*pi)*2","v", _ //plot
connection column

"",false,"red",null,"cyan",null,-pi,pi,pi/10,-
pi/2,pi/2,pi/10,"10*cos(u)*cos(v)","10*sin(u)*cos(v)","10*sin(v)+20", _   //plot the big ball

"",false,"green",null,"blue",null,0,8,0.25,20,70,50,"iff(u<=2,1.5*cos(u*pi)-2,and(u>=3,u<=5),
1.5*cos(u*pi), u>=6, 1.5*cos(u*pi)+2, Nan)","iff(u<=2,1.5*sin(u*pi)+2/sqrt(3),and(u>=3,u<=5),
1.5*sin(u*pi)- 4/sqrt(3), u>=6, 1.5*sin(u*pi)+2/sqrt(3), Nan)","v", _ //plot the connection
columns between the big ball and the small ball

"",false,"magenta",null,"white",null,-pi,pi,pi/10,-
pi/2,pi/2,pi/10,"6*cos(u)*cos(v)","6*sin(u)*cos(v)","6*sin(v)+70", _ //plot the small

"",false,"yellow",null,"green",null,0,2,0.25,70,85,15,"cos(u*pi)*1.5","sin(u*pi)*1.5","v", _ //plot
another column above the small ball

"",false,"red",null,"cyan",null,-pi,pi,pi/10,-
pi/2,pi/2,pi/10,"2*cos(u)*cos(v)","2*sin(u)*cos(v)","2*sin(v)+85", _ //Plot the smaller ball

"",false,"red",null,"ltgray",null,-1,1,0.2,85,115,10,"0.5*max(0.2,(115-
v)/30)*cos(u*pi)","0.5*max(0.2,(115-v)/30)*sin(u*pi)","v") //Plot the antenna on top
```

User can write the above statement in an MFP function and save it in a script file, then simply type the function name and press ENTER key to draw the TV tower in Command Line. This is much easier than typing the eight groups of the parameters in Chart Plotter.

The plotted graph is shown below. Note that if using revision 1.6.6 or earlier, user needs to click the ratio adjusting button (the icon is a small 1 in a magnifying glass) to ensure that x:y:z is 1:1:1, also needs to click the gear button to hide axes and title if using revision 1.6.6 (in earlier revision axes and title cannot be hidden).

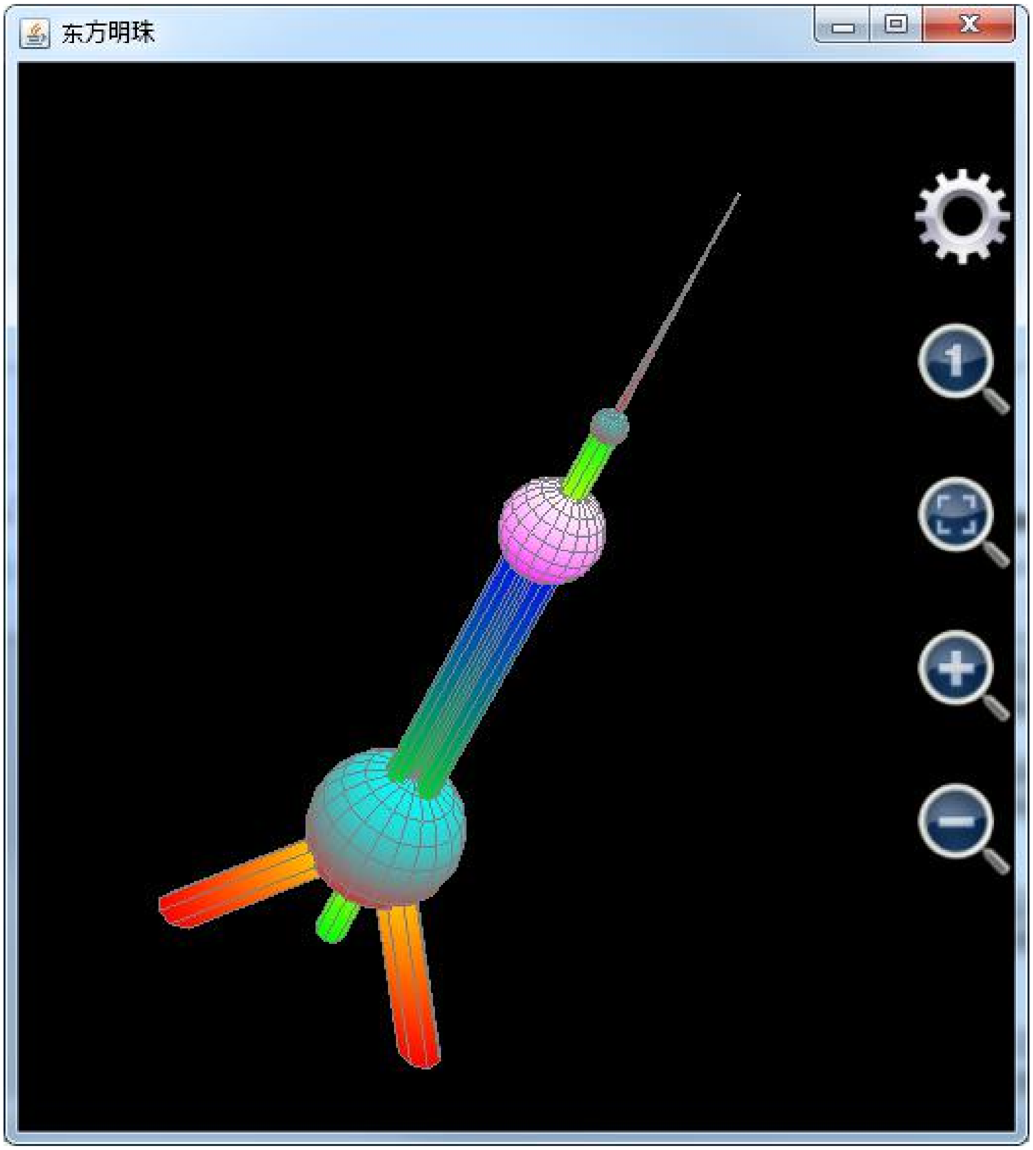


Figure 5.19: Using plot3d function to draw Shanghai Oriental Pearl Radio & TV Tower.

User may want to draw 3D curves instead of surfaces. Function plot_3d_surfaces can also do this job. When plotting 3D curves, only one of u and v can be included in the x, y and z's expressions. Otherwise, surface instead of curve is plotted. For example, to draw a spiral line with a radius of 5, user may look on u as the angle and set it to change from -2*pi to 2*pi. As such x is 5*cos(u), y is 5*sin(u) and z is u. The statement is:

```
Plot_3d_surfaces("spiralline", "Spiral Line", "x", "y", "z", "", true, "cyan", "cyan", null, "red", "red", null, "u", -2*pi, 2*pi, pi/50, "v", 0, 1, 1, "5 * cos(u)", "5 * sin(u)", "u")
```

The plotted spiral line is:

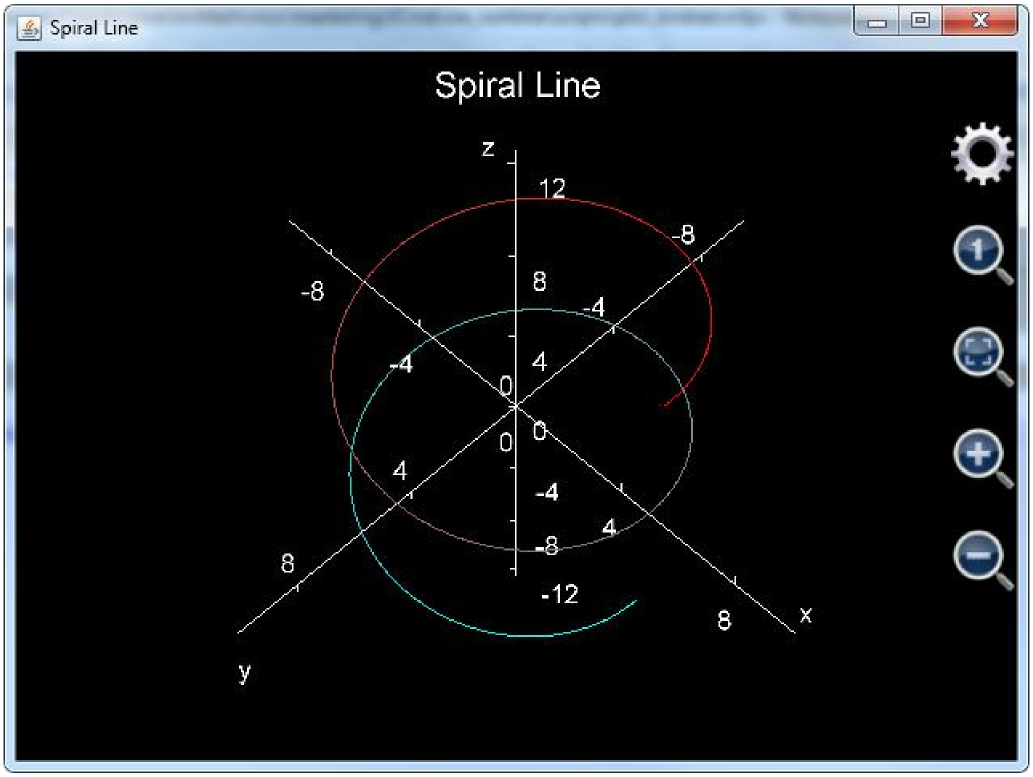


Figure 5.20: Using function plot_3d_surfaces to draw a spiral line.

User may keep in mind that:

1. Although v is not used in x, y and z's expressions, it's varying range and step length should be carefully selected because the number of points to draw equals u's number of steps times v's number of steps. To save calculation time, v can be set from 0 to 1 and its step length is 1;

2. The is_grid flag should be true to draw a 3D curve. Otherwise, the colour of the curve will be gray so that very difficult to observe;

3. The number of steps of u cannot be too small. Otherwise, the plotted curve will not be smooth. In the above example, u's step length is pi/50 and u has 200 steps.

Function plot_3d_data generates 3D data chart. This function is called in two ways. It can accept only one 2D array parameter to draw a surface. In this parameter, every element must be a real value, which is the z value of a point on the surface. The element's indices are x and y values of the point. For example, running the following statement

```
Plot_3d_data([[7,5,3,6,10,14],[4,7,2,8,9,14],[4,3,9,2,9,15],[2,8,NaN,5,8,16],[-1,9,11,6,8,17],[-
4,7,12,5,9,20]])
```

user will get (note that the graph has been rotated and enlarged):

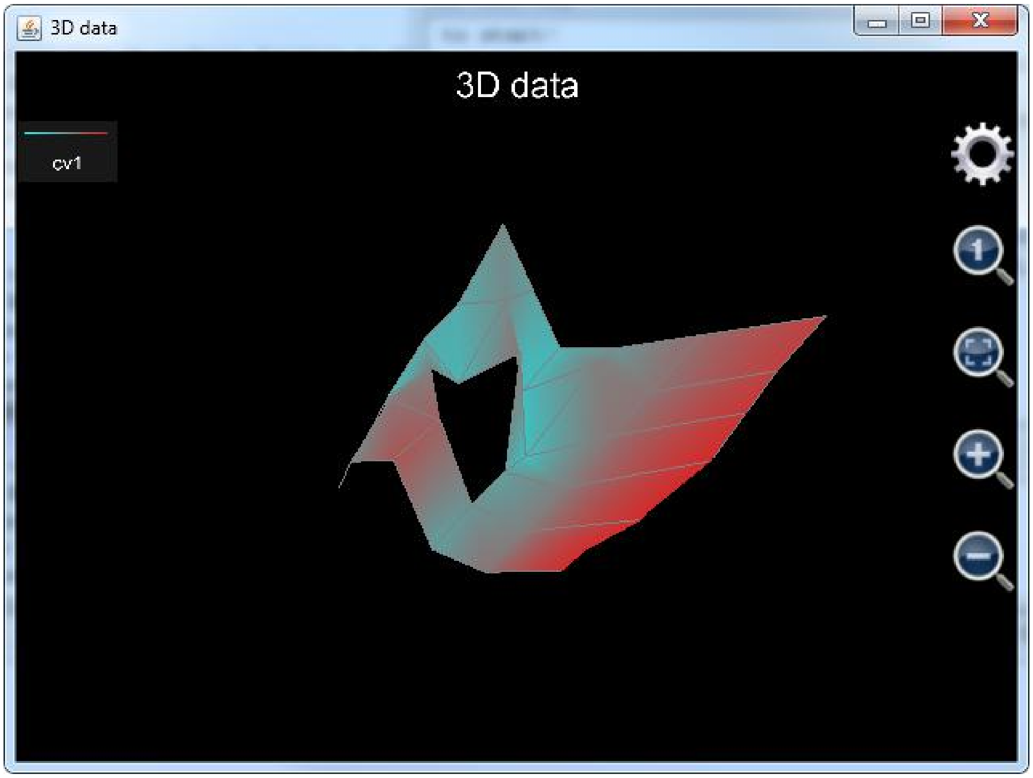


Figure 5.21: Using function plot_3d_data to draw a single surface.

Notice that there is a hole on the surface. This is because in the data array parameter value of an element is Nan. Nan cannot be plotted so that the surface includes the hole.

Plot_3d_data can also plot multiple surfaces. In this case the number of the parameters must be multiplication of 3. Every three parameters comprise a parameter group. In the parameter group, the first parameter must be a 1D array whose elements are x values of the points; the second parameter is also a 1D array whose elements are y values of the points; the third parameter should be a 2D array whose elements are z values of the points on the surface. Note that the length of the first parameter should match the size of the first dimension of the third parameter, and the length of the second parameter should match the size of the second dimension of the third parameter.

Function plot_3d_data can also be applied in engineering where analyzing 3D data is important. For instance, in financial engineering, risk analysts need to analysis the volatility surface of a group of options, and use the volatility surface to price a particular option contract. Plot_3d_data, therefore, can be called to visualize the surface, as shown in the following example:

```
plot_3d_data([48.000000, 50.000000, 52.000000, 54.000000, 56.000000, 58.000000, 60.000000,
62.000000, 64.000000, 66.000000, 68.000000, 70.000000, 72.000000, 74.000000, 76.000000,
78.000000, 80.000000, 82.000000, 84.000000, 86.000000, 88.000000, 90.000000, 92.000000,
94.000000, 96.000000, 98.000000, 100.000000, 105.000000], [30, 58, 121, 212, 576, 940], _ //x and
y of surface 1

[[NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, 0.49508067351396218000,
0.45756582984888738000, 0.41913711426069727000, 0.37990131595995524000,
0.34996178524456606000, 0.30776619051400522000, 0.28462015821129766000,
0.27075500739772851000, 0.26301012549556918000, 0.24950232072545608000,
0.24019484695203744000, 0.23175291515931623000, 0.21112501922301888000,
0.20651763047720664000, 0.21070439806536975000, 0.22206990800626822000,
0.23691523835915387000, 0.26035129175640970000, NAN, 0.35693427858118065000], _

[NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, 0.36859505107478957000,
0.34682167993243251000, 0.33284975263494410000, 0.32119915959842893000,
0.31050760053766019000, 0.29974406021726552000, 0.29453798692550298000,
0.28157889138027392000, 0.27318479365993703000, 0.26342709777494933000,
0.25752572211832075000, 0.24780946658943892000, 0.24166776632146400000,
0.23722978504246392000, 0.23195815505284242000, 0.22898424812758009000,
0.22833835748043799000, 0.23681894432023268000, 0.26478408970435013000], _

[NAN, NAN, NAN, NAN, NAN, NAN, NAN, 0.36447017097320361000,
0.35449192506546090000, 0.34516619206807542000, 0.34261461130215798000,
0.32635501530861477000, 0.32107173363018415000, 0.31233375990009160000,
0.30479530303155050000, 0.29817914152719677000, 0.29058822590764583000,
0.28282080501333134000, 0.27496574457106382000, 0.26851242637016437000,
0.26141077894592291000, 0.25587622110424685000, 0.25097496943207720000,
0.24646926304153294000, 0.24360994236677280000, 0.24074283746453087000,
0.23796452973380869000, 0.23534059389240872000], _

[0.42886625487784302000, 0.42275377605823883000, 0.41329219686969904000,
0.40460391970410370000, 0.39481551194770520000, 0.38291712255814248000,
0.37662551028641211000, 0.36478616087804611000, 0.36022367426251140000,
0.35255567514870667000, 0.34632136686091713000, 0.33619033083866695000,
0.32940848011458052000, 0.32550914476490195000, 0.31762251703077932000,
0.31380139485946612000, 0.30905226419037485000, 0.30338087644402684000,
0.29873679230470152000, 0.28685190784393211000, 0.28138845244953115000,
0.27662410367186036000, 0.27058634105931750000, 0.26931959970842401000,
0.26493899498451701000, 0.26164809336719214000, 0.25887643135300442000,
0.25318504482282400000], _

[NAN, NAN, NAN, NAN, NAN, NAN, NAN, 0.36501592858551429000,
0.36002318219714213000, 0.35559613466145090000, 0.34848867397787564000,
0.34653605316601327000, 0.34331817675589471000, 0.33758506685395551000,
0.33494376931090725000, 0.33249369924862260000, 0.32894957372789690000,
0.32563131380755028000, 0.32252394427107839000, 0.31590635444985415000,
0.31230809418058103000, 0.30891316532484459000, 0.30690810447495731000, NAN, NAN,
NAN, NAN, NAN], _

[NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN,
0.35115605953314510000, 0.34821397817102240000, 0.34569662266907020000,
0.34358159686638989000, 0.34085234801142689000, 0.34839263577550034000,
0.33660760870094886000, 0.33959719768707108000, 0.33713092050360410000,
0.33603184408546544000, NAN, NAN, NAN, NAN, NAN]], _ //z values of surface 1
[50.000000, 52.000000, 54.000000, 56.000000, 58.000000, 60.000000, 62.000000, 64.000000,
66.000000, 68.000000, 70.000000, 72.000000, 74.000000, 76.000000, 78.000000, 80.000000,
82.000000, 82.500000, 83.000000, 84.000000, 86.000000, 88.000000, 90.000000, 92.000000,
94.000000, 96.000000], [24, 52, 143, 233, 506, 877], _ //x and y of surface 2

[[NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN,
0.22964633802072443000, 0.18031707781034231000, 0.13034337245591013000,
0.11131700480412310000, NAN, 0.10619822668851642000, 0.10041352495351766000,
0.10939206628254365000, 0.14908566947743185000, 0.16602982367522820000, NAN, NAN,
NAN], _

[NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN,
0.22540094145088913000, 0.19705350522103846000, 0.16395345741263651000,
0.14144557031336311000, 0.12637305801604665000, NAN, 0.11119409065181833000, NAN,
NAN, NAN, NAN, NAN, NAN, NAN, NAN], _

[NAN, NAN, NAN, NAN, NAN, NAN, 0.25715041824992257000, 0.23621769883269250000,
0.21525558611698195000, 0.20045165924029371000, 0.18541707800373045000,
0.17359058112818165000, 0.16231450779286907000, 0.14916833017145850000,
0.13963488906422594000, 0.13177711734828756000, 0.12730661724897638000, NAN, NAN,
0.10557373569647757000, 0.10119196691910112000, 0.10116033427429517000,
0.10388742349750228000, 0.10620359931911844000, 0.11622872282660483000,
0.12972672550374550000], _

[NAN, NAN, NAN, NAN, NAN, NAN, NAN, 0.20352671403747510000,
0.19298953607665226000, 0.18372653703149414000, 0.17460337106727522000,
0.16855579275820740000, 0.16219673193089182000, 0.15533583755104832000,
0.15143304483201725000, 0.14966908624163464000, 0.14551826337243573000, NAN, NAN,
0.13133944009346873000, 0.12356296864493185000, 0.11899060584716444000,
0.11960170233648706000, 0.11951725172463327000, 0.11866793224195711000,
0.12167362000206712000], _

[0.24384524786557735000, 0.23533516044988553000, 0.22716883635794988000,
0.21453230778081070000, 0.21474513249393276000, 0.20918925878245592000,
0.20609984918018193000, 0.20191453785290187000, 0.19781979463707985000,
0.19448827786958967000, 0.19106299814050737000, 0.19227308566292306000,
0.18922818688715029000, 0.18886743252564508000, 0.18912087690028995000,
0.18990974472166203000, 0.19268838899006788000, NAN, NAN, 0.13665902914514916000,
0.13309865533237508000, 0.13053916709176369000, 0.12692797194421160000,
0.12528654150114951000, NAN, NAN], _

[NAN, NAN, NAN, NAN, 0.20896766837849659000, 0.20149697646213488000, NAN, NAN,
NAN, NAN, NAN, 0.20549683791479759000, 0.20493999835449925000,
0.20823799582345237000, 0.21129319127054960000, 0.21169404646035919000,
0.19212457911706818000, NAN, NAN, 0.18574033886119370000, 0.17054792142025460000,
NAN, NAN, NAN, NAN, NAN]]) // z value of surface 2
```

In the above statement, x and y values are stored in 1D arrays for surfaces 1 and 2 respectively. X is moneyness, and y is expiry date. Z values are the corresponding volatility value at each moneyness and expiry. Also note that sometimes some data are unreasonable and should be discarded. Plot_3d_data function is able to ignore bad data by setting z value to be Nan. This is the reason that user sees many Nans in the 2D array parameters.

The graph drawn by the statement is shown below. Note that if Scientific Calculator Plus is revision 1.6.7 or later, the plotted graph looks like a belt when generated because the span of y values is significantly larger than the span of x. This prevents user from reading the details of the volatility surface. To stretch the surfaces, user needs to click the button in the red square to fill the chart window with the surfaces. If Scientific Calculator Plus is revision 1.6.6 or earlier, the surfaces are automatically adjusted to fit the chart window so no action is required.

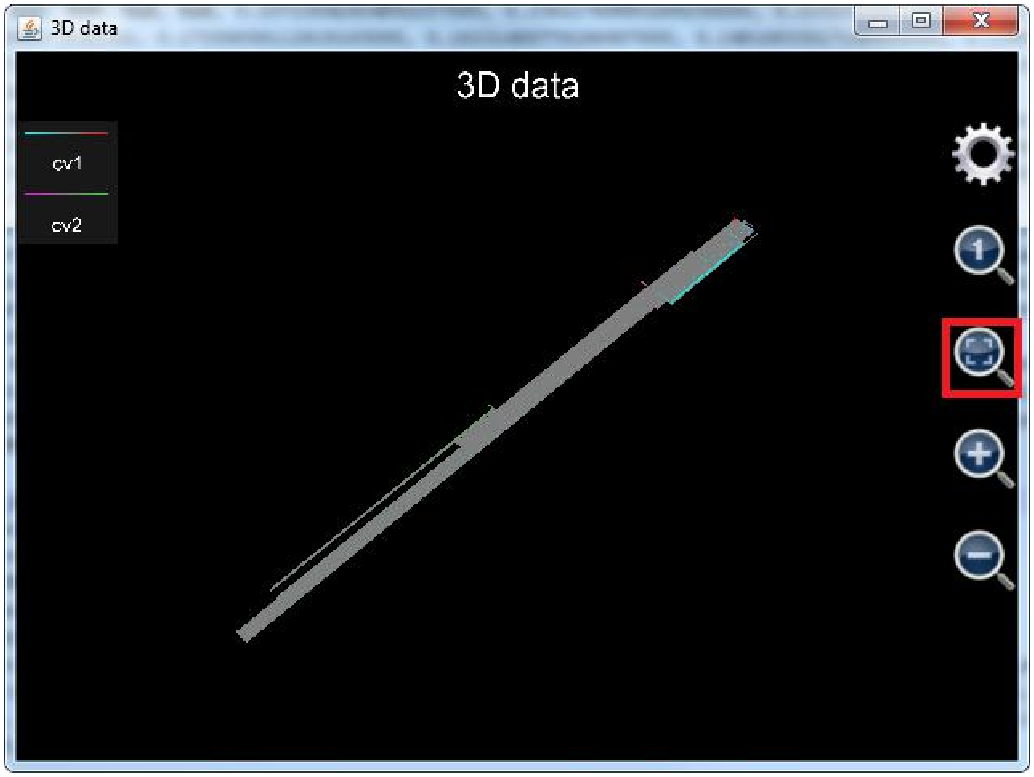


Figure 5.22: Using function plot_3d_data to draw two volatility surfaces. The surfaces haven't been stretched.

After adjustment, the surfaces look like Figure 5.23: .Note that to place the surfaces in the center of the chart, z axis has been shifted by -0.3 and the axes are set to shown so that user can see clearly the ratio of x:y:z.

Like plot_2d_data, function plot_3d_data does not include a parameter to set chart file name so that plotted graph is not saved and cannot be reopened later on.

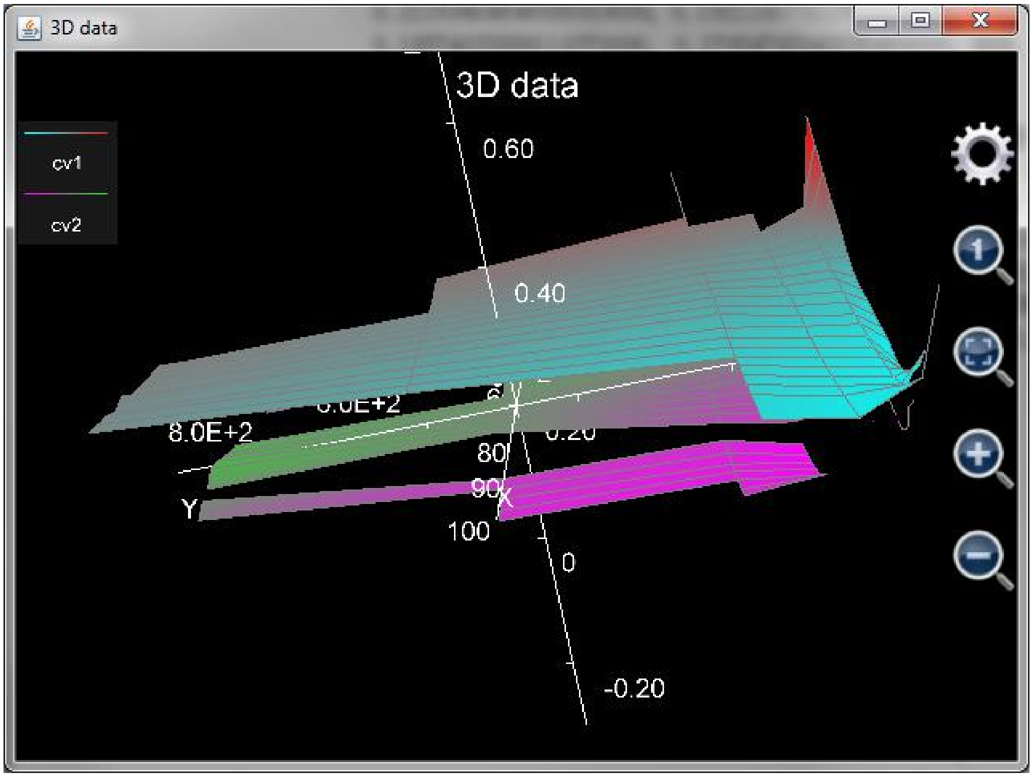


Figure 5.23: Two volatility surfaces after adjustment.

Function plot_multi_xyz is a very low-level API. Similar to plot_3d_data, this function accepts data values instead of expressions as parameter. Function plot_3d_data calls this function directly (although some data conversion work is required) while function plot3d first calculates values of the to-be-plotted expressions at each point and then calls this function.

The first parameter of plot_multi_xyz is string based chart name, i.e. chart file name without .mfpc extension.

The second parameter is chart level settings. All the configurations at chart level are included in the string based parameter. Each configuration follows the format of

Setting's name:setting's value;

. All the configurations are then concatenated into a string, like:

"chart_type:multiXYZ;chart_title:This is a graph;x_title:x axis;x_min:-24.43739154366772;x_max:24.712391543667717;x_labels:10;y_title:Y axis;y_min:-251.3514430737091;y_max:268.95144307370913;y_labels:10;z_title:Z axis;z_min:-1.6873277335234405;z_max:1.7896774628184482;z_labels:10"

, here chart_type is the type of chart, which must be multiXYZ, chart_title is the title of the chart, x_title is the name of x-axis, x_min is the minimum value x-axis shows when the chart is plotted, x_max is the maximum value x-axis shows when the chart is plotted, x_label is the number of scale marks in x-axis, y_title is the name of y-axis, y_min is the minimum value y-axis shows when the chart is plotted, y_max is the maximum value y-axis shows when the chart is plotted, y_label is the number of scale marks in y-axis, z_title is the name of z-axis, z_min is the minimum value z-axis shows when the chart is plotted, z_max is the maximum value z-axis shows when the chart is plotted, z_label is the number of scale marks in z-axis.

The third parameter of plot_multi_xyz is settings for a single curve (or surface). Like the previous parameter, all the configurations are included in a string and each of them follows the format of

Setting's name:setting's value;

. For example:

"curve_label:cv2;is_grid:true;min_color:blue;min_color_1:cyan;min_color_value:-2.0;max_color:white;max_color_1:yellow;max_color_value:2.0"

, where curve_label is the title of the curve, is_grid determines whether drawing grid only or filling the whole surface by the selected colours, min_color means the colour painted on the front side of the surface when z value equals minimum_color_value, min_color_1 means the colour painted on the back side of the surface when z value equals minimum_color_value, minimum_color_value is the z value corresponding to the min_color and min_color_1, max_color means the colour painted on the front side of the surface when z value equals max_color_value, max_color_1 means the colour painted on the back side of the surface when z value equals max_color_value, maximum_color_value is the z value corresponding to the max_color and max_color_1. The front side colour of a point whose z value is between min_color_value and max_color_value is a transient colour between min_color and max_color. The back side colour of a point whose z value is between min_color_value and max_color_value is a transient colour between min_color_1 and max_color_1. If min_color_value or max_color_value is null, software will automatically determine the minimum z value or maximum z value of the surface.

The fourth parameter of plot_multi_xyz is the x values of all the points in this curve. Note that this parameter must be a 2-D array and each element in the array must be a real number. If an element is Nan, like function plot_3d_data, the surface is broken at the Nan point (which means a hole).

The fifth parameter of plot_multi_xyz is the y values of all the points in this curve. Note that this parameter must be a 2-D array with the same size as parameter four, and each element in the array must be a real number. If an element is Nan, like function plot_3d_data, the surface is broken at the Nan point (which means a hole).

The sixth parameter of plot_multi_xyz is the z values of all the points in this curve. Note that this parameter must be a 2-D array with the same size as parameters four and five, and each element in the array must be a real number. If an element is Nan, like function plot_3d_data, the surface is broken at the Nan point (which means a hole).

If user wants to draw more than one curves (surfaces), the additional curves need their own parameter groups and each group includes parameters 3, 4, 5 and 6. User can draw at most 1024 surfaces (or at most 8 surfaces if using revision 1.6.6 or earlier). Therefore the max number of parameters of this function is 2 + 1024*4 = 4098.

An example of this function is:

```
plot_multi_xyz("chartII", "chart_type:multiXYZ;chart_title:This is a graph;x_title:x;x_min:-
5;x_max:5;x_labels:6;y_title:Y;y_min:-6;y_max:6;y_labels:3;z_title:Z;z_min:-3;z_max:1;z_labels:4",
"curve_label:cv1;min_color:blue;min_color_1:green;max_color:yellow;max_color_1:red", [[-4, -2, 0,
2, 4],[-4, -2, 0, 2, 4],[-4, -2, 0, 2, 4]], [[-5, -5, -5, -5, -5], [0, 0, 0, 0, 0], [-5, -5, -5, -5, -5]], [[-2.71, -2.65,
-2.08, -1.82, -1.77], [-2.29, -2.36, -1.88, -1.45, -1.01], [-1.74, -1.49, -0.83, -0.17, 0.44]])
```

. The plotted graph is similar to a piece of paper lying on the x-y plane-surface, and then it is folded roughly along x-axis from positive y to negative y. This example shows that plot_multi_xyz function can draw a distorted surface, even if on the surface some (x, y) pairs correspond to several z values.

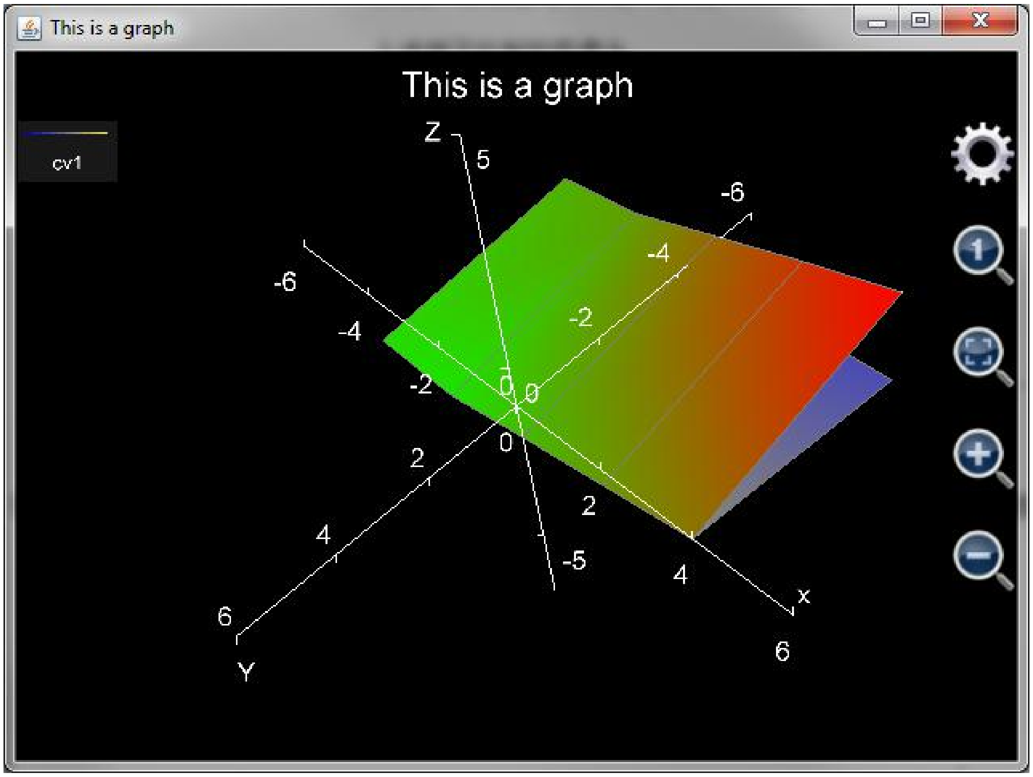


Figure 5.24: Using plot_multi_xyz function to draw a folded surface.

Since function plot_multi_xyz has no particular requirements on the x, y and z values, user can plot any 3D graph using this tool if the coordinates of the points are known.

Summary

This chapter introduces MFP graphing functions. There are two types of graphing functions. One is plotting expressions like plot_exprs. User needs not to input x, y and z's varying range. Only to-be-plotted expressions are required as parameters. After chart is generated, plotting range can be adjusted by user. Plotting expressions is quick and easy. However, each time user adjusts plotting range part or even all of the point values have to be recalculated. Point recalculation is time expensive and may result in significant delay to user's actions.

The second type of graphing functions needs user's input of plotting range for internal variable(s), i.e. t for 2D or polar charts and u and v for 3D charts. Expressions of x, y and z are functions of the internal variables which are also input by user. After graph is plotted, plotting range will not change so that shifting or zooming the graph will be swift and smooth. Another benefit is this type of graphing functions is capable of drawing very complicated graph, like Shanghai Oriental Pearl Radio & TV Tower. Because of their flexibility, these graphing functions are able to draw any graphs for user theoretically.

Chapter 6 Input/Output and File Operations

Input/Output is the communication between user and MFP program. MFP programming language provides user a set of C-like input/output functions. Using these functions, user can easily read information from/write information to console (i.e. Command Line or Scientific Calculator Plus for JAVA), string and file. However, please note that Smart Calculator does not support reading / writing console functions.

To create, copy, move and delete files, MFP also provides a group of Dos (Unix) command-like functions. With authorization, user is able to create, read, modify, copy, move or delete any file or folder, just like working in a Dos or Unix command line window.

Section 1 Input and Output in Command Line

MFP provides the following functions to input/output in Command Line or Scientific Calculator Plus for JAVA:

Function Name	Function Info
input	input(2) : input(prompt,input_type) prints string prompt in the console and waiting for user to input. The second parameter, input_type, is optional. At this stage only if the second parameter exists and it is string ″s″ or ″S″, user's input is looked on as a string and this function returns the string. Otherwise, input is treated as an expression and this function returns the value of the expression. If input is not a valid expression, this function will reprint the prompt and wait for user to input again. An input is finished by pressing ENTER key. If multiple lines are input, only the first line is read. For example, user runs input(″$″, ″S″), then types 4 + 3 after prompt, presses ENTER, this function returns a string ″4 + 3″. If user runs input(″%″), then types 4 + 3 after prompt, presses ENTER, this function returns 7.

pause	pause(1) : pause(message) suspends current running program waiting for an ENTER input by user. Message, which is a string and is optional, will be printed on screen as a prompt if provided.
print	print(1) : print(x) prints the value of x to output, x can be any value type.
printf	printf(1...) : printf(format_string, ...), sprintf(format_string, ...) and fprintf(fd, format_string, ...) work like corresponding C/C++ functions. Function printf prints formatted string constructed from format_string and other parameter values to output console, sprintf constructs a new string from format_string and other parameters, and returns the new string, fprintf prints the formated string from format_string and other parameter values to the text file whose id is fd. The format_string parameter supports integer (%d, %i, %x, etc), float (%e, %f, etc), character (%c), string (%s) etc. User can find detailed information for construction of a format string by reading C language manual for these functions. For example, printf("Hello world!%f", 3.14) will output "Hello world!3.140000" on the screen, sprintf("%c%d", "A", 9) returns "A9" (MFP does not support single character type, so single character is stored as a one-char string).
scanf	scanf(1) : scanf(format_string), sscanf(input_from, format_string) and fscanf(fd, format_string) work like corresponding C/C++ functions. Function scanf reads one line input from user, sscanf reads string based parameter input_from, and fscanf reads content from a file whose id is fd. The format_string parameter supports integer

	(%d, %i, %x, etc), float (%e, %f, etc), character (%c), string (%s) etc. User can find detailed information for construction of a format string by reading C language manual for these functions. Different from C language, these functions do not accept additional parameters to store read values. These functions simply return all the read values in an array. For example, sscanf(″3Hello world!″, ″%d%c%c%s″) returns [3, ″H″, ″e″, ″llo″] (MFP does not support single character type, so single character is stored as a one-char string).

As shown in the above table, functions input, pause and print are easy to use. The only thing to note is function print only accepts a string based parameter. Since a string plus any value is a string, user may use an empty string plus a data value as the parameter to print. For example, to print value of array [1,2,3+4i], user may call print(""+[1,2,3+4i]) so that [1,2,3+4i] is converted to string [1, 2, 3 + 4i] and then is printed on the screen. Also note that, some special characters, e.g. double-quote " and backward slash \, need to escape (i.e. add a \ before the character) in string. For example,

print("\"\\")

will print "\ on the screen. This escaping requirement is applied to all the functions using string based parameters, including functions print, input, pause and printf and scanf which will be introduced later.

The following example is for the above functions. It can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder:

```
Help

@language:

  test input, pause and print functions

@end

@language:simplified_chinese

  测试 input，pause 和 print 函数

@end

endh

function io2console1()
```

```
 variable a = input("Please input a number") //input a number

 variable b = input("Please input a string:", "S") //input a string

 pause("Press ENTER to continue!") //press ENTER to continue

 //print what has been input

 print("You have input " + a + b)

 //print special character.

 print("\n\"\\")

endf
```

When running the above example, user inputs sind(30)*2+3i after the first prompt, and inputs a string “Is a complex value”, then press ENTER and sees the output is:

```
You have input 1 + 3iIs a complex value

"\
```

Functions printf and scanf are used in formatted input/output. Their usage is similar to the corresponding C language functions. The way to call printf() function is:

Printf("<format string>", <argument list>)

. The format string includes two parts. The first part includes normal characters. These characters will be output as they are shown in the format string. The second part includes special characters. They immediately follow a “%”. Character(s) following a “%”generally determine output format of a value. The only exception is character “%” which needs another “%” before it to escape. And instead of controlling the output format of a value, it simply means printing a “%” on the screen.

Argument list is a list of to-be-output values. The number and order of values must match the definition of format string. Otherwise error will be reported.

Below is a list of supported output formats of printf:

%d: means a value is output as a decimal integer;

%f: means a value is output as a floating value;

%s: means a value is output as a string;

%e: means a value is output using scientific notation;

%x and %X: means a value is output as a hexadecimal integer. Note that there is no 0x initial in the output. For example, in MFP, hexadecimal integer FF (i.e. 255) should be written as 0xFF. However, if user calls printf to output decimal integer 255 using %x or %X format, FF instead of 0xFF will be printed;

%o: means a value is output as an octal integer. Note that there is no 0 initial in the output. For example, in MFP octal integer 77 (i.e. 63) should be written as 077. However, if user calls printf to output decimal integer 63 using %o format, 77 instead of 077 will be printed;

%g: select a suitable format for a real value. Note that string is not supported.

For example, if user wants to output 8.776 followed by a comma and then followed by a string "Hello", and then followed by a blank character, and then output 1234 using hexadecimal format, then output three letters which are abc, then output 255 as a normal decimal integer, finally output a string "finish", function printf can be called as below:

```
printf("%f,%s %Xabc%dfinish",8.776,"Hello",1234,255)
```

The output of the above statement is:

```
8.776000,Hello 4D2abc255finish
```

Note that in the output result, hexadecimal integer does not start with 0x (i.e. 1234 is output as 4D2 not 0x4D2).

Basically, values in the argument list will be formatted and then output by the format string. If a value cannot be directly output using the desired format, printf will try to convert the value to another data type and try again. If any value cannot be formatted even after data type conversion, an error will be triggered. For example,

```
printf("%d", 123)
```

and

```
printf("%f", 123.1)
```

are both right because 123 is an integer and %d is the format indicator. And 123.1 is a floating value and %f is exactly the format indicator to output floating value.

For another example,

```
printf("%f", 123)
```

and

```
printf("%d", 123.1)
```

are also both right because 123 can be automatically converted to a floating value, and 123.1 can be rounded to an integer. However,

```
printf("%d", 123 + 123i)
```

and

```
printf("%f", [1.2,2.3])
```

are wrong because 123 + 123i is a complex value which cannot be converted to integer . [1.2, 2.3] is an array which cannot be converted to a single value. So if user wants to output complex or array, real and image parts or each element in the array must be output separately, like follows:

```
printf("%d + %di", 123, 123)  //output 123 + 123i

printf("[%f, %f]", 1.2, 2.3)  //output [1.200000, 2.300000]
```

. Alternatively, a trick to output them as string (i.e. using %s) can be played:

```
printf("%s", 123 + 123i)  //output 123 + 123i

printf("%s", [1.2,2.3])  //output [1.2, 2.3]
```

. Since any value in MFP can be converted to a string, this trick always works.

In order to accurately control the output format, user may insert some characters between "%" and following format definition character(s). For example, user may insert a positive integer value between "%" and following format definition character to tell printf function what's the maximum output length. For example:

%3d means outputting 3 or more digit integer, and the output is right-aligned if the integer includes less than 3 digits;

%9.2f means outputting 9 or more digit floating value, 2 digits after decimal point and 6 digits before. Decimal point takes one digit. The output is right-aligned if the floating value has less than 9 characters;

%8s means outputting an 8 or more character string. Blank characters are padded in front of the string if it includes less than 8 characters.

In the above examples, if the length of an integer or a string is longer than the specified output length, the full integer or string will be printed regardless of the output length setting; if the length of the integer part of a floating value is longer than the specified integer part of output length, the full integer part will be printed regardless of the setting; if the length of the fractional part of a floating value is longer than the specified fractional part of output length, the fractional part will be rounded.

If user wants to pad 0 before the output, s\he can add a zero before the output length setting. For example, %04d means 0s are padded in front of an integer's output to make the total output length is 4 if the integer's length is less than 4.

If output length setting for string is a floating value, the integer part of the floating value means the minimum output length, and the fractional part means the maximum output length. If the string's length is larger than the maximum output length, the output will be truncated. For example, %6.9s means outputting a string whose length is no less than 6 and no greater than 9. If a string includes less than 6 characters, blank will be padded in front of it; if a string includes more than 9 characters, the characters after character no. 9 will be discarded.

Function printf supports some special characters including \n (new line), \r (return) and \t (tabulator). These characters can be used in the format string parameter. For example:

```
printf("abc\ndef")
```

outputs two lines which are abc and def because \n means changing to a new line.

Opposite to printf, function scanf reads a line of user's input from console. Reading terminates at the place where an ENTER key is pressed, i.e. the end of the line, and returns an array whose elements are the values user input. Scanf only has one parameter which is the input format string. This parameter tells scanf what type of data the input should be recognized. Basically function scanf supports the following input formats:

%d: input a decimal integer;

%f and %e: input a floating value;

%s: input a string;

%x: input a hexadecimal integer. Note that there should be no 0x initial, e.g. 0x10AB should be input as 10AB. Otherwise, scanf cannot recognize it. This is different from the format of hexadecimal values in MFP, but matches the requirement of printf function. Also note that x in %x must be a small letter and %X is not supported.

%o: input an octal integer. Note that there should be no 0 initial, e.g. 017 should be input as 17. Otherwise, scanf may not recognize it. This is different from the format of octal values in MFP, but matches the requirement of printf function.

%g: input a decimal value.

Scanf's format string may include some characters between the format indicators, i.e. %d, %f, etc. When scanf reads the input line, it ignores these characters and read input after them. The delimiters can include both special characters like space, \t, \n, and \r and normal characters like a, b, c, etc. Note that same as printf, some characters need escaping, i.e. % must be written as %% in the format string, \ must be written as \\ and " must be written as \". A blank char (i.e. space, \t, \n and \r) delimiter results in ignoring one or several consecutive blank characters. Otherwise, scanf just ignores the same character(s) as in the delimiter. If no delimiter between two format indicators, scanf will skip 0, 1 or several adjacent blank characters in the format string after reading the first input value and before reading the second input value.

The strategy of scanf is reading until cannot read. For example, if the format string is "%d%s" and user's input is 123abd, scanf first reads the digits, i.e. 123, because the first format indicator is %d. Then scanf sees character a but clearly a cannot be a part of decimal integer. Therefore, scanf tries to use %s, which is immediately after %d, to match it. %s means a string. As such scanf will read all the characters from a to the end of the line or until a blank character is found. Thus abd is formatted as a string input.

If during reading, scanf finds nothing that can be input for a format indicator, it will return an array including all the read data values. For example, the format string of scanf is "%f %d%s" and user's input is 1.23   xabd. The first format indicator is %f which corresponds to 1.23. Then there are blanks which will be looked on as a delimiter. Then scanf tries to read a decimal integer corresponding to %d. However, no decimal integer can be read after the blanks in the input (i.e. 1.23 xabd) so that scanf returns an array which is [1.23].

The following example shows how scanf works. The statement is

Scanf("%f\n%x%dABC%s%d%s %e")

. Assume user inputs

8.67   6E8d 232ABC hello 12

. The returned value of scanf is [8.67, 28301, 232, "hello", 12, ""]. Note that in the format string after ABC scanf expects a string. However, in the input after ABC is blank which can only be looked on as a (part of) delimiter. Thus scanf skips the blank so that string "hello" not " hello" is input. In the end of the format string there is a format indicator which is %e. However, no corresponding input can be found so that scanf returns.

Format indicator can also tell scanf at most how many characters should be read. In the following example

scanf("%3f %2s")

, %3f means at most 3 characters should be read to input a floating value. Similarly, %2s means at most 2 characters should be read for the string. If user inputs

1235.324 aerf

, the output would be [123, "5."] because %3f only reads 3 characters from user's input so that 123 is read. Then scanf finds a blank in the format string. As explained above, a blank character delimiter may mean 0 or 1 or several consecutive blank character(s) in the input. Since no blank character after 3, this delimiter means zero blank character so that scanf tries to read two characters to match %2s then. The string input therefore is "5." and it is returned by function scanf with 123.

Also note that, different from printf, scanf does not automatically convert data types. Only if user's input exactly matches the format indicator scanf can read. Otherwise it stops reading input for this format indicator. For example,

Scanf("%d,%f")

tries to read an integer and a floating value from input. If user's input is

12.34,5

, scanf can only return [12] because 12.34 is a floating value which cannot be automatically rounded to an integer. Thus when scanf reads an integer for %d, it stops at decimal point, i.e. ".". However, scanf finds that after %d a "," exists in the format string as a delimiter. However, decimal point is not a comma so that reading fails. So the final return is only [12].

The following example is for the above functions. It can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder:

```
Help

@language:

  test printf and scanf functions

@end

@language:simplified_chinese

  测试 printf 和 scanf 函数

@end

endh
```

```
function printfscanf()
 printf("Now test printf function\n")
 printf("%f,%s %Xabc%dfinish",8.776,"Hello",1234,255)
 printf("\n")
 printf("%d", 123)
 printf("\n")
 printf("%f", 123.1)
 printf("\n")
 printf("%f", 123)
 printf("\n")
 printf("%d", 123.1)
 printf("\n")
 printf("%d + %di", 123, 123)  //print 123 + 123i
 printf("\n")
 printf("[%f, %f]", 1.2, 2.3)  //print [1.200000, 2.300000]
 printf("\n")
 printf("%s", 123 + 123i)  //print 123 + 123i
 printf("\n")
 printf("%s", [1.2,2.3])  //print [1.2, 2.3]
 printf("\n")
 printf("%3s", "abcdefg")
 printf("\n")
 printf("%019.6f", 12.2342154577) // print 000000000012.234215
 printf("\n")
 printf("abc\ndef")
 printf("\n")
 printf("Now test scanf function\n")
 variable result_of_scanf
 printf("Please input:\n123abd\n")
 result_of_scanf = scanf("%d%s")
 // remember, to simply print % using printf, we need to use %%
 printf("scanf(\"%%d%%s\") returns " + result_of_scanf)
 printf("\n")
 
 printf("Please input:\n1.23  xabd\n")
 result_of_scanf = scanf("%f %d%s")
 print("scanf(\"%f %d%s\") returns " + result_of_scanf)
 printf("\n")
 
 printf("Please input:\n8.67   6E8d 232ABC hello 12\n")
 result_of_scanf = scanf("%f\n%x%dABC%s%d%s %e")
 print("scanf(\"%f\n%x%dABC%s%d%s %e\") returns " + result_of_scanf)
 printf("\n")
 
 printf("Please input:\n1235.324 aerf\n")
 result_of_scanf = scanf("%3f %2s")
 print("scanf(\"%3f %2s\") returns " + result_of_scanf)
 printf("\n")
 
 printf("Please input:\n12.34,5\n")
 result_of_scanf = scanf("%d,%f")
 print("scanf(\"%d,%f\") returns " + result_of_scanf)
 printf("\n")
endf
```

Running function printfscanf() in Command Line and the result is shown below. Note that the result includes both user’s input and scanf’s output.

```
Now test printf function

8.776000,Hello 4D2abc255finish

123

123.100000

123.000000

123

123 + 123i

[1.200000, 2.300000]

123 + 123i

[1.2, 2.3]

abcdefg

000000000012.234215

abc

def

Now test scanf function

Please input:

123abd

123abd

scanf("%d%s") returns [123, "abd"]

Please input:

1.23  xabd

1.23  xabd

scanf("%f %d%s") returns [1.23]
```

```
Please input:

8.67   6E8d 232ABC hello 12

8.67   6E8d 232ABC hello 12

scanf("%f

%x%dABC%s%d%s %e") returns [8.67, 28301, 232, "hello", 12, ""]

Please input:

1235.324 aerf

1235.324 aerf

scanf("%3f %2s") returns [123, "5."]

Please input:

12.34,5

12.34,5

scanf("%d,%f") returns [12]
```

Section 2 Input from and Output to String

Similar to printf and scanf, MFP programming language provides sprintf and sscanf to output to and input from string. Sprintf has very similar usage to printf except that printf does not have any returned value because all the output is printed on the screen while sprint returns the output string.

Sscanf is also similar to scanf. Their difference is, first, sscanf needs two parameters, first is input string and second is the format string; and second, input string of sscanf may have multiple lines while scanf can only read one line from console.

The following example is for the above functions. It can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder:

```
Help

@language:

  test sprintf and sscanf functions

@end

@language:simplified_chinese
```

```
  测试 sprintf 和 sscanf 函数
@end
endh
function sprintfsscanf()
 variable result_str
 printf("Now test sprintf function\n")
 result_str = sprintf("%f,%s %Xabc%dfinish",8.776,"Hello",1234,255)
 printf(result_str + "\n")
 result_str = sprintf("%d", 123)
 printf(result_str + "\n")
 result_str = sprintf("%f", 123.1)
 printf(result_str + "\n")
 result_str = sprintf("%f", 123)
 printf(result_str + "\n")
 result_str = sprintf("%d", 123.1)
 printf(result_str + "\n")
 result_str = sprintf("%d + %di", 123, 123)  //return 123 + 123i
 printf(result_str + "\n")
 result_str = sprintf("[%f, %f]", 1.2, 2.3)  //return [1.200000, 2.300000]
 printf(result_str + "\n")
 result_str = sprintf("%s", 123 + 123i)  //return 123 + 123i
 printf(result_str + "\n")
 result_str = sprintf("%s", [1.2,2.3])  //return [1.2, 2.3]
 printf(result_str + "\n")
 result_str = sprintf("%3s", "abcdefg")
 printf(result_str + "\n")
 result_str = sprintf("%019.6f", 12.2342154577) // return 000000000012.234215
```

```
printf(result_str + "\n")

result_str = sprintf("abc\ndef")

printf(result_str + "\n")


printf("Now test sscanf function\n")

variable result_of_sscanf

result_of_sscanf = sscanf("123abd","%d%s")

// remember, to simply print % using printf, we need to use %%

printf("scanf(\"123abd\",\"%%d%%s\") returns " + result_of_sscanf)

printf("\n")


result_of_sscanf = sscanf("1.23  xabd","%f %d%s")

print("scanf(\"1.23  xabd\",\"%f %d%s\") returns " + result_of_sscanf)

printf("\n")


// read string including multiple lines

result_of_sscanf = sscanf("8.67   6E8d 232ABC\nhello 12", _

  "%f\n%x%dABC%s%d%s %e")

print("scanf(\"8.67    6E8d 232ABC\\nhello 12\",\"%f\n%x%dABC%s%d%s %e\") returns " + result_of_sscanf)

printf("\n")


result_of_sscanf = sscanf("1235.324 aerf","%3f %2s")

print("scanf(\"1235.324 aerf\",\"%3f %2s\") returns " + result_of_sscanf)

printf("\n")


result_of_sscanf = sscanf("12.34,5","%d,%f")

print("scanf(\"12.34,5\",\"%d,%f\") returns " + result_of_sscanf)
```

```
  printf("\n")

endf
```

Run function sprintfsscanf() in Command Line and the output is shown below. Note that different from the example for scanf, no user input is required here.

```
Now test sprintf function

8.776000,Hello 4D2abc255finish

123

123.100000

123.000000

123

123 + 123i

[1.200000, 2.300000]

123 + 123i

[1.2, 2.3]

abcdefg

000000000012.234215

abc

def

Now test sscanf function

scanf("123abd","%d%s") returns [123, "abd"]

scanf("1.23  xabd","%f %d%s") returns [1.23]

scanf("8.67   6E8d 232ABC\nhello 12","%f

%x%dABC%s%d%s %e") returns [8.67, 28301, 232, "hello", 12, ""]

scanf("1235.324 aerf","%3f %2s") returns [123, "5."]

scanf("12.34,5","%d,%f") returns [12]
```

Section 3 Functions to Read and Write File

MFP programming language provides user a set of C-like file reading and writing functions. They include text file reading/writing functions, i.e. fprintf and fscanf (similar usage to printf and scanf), line by line text file reader freadline, binary file reading/writing functions, i.e. fread and fwrite, and fopen and fclose to open and close file respectively. The functions are listed in the following table:

Function Name	Function Info
fclose	fclose(1) : fclose(fd) closes the file whose id is fd. If fd is invalid, returns -1. Otherwise, returns 0.
feof	feof(1) : feof(fd) identifies if it has been the end of a read mode file whose id is fd. If so, returns true. Otherwise, returns false. If fd is invalid, throws an exception.

fopen	fopen(2) :
	fopen(path, mode) opens file at path to read or write and returns the file' s id number. It is similar to C and Matlab's same name function. However, only ″r″, ″a″, ″w″, ″rb″, ″ab″ and ″wb″ modes are supported. Examples are fopen(″C:\\Temp\\Hello.dat″, ″ab″) (Windows) and fopen(″./hello.txt″, ″r″) (Android).
	fopen(3) :
	fopen(path, mode, encoding) opens a text file at path using character set encoding to read or write and returns the file id number. Because only text file supports encoding, parameter mode can only be ″r″, ″a″ and ″w″. Examples are fopen(″C:\\Temp\\Hello.txt″, ″a″, ″LATIN-1″) (Windows) and fopen(″./hello.txt″, ″r″, ″UTF-8″) (Android).

fprintf	fprintf(2...) : printf(format_string, ...), sprintf(format_string, ...) and fprintf(fd, format_string, ...) work like corresponding C/C++ functions. Function printf prints formatted string constructed from format_string and other parameter values to output console, sprintf constructs a new string from format_string and other parameters, and returns the new string, fprintf prints the formatted string from format_string and other parameter values to the text file whose id is fd. The format_string parameter supports integer (%d, %i, %x, etc), float (%e, %f, etc), character (%c), string (%s) etc. User can find detailed information for construction of a format string by reading C language manual for these functions. For example, printf(″Hello world!%f″, 3.14) will output ″Hello world!3.140000″ on the screen, sprintf(″%c%d″, ″A″, 9) returns ″A9″ (MFP does not support single character type, so single character is stored as a one-char string).
fread	fread(4) : fread(fd, buffer, from, length) reads length number of bytes from file whose id is fd and stores the bytes in buffer starting from parameter from. Note that from and length must be non-negative and from + length should be no greater than buffer size. From and length are optional. If they do not exist, fread reads buffer size of bytes and fills the buffer. Buffer is also optional. If Buffer does not exist, fread returns a single byte. If fread finds that it is at the end of the file before reading, it returns -1. Otherwise, if using buffer, it returns the number of bytes that are read (if buffer is provided). If file does not exist, or is invalid or inaccessible, exception will be thrown. Examples are fread(1), fread(2, byte_buffer) and fread(2, byte_buffer, 3, 7).

freadline	freadline(1) : freadline(fd) reads one line from text file whose id is fd. If before reading, freadline finds it is at the end of the file, it returns NULL. Otherwise, it returns the string based line excluding the change-line character(s) at the end.
fscanf	fscanf(2) : scanf(format_string), sscanf(input_from, format_string) and fscanf(fd, format_string) work like corresponding C/C++ functions. Function scanf reads one line input from user, sscanf reads string based parameter input_from, and fscanf reads content from a file whose id is fd. The format_string parameter supports integer (%d, %i, %x, etc), float (%e, %f, etc), character (%c), string (%s) etc. User can find detailed information for construction of a format string by reading C language manual for these functions. Different from C language, these functions do not accept additional parameters to store read values. These functions simply return all the read values in an array. For example, sscanf(″3Hello world!″, ″%d%c%c%s″) returns [3, ″H″, ″e″, ″llo″] (MFP does not support single character type, so single character is stored as a one-char string).
fwrite	fwrite(4) : fwrite(fd, buffer, from, length) writes length number of bytes to file whose id is fd. The data to write store in parameter buffer starting from parameter from. Note that from and length must be non-negative and from + length should be no greater than buffer size. From and length are optional. If they do not exist, fwrite writes whole buffer to file. Buffer can also be a single byte which means fwrite writes only 1 byte to file. If file does not exist, or is invalid or inaccessible, exception will be thrown. Examples are

	fwrite(1, 108), fwrite(2, byte_buffer) and fwrite(2, byte_buffer, 3, 7).

1. Open and Close File

Similar to C language, the first step in MFP's file operating routine is to open file. The function to open a file is fopen. This function has three parameters. The first one is string based file name (path). Here path can be either an absolute path (e.g. "/mnt/sdcard/a.txt" or "C:\\temp\\b.exe") or a relative path (e.g. "a.txt" or "temp\\b.exe"). Note that relative path is relative to the current working folder. In Android, the working folder when Scientific Calculator Plus starts is the AnMath folder in SD card. In PC, the working folder when Scientific Calculator Plus starts is where the JCmdLine.jar file is located. After Scientific Calculator Plus starts, user is able to change working folder using cd function.

The second parameter of fopen function is the mode to open the file. This parameter is also a string and has six choices which are:

"r": open a text file to read. Generally a text file's extension is .txt, but can also be .c, .cpp, .mfps, .xml etc. These files can be opened and edited by notepad;

"w": open a text file to write. The original content will be wiped off;

"a": open a text file to append. The input will be appended to the tail of the text file;

"rb": open a binary file to read. Generally binary files cannot be opened and edited by notepad;

"wb": open a binary file to write. The original content will be wiped off;

"ab": open a binary file to append. The input will be appended to the tail of the binary file;

Note that functions for text files are different from functions for binary files. Reading/writing text files should call function fprintf, fscanf and freadline, while reading/writing binary files should call fread and fwrite. If user uses text file function to read/write a binary file or vice versa, an error will be reported.

Besides the above two parameters, fopen has another optional parameter which is the encoding mode of the file. This parameter is only useful to text files. It is quite important to open a text file using correct encoding mode. Otherwise, user may see it just includes bad codes.

The default encoding mode of fopen function is OS's encoding mode. For example, if user is using simplified Chinese Windows, encoding is GB2312. This encoding mode fully supports simplified Chinese characters so that if user calls fopen to open a text file to write some Chinese words into it without an encoding mode parameter, and then opens it again in Nodepad, the Chinese words are still there. However, English Windows's character set is IBM437 which does not support Chinese characters. If user calls fopen to write some Chinese words in an English Windows system, and then reopens it again in Nodepad, nothing but bad codes or question marks are found.

This problem is not that severe in Android because Android's system adopts UTF-8 as encoding mode. UTF-8 supports Chinese as well as many other character sets. As such text files created by fopen in an Android device generally have no problem to open by a text editor on Windows.

So if user of MFP programming language wants to develop a script to create or modify some text files and share the script with other people using different languages, encoding mode parameter of fopen function cannot be neglected and encoding mode "UTF-8" is strongly recommended.

If fopen opens a file successfully, it returns a non-negative integer which is the file id. Every time fopen is called, whether operating the same file or not, a different file id is returned. The returned file id will be a parameter for the following file reading/writing/closing functions. Without this parameter, file reading/writing/closing functions are not able to know which file to operate.

After file is opened and read or written, user needs to close it to release the resource. The function to close an open file is fclose(fd) where fd is the returned file id from fopen.

Fclose function is not negligible. If a file is opened but not closed, first memory is leaked; second future writing operations on this file may fail.

Here examples are provided for fopen/fclose functions:

variable fd1 = fopen("test.exe","wb") // open binary file test.exe to write.

fclose(fd1)

variable fd2 = fopen("/mnt/sdcard/AnMath/test.txt","r","UTF-8") // open text file test.txt using UTF-8 encoding mode to read.

fclose(fd2)

2. Read and Write Text File

So far, MFP only provides sequentially reading / writing functions for text files. This means after a text file is opened, user can only read or write the file towards the end of the file. User cannot jump to an arbitrary point and start reading or writing. When reading arrives at the end of the file, it stops. MFP provides feof function to identify if it is at the end of the file. Feof has one parameter which is file id. It returns true if it is at the end of the file, or false otherwise.

Text file reading functions provided by MFP are fscanf and freadline. The usage of fscanf is similar to functions scanf and sscanf which have been introduced before. Fscanf needs two parameters, the first one is file id, i.e. fopen's return value; the second one is format string. Fscanf returns an array whose elements are input values from the file. Note that the file may include multiple lines and the changing line characters are treated as a type of blanks.

Function freadline reads a text file line by line. Its usage is much simpler than fscanf. It only needs one parameter which is file id. Every time it is called, it returns a line of the file and sets file reading pointer to the beginning of the next line. If it arrives at the end of the file, next time it is called it returns an empty string.

The following example demonstrates user how to read text file using the above two functions. It can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder.

Before running this example, user needs to create a text file named test_read.txt in the io and file libs folder (i.e. the same folder of the source file). The content of the file is:

123 456

Good,2*9=18

Hello!

abc

. The MFP code to read the file is shown below:

```
Help

@language:

  test reading text file

@end

@language:simplified_chinese

  测试读取文本文件

@end

endh

function readTextFile()

  // here assume current directory is AnMath and test_read.txt
  // is saved in scripts/manual/io and file libs/ folder.
  // Content of the file :
  //123 456
  //Good,2*9=18
  //Hello!
  //abc
  variable fd = fopen("scripts/manual/io and file libs/test_read.txt", "r")
  // read two integers
  variable int_array = fscanf(fd, "%d%d")
  // print what has been read
  printf("read " + int_array + "\n")
  variable read_str = ""
  // if we are not at the end of the file and we haven't read string Hello!
  while and(feof(fd) == false, read_str != "Hello!")
    read_str = freadline(fd)
    // print what has been read
    printf("read " + read_str + "\n")
  loop
  variable str_array = fscanf(fd, "%s")
  // print what has been read
  printf("read " + str_array + "\n")
  if (feof(fd))
    // it is right if we are at the end of the file
    print("End of the file found!\n")
  else
    // it is wrong if we are still not at the end of file
    print("Something wrong!\n")
  endif
  fclose(fd) //don’t forget to close the file
endf
```

The output of the above example is:

```
read [123, 456]
read
read Good,2*9=18
read Hello!
read ["abc"]
End of the file found!
```

In the above code, after reading 123 and 456, freadline is called in the while loop to read the file line by line. However, the first line freadline returns is not Good,2*9=18, but an empty string. The reason is, before freadline is called, fscanf only moves reading pointer to the end of 456 and returns. This means the changing line character hasn’t been read. As such freadline starts reading the character after 6 until it sees a changing line character. However, no character is between 6 and changing line so that it returns an empty string.

The text file writing function that MFP provides is fprintf. This function is very similar to printf and sprintf. The only difference is, fprintf has an extra parameter, i.e. its first parameter, which is the file id returned from fopen. The second parameter is format string and the following parameters (if there are) are the values which will be written into the file.

The following example demonstrates user how to write text file. It can be found in the examples.mfps file in io and file libs sub-folder in the manual’s sample code folder.

```
Help
@language:
  test writing text file
@end
@language:simplified_chinese
  测试写入文本文件
```

```
@end

endh

function writeTextFile()

  // first test write to replace mode. If file does not exist, it will be

  // created. If file does exist, its content will be wiped off.

  variable fd = fopen("scripts/manual/io and file libs/test_write.txt", _

     "w", "UTF-8")

  // first inputs some numbers and string with prompt information

  fprintf(fd, "The first line includes %d, %f, %s\n", 123.71, 56.48, "hi")

  // then input 4 Chinese characters with prompt information

  fprintf(fd, "Now input some Chinese characters： " + "汉字中文\n")

  fclose(fd) // close the file



  // Then test append mode. If file does not exist, it will be

  // created. If file does exist, fprintf will append some text to its

  // current content.

  fd = fopen("scripts/manual/io and file libs/test_write.txt", _

     "a", "UTF-8")

  // inputs some numbers and string with prompt information

  fprintf(fd, "Now add a new line %d, %f, %s\n", -999, -48.73, "why?")

  fclose(fd) // close the file

endf
```

After running the above code, user will find a test_write.txt file is generated in the io and file libs folder. The content of the file is below:

```
The first line includes 123, 56.480000, hi

Now input some Chinese characters： 汉字中文

Now add a new line -999, -48.730000, why?
```

Because the code uses UTF-8 encoding mode to open the text file, these Chinese characters can be properly shown in any text editor, e.g. notepad++.

MFP doesn't support global variables. However, user is able to write and read the same text file in different functions. The text file works as if a global variable. The content of the file is the value of the global variable.

3. Read and Write Binary File

In MFP, the function to read binary file is fread. Fread has four parameters. The first one is file id (fd) which is the returned value from fopen. The second one is the buffer where the read bytes are stored. This parameter must be an array. The third one is the starting place in the buffer from where the read bytes are stored. The last parameter is the number of bytes to read. Note that the starting point in the buffer plus the number of bytes to read should not be larger than the buffer length.

The third and fourth parameters are optional. If they don't exist, the default starting point is 0 and number of bytes to read is the length of buffer. The second parameter is also optional. If it doesn't exist, fread simply reads one byte and returns the read byte. If it exists, fread stores all the read bytes in the buffer and returns the number of bytes it has read. If before reading fread finds it has arrived at the end of the file, fread returns -1. To avoid unnecessary reading when arriving at the end of the file, user can use feof function before calling fread. If the file to read does not exist, an exception is thrown.

MFP's binary file writing function is fwrite. Fwrite(fd, buffer, from, length) writes length bytes to the file whose id is fd. These bytes are stored in string buffer from index from. Note that from and length must be non-negative and from + length shouldn't be larger than buffer's capacity. Parameters from and length can be neglected together. If they don't exist, fwrite writes the whole buffer into the file. Buffer can also be a single byte instead of an array. In this case, fwrite simply writes the byte into the file. If the file doesn't exist or cannot be accessed, an exception will be thrown.

The following code is an example for the above functions. It can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder.

```
Help
@language:
  test reading & writing binary file
@end
@language:simplified_chinese
  测试读写二进制文件
```

```
@end
endh
function readWriteBinaryFile()
 // remember write binary file should use "wb" not "w"
 variable fd = fopen("scripts/manual/io and file libs/test_rw.bin", "wb")
 fwrite(fd, 108) // write one byte whose value is 108
 // note that buffer should be an array of bytes (i.e. integer whose
 // value is no less than -127 and no greater than 128). Here 1000
 // is larger than 128 so that it will be casted to a byte whose
 // value is -24 when write to a binary file. Its upper bits will lose
 variable buffer = [-18,79,126,-55,48,-23,-75,7,98,6,0,-34,1000]
 fwrite(fd,buffer) //write everything in the buffer into the file
 fclose(fd)

 // remember append binary file should use "ab" not "a"
 fd = fopen("scripts/manual/io and file libs/test_rw.bin", "ab")
 // write 7 bytes from index 3 of buffer
 fwrite(fd,buffer, 3, 7)
 fclose(fd)

 //print original buffer content
 print("Originally buffer includes " + buffer + "\n")
 // remember read binary file should use "rb" not "r"
 fd = fopen("scripts/manual/io and file libs/test_rw.bin", "rb")
 // read 5 bytes from file, and store the read bytes in buffer from
 // index 2
 fread(fd, buffer, 2, 5)
 print("Now buffer is " + buffer + "\n") //print the buffer's content
 variable byte_value = fread(fd) // read 1 byte
 print("Read one byte which is " + byte_value + "\n")
 variable read_byte_cnt = fread(fd, buffer) // try to read buffer length
                            // bytes
 print("Read " + read_byte_cnt + " bytes" + "\n") // print how many
                                // bytes read
 print("Now buffer is " + buffer + "\n") //print the buffer's content
 read_byte_cnt = fread(fd, buffer) // try to read buffer length bytes
                       // again
 print("Read " + read_byte_cnt + " bytes" + "\n") // print how many
                                // bytes read
 // check if we have arrived at the end of file
 if (feof(fd))
  // check how many bytes we can read if we have arrived at the
         // end of file
  print("We have arrived at the end of file.\n")
         print("Now check how many bytes can be read.\n")
  read_byte_cnt = fread(fd, buffer) // try to read buffer length
                       // bytes again
  print("Read " + read_byte_cnt + " bytes" + "\n") // print how
                                // many bytes
                                // read
 endif
 fclose(fd)
endf
```

The output of the example is:

```
Originally buffer includes [-18, 79, 126, -55, 48, -23, -75, 7, 98, 6, 0, -34, 1000]

Now buffer is [-18, 79, 108, -18, 79, 126, -55, 7, 98, 6, 0, -34, 1000]

Read one byte which is 48

Read 13 bytes

Now buffer is [-23, -75, 7, 98, 6, 0, -34, -24, -55, 48, -23, -75, 7]

Read 2 bytes

We have arrived at the end of file.

Now check how many bytes can be read.

Read -1 bytes
```

Section 4 Functions to Read and Modify File Properties

MFP programming language provides functions to help user read or even modify file properties including file name, path, type, size and last-modified time. The details of these functions are listed in the table below:

Function Name	Function Info
get_absolute_path	get_absolute_path(1) : get_absolute_path(fd_or_path) returns a string value which is the absolute path of the file either whose id number is fd_or_path (here fd_or_path is an integer) or whose relative path is fd_or_path (here fd_or_path is a string).
get_canonical_path	get_canonical_path(1) : get_canonical_path(fd_or_path) returns a string value which is the canonical path (path which is absolute and does not rely on symbol link) of file either whose id number is fd_or_path (here fd_or_path is an integer) or whose relative path is fd_or_path (here fd_or_path is a string).

get_file_last_modified_time	get_file_last_modified_time(1) : get_file_last_modified_time(path) returns the last-modified time of the file or folder corresponding to a string based path. The time is measured by the number of milliseconds since midnight on January 1st, 1970. If path does not exist or the file is not accessible, returns -1.
get_file_path	get_file_path(1) : get_file_path(fd) returns a string value which is the path of file whose id number is fd.
get_file_separator	get_file_separator(0) : get_file_separator() returns the separator used in path. In Windows it returns string ″\\″. In Linux or Android it returns string ″/″.
get_file_size	get_file_size(1) : get_file_size(path) returns the size of the file corresponding to a string based path. If path does not correspond to a file, or the file does not exist or the file is not accessible, it returns -1.
is_directory	is_directory(1) : is_directory(path) identifies if the file (or folder) at string based parameter path is a directory or not. If it exists and is a directory the function returns true, otherwise false. Examples are is_directory(″E:\\″) (Windows) and is_directory(″/home/tony/Documents/cv.pdf

	″) (Android).
is_file_executable	is_file_executable(1) : is_file_executable(path) identifies if the file (or folder) at string based parameter path is executable or not. If it exists and is executable the function returns true, otherwise false. Examples are is_file_executable(″E:\\″) (Windows) and is_file_executable(″/home/tony/Documents/cv.pdf″) (Android).
is_file_existing	is_file_existing(1) : is_file_existing(path) identifies if the file (or folder) at string based parameter path exists or not. If exists it returns true, otherwise false. Examples are is_file_existing(″E:\\″) (Windows) and is_file_existing(″/home/tony/Documents/cv.pdf″) (Android).
is_file_hidden	is_file_hidden(1) : is_file_hidden(path) identifies if the file (or folder) at string based parameter path is hidden or not. If it exists and is hidden the function returns true, otherwise false. Examples are is_file_hidden(″E:\\″) (Windows) and is_file_hidden(″/home/tony/Documents/cv.pdf″) (Android).
is_file_normal	is_file_normal(1) : is_file_normal(path) identifies if the file (or folder) at string based parameter path is a normal file (not a

	folder) or not. If it exists and is a normal file the function returns true, otherwise false. Examples are is_file_normal(″E:\\″) (Windows) and is_file_normal(″/home/tony/Documents/cv.pdf″) (Android).
is_file_readable	is_file_readable(1) : is_file_readable(path) identifies if the file (or folder) at string based parameter path is readable or not. If it exists and is readable the function returns true, otherwise false. Examples are is_file_readable(″E:\\″) (Windows) and is_file_readable(″/home/tony/Documents/cv.pdf″) (Android).
is_file_writable	is_file_writable(1) : is_file_writable(path) identifies if the file (or folder) at string based parameter path is writable or not. If it exists and is writable the function returns true, otherwise false. Examples are is_file_writable(″E:\\″) (Windows) and is_file_writable(″/home/tony/Documents/cv.pdf″) (Android).
is_path_absolute	is_path_absolute(1) : is_path_absolute(path) identifies if the string based path is an absolute path (i.e. not relative to current folder). If it is an absolute path the function returns true, otherwise false. Examples are is_path_absolute(″E:\\temp″) (Windows) and is_path_absolute(″Documents/cv.pdf″)

	(Android).
is_path_parent	is_path_parent(2) : is_path_parent(path1, path2) identifies if the string based path1 is the parent of string based path2. If it is returns true, otherwise false. Examples are is_path_parent(″E:\\temp″, ″E:\\temp\\..\\temp\\test″) (Windows) and is_path_parent(″.″, ″Documents/cv.pdf″) (Android).
is_path_same	is_path_same(2) : is_path_same(path1, path2) identifies if the string based path1 is actually the same as string based path2. If it is returns true, otherwise false. Examples are is_path_same(″E:\\temp″, ″E:\\temp\\..\\temp\\″) (Windows) and is_path_parent(″/home/tony/Documents″, ″Documents/″) (Android).
is_symbol_link	is_symbol_link(1) : is_symbol_link(path) identifies if the file (or folder) at string based parameter path is a symbol link or not. If it exists and is a symbol link the function returns true, otherwise false. Examples are is_symbol_link(″E:\\″) (Windows) and is_symbol_link(″/home/tony/Documents/cv.pdf″) (Android).
set_file_last_modified_time	set_file_last_modified_time(2) : set_file_last_modified_time(path, time) sets the last-modified time of the file or folder corresponding to a string based

	path to be time. Here time is measured by the number of milliseconds since midnight on January 1st, 1970. If path does not exist or the file is not accessible, it returns false. Otherwise, it returns true. Examples are set_file_last_modified_time("C:\\Temp\\Hello\\", 99999999) (Windows) and set_file_last_modified_time("./hello.txt", 1111111111) (Android).

The usage of the above functions is very simple. The functions starting with get_ only need one parameter. The parameter is either file id returned by fopen function, or string based file path. The return value of the functions is either a string based file path, or an integer which is last-modified time or file size.

The functions starting with is_ confirm a property of a file. They only have one parameter which is either file id or string based file path. The return value is either True or False.

The function starting with set_ is set_file_last_modified_time. This is the only function that can modify file's property. This function sets a file's last-modified time. It needs two parameters. The first one is file path, and the second one is the new modification time.

The following code is the example of the above functions. This example can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder.

```
Help
@language:
  test file properties operators
@end
@language:simplified_chinese
  测试读写文件属性的函数
@end
endh
function fileProperties()
  // Assume current working directory is AnMath in Android
  // or the folder where JCmdline.jar is located (for
  // Scientific Calculator for JAVA)
  print("Current working directory is "+get_working_dir()+"\n")
  variable retval
  // open current function's source code file
  variable strPath = "scripts/manual/io and file libs/examples.mfps"
  variable fd = fopen(strPath, "r")

  // get source code file's absolute path
  retval = get_absolute_path(fd)
  print("Current source file's absolute path is " + retval + "\n")

  fclose(fd)

  // get source code file's canonical path
  retval = get_canonical_path(strPath)
  print("Current source file's canonical path is " + retval + "\n")

  // get source code file's last-modified time
  retval = get_file_last_modified_time(strPath)
  print("Current source file's last modify time is " + retval + "\n")

  // set source code file's last-modified time to be 1970/01/01
  set_file_last_modified_time(strPath, 0)
  retval = get_file_last_modified_time(strPath)
  print("After set last modify time to be 0, " _
    + "current source file's last modify time is " + retval + "\n")

```

```
 // get source code file's size
 retval = get_file_size(strPath)
 print("Current source file's size is " + retval + "\n")

 // is source code file a directory?
 retval = is_directory(strPath)
 print("Is current source file a directory: " + retval + "\n")

 // is source code file executable?
 retval = is_file_executable(strPath)
 print("Is current source file executable: " + retval + "\n")

 // is source code file existing?
 retval = is_file_existing(strPath)
 print("Is current source file existing: " + retval + "\n")

 // is source code file hidden?
 retval = is_file_hidden(strPath)
 print("Is current source file hidden: " + retval + "\n")

 // is source code file normal?
 retval = is_file_normal(strPath)
 print("Is current source file normal: " + retval + "\n")

 // is source code file readable?
 retval = is_file_readable(strPath)
 print("Is current source file readable: " + retval + "\n")

 // is source code file writable?
 retval = is_file_writable(strPath)
 print("Is current source file writable: " + retval + "\n")

 // is source code file path absolute?
 retval = is_path_absolute(strPath)
 print("Is current source file path absolute: " _
   + retval + "\n")

 // is source code file path a symbol link?
 retval = is_symbol_link(strPath)
 print("Is current source file path symbol link: " _
   + retval + "\n")

 // is path1 the parent of source code file path?
 Variable strPath1 = "scripts/manual/io and file libs"
 retval = is_path_parent(strPath1, strPath)
 print("Is " + strPath1 + " parent of " + strPath + " : " _
   + retval + "\n")

 // is path2 the same as source code file path?
 Variable strPath2 = "scripts/./../scripts/manual/../manual/io and file libs/examples.mfps"
 retval = is_path_same(strPath2, strPath)
 print("Is " + strPath2 + " same as " + strPath + " : " _
   + retval + "\n")
endf
```

Output of the above example is shown below. Please note that if the working directory is not AnMath (on Android) or the folder where JCmdLine.jar is located (in PC), running the above example user may see a very different result, or even some errors.

```
Current working directory is
E:\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime

Current source file's absolute path is
E:\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scrip
ts\manual\io and file libs\examples.mfps

Current source file's canonical path is
E:\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scrip
ts\manual\io and file libs\examples.mfps

Current source file's last modify time is 1439531707807

After set last modify time to be 0, current source file's last modify time is 0

Current source file's size is 16761

Is current source file a directory: FALSE

Is current source file executable: TRUE

Is current source file existing: TRUE

Is current source file hidden: FALSE

Is current source file normal: TRUE

Is current source file readable: TRUE

Is current source file writable: TRUE

Is current source file path absolute: FALSE

Is current source file path symbol link: FALSE

Is scripts/manual/io and file libs parent of scripts/manual/io and file
libs/examples.mfps : TRUE

Is scripts/./../scripts/manual/../manual/io and file libs/examples.mfps same as
scripts/manual/io and file libs/examples.mfps : TRUE
```

Section 5 Functions Similar to Dos and Unix Commands

The above functions only operate on a single file and none of them can delete a file. If user wants to read the file name list in a folder, or wants to move a folder, the above functions cannot be used.

User can use Dos or Unix command to do this work. For example, pwd returns current folder, cd changes current working folder, xcopy or cp copies file, etc. MFP also provides below functions to mimic the OS commands so that user can work in the Command Line as if in a Unix terminal.

Function Name	Function Info
cd	cd(1) : change_dir(path) (with alias cd(path)) changes current directory to string based value path. If successful, it returns true. Otherwise, it returns falses. Examples are change_dir("D:\\Windows") (Windows) and cd("/") (Android).
change_dir	change_dir(1) : change_dir(path) (with alias cd(path)) changes current directory to string based value path. If successful, it returns true. Otherwise, it returns falses. Examples are change_dir("D:\\Windows") (Windows) and cd("/") (Android).
copy_file	copy_file(3) : copy_file(source, destination, replace_exist) copies file or folder whose path is string source to file or folder whose path is string destination. If the 3rd parameter, replace_exist, is true, then source file (or any file in source folder) will replace destination file (or corresponding file in destination folder) if destination exists. Note that the 3rd parameter is optional. By default it is false.

	Examples are copy_file(″c:\\temp\\try1″, ″D:\\″, true) (Windows) and copy_file(″/mnt/sdcard/testfile.txt″, ″./testfile_copy.txt″) (Android).
create_file	create_file(2) : create_file(path, is_folder) creates a file (if is_folder is false or does not exist) or folder (if is_folder is true). If the parent of string based parameter path does not exist, the parent will be created. If the file can be created, this function returns true, otherwise, returns false. Examples are create_file(″c:\\temp\\try1″, true) (Windows) and create_file(″testfile_copy.txt″) (Android).
delete_file	delete_file(2) : delete_file(path, delete_children_in_folder) deletes a file or folder at string based parameter path. If it is a folder and the second parameter is true, all the files in the folder will be recursively deleted. The second parameter is optional. By default, it is false. If the file or folder can be deleted, this function returns true, otherwise, it returns false. Examples are delete_file(″c:\\temp\\try1″, true) (Windows) and delete_file(″testfile_copy.txt″) (Android).
dir	dir(1) : print_file_list(path) (alias ls(path) or dir(path)) works like ls command in Linux or dir command in Windows. It prints the information for the file or all the files in folder at string based path. It returns the number of entries printed. If the path does not correspond to an existing file or folder, it returns -1. Note that path is optional. By default it is

	current folder (″.″). Examples are dir() (Windows) and ls(″../testfile_copy.txt″) (Android).
get_working_dir	get_working_dir(0) : get_working_dir() (with alias pwd()) returns string based current directory.
list_files	list_files(1) : list_files(path) returns all the file names in the folder whose path is the string based parameter path, or it returns the file at path if path corresponds to a file. If the path does not correspond to a file or folder, it returns NULL. Note that parameter path is optional. By default it is current folder (″.″). Examples are list_files(″c:\\temp\\try1″) (Windows) and list_files(″../testfile_copy.txt″) (Android).
ls	ls(1) : print_file_list(path) (alias ls(path) or dir(path)) works like ls command in Linux or dir command in Windows. It prints the information for the file or all the files in folder at string based path. It returns the number of entries printed. If the path does not correspond to an existing file or folder, it returns -1. Note that path is optional. By default it is current folder (″.″). Examples are dir() (Windows) and ls(″../testfile_copy.txt″) (Android).
move_file	move_file(3) : move_file(source, destination, replace_exist) moves file or folder whose path is string source to file or INTO (not to) folder whose path is string destination. If the 3rd parameter,

	replace_exist, is true, then source file (or any file in source folder) will replace destination file (or corresponding file in destination folder) if corresponding file exists. Note that the 3rd parameter is optional. By default it is false. Examples are move_file(″c:\\temp\\try1″, ″D:\\″, true) (Windows) and move_file(″/mnt/sdcard/testfile.txt″, ″./testfile_copy.txt″) (Android).
print_file_list	print_file_list(1) : print_file_list(path) (alias ls(path) or dir(path)) works like ls command in Linux or dir command in Windows. It prints the information for the file or all the files in folder at string based path. It returns the number of entries that printed. If the path does not correspond to an existing file or folder, it returns -1. Note that path is optional. By default it is current folder (″.″). Examples are dir() (Windows) and ls(″../testfile_copy.txt″) (Android).
pwd	pwd(0) : get_working_dir() (with alias pwd()) returns string based current directory.

In the function list, functions cd and change_dir, functions pwd and get_working_dir, and functions ls, dir and print_file_list are three groups of functions. In each group, the functions are exactly the same except the different names. Moreover, the above functions correspond to Dos and Unix commands, as shown below:

1. cd and change_dir: correspond to the cd command in Dos and Unix;

2. pwd and get_working_dir: correspond to the pwd command in Dos and Unix;

3. ls, dir and print_file_list: correspond to the dir command in Dos and the ls command in Unix;

4. copy_file: corresponds to the xcopy command in Dos and cp command in Unix;

5. move_file: corresponds to the move command in Dos and the mv command in Unix;

6. delete_file: corresponds to the del command in Dos and the rm command in Unix;

And there is a list_files function. This function returns all the file names in a folder. Its difference from function dir (or ls or print_file_list) is that list_files returns an array and each element in the array is a string based file name, while dir prints all the files' names, modified time and other properties on the screen. Function dir is very useful when user works in Command Line while list_files should be chosen when programming.

The following example is for the above functions. This example can be found in the examples.mfps file in io and file libs sub-folder in the manual's sample code folder.

```
Help

@language:

 test file operation commands

@end

@language:simplified_chinese

  测试文件整体操作函数

@end

endh

function fileOpr()

 // Assume current working directory is AnMath in Android

 // or the folder where JCmdline.jar is located (for

 // Scientific Calculator for JAVA)

 Variable strOriginalPWD = pwd()

 printf("Current directory is " + strOriginalPWD + "\n")

 

 // now move to scripts/manual/io and file libs/
```

```
cd("scripts/manual/io and file libs/")
printf("After cd, current directory is " + pwd() + "\n")

// now print content in the folder
dir(pwd())

// now create a file in a sub-folder
create_file("test_folder/test_file.txt")

// print content in the folder after create_file
print("After create test_folder/test_file.txt, run dir:\n")
dir(pwd())
print("After create test_folder/test_file.txt, run dir for test_folder:\n")
dir("test_folder")

move_file("test_folder","test_folder1")
// print content in the folder after move_file
print("After move folder test_folder to test_folder1, run dir:\n")
dir(pwd())

if delete_file("test_folder1") == false
 print("Cannot delete a folder with file inside " _
       + "if delete_children_in_folder flag is not set\n")
 if delete_file("test_folder1", true)
  print("Can delete a folder with file inside " _
        + "if delete_children_in_folder flag is set to true\n")
      endif
endif
```

```
 // print content in the folder after delete_file

 print("After delete folder test_folder1, run dir:\n")

 dir(pwd())

 

 // return to the original working directory

 cd(strOriginalPWD)

endf
```

Output of the above example is shown below. Please note that if the working directory is not AnMath (on Android) or the folder where JCmdLine.jar is located (in PC), running the above example user may see a very different result, or even some errors.

```
Current directory is
E:\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime

After cd, current directory is
E:\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scrip
ts\manual\io and file libs

-rwx            examples.mfps    18897    2015-08-14 18:17:29

-rwx            test_read.txt      33    2015-08-13 14:58:55

-rwx              test_rw.bin      21    2015-08-14 13:04:55

-rwx           test_write.txt     135    2015-08-13 15:37:46

After create file test_folder/test_file.txt, run dir:

-rwx            examples.mfps    18897    2015-08-14 18:17:29

drwx             test_folder\       0    2015-08-14 18:17:42

-rwx            test_read.txt      33    2015-08-13 14:58:55

-rwx              test_rw.bin      21    2015-08-14 13:04:55

-rwx           test_write.txt     135    2015-08-13 15:37:46

After create file test_folder/test_file.txt, run dir for test_folder:

-rwx            test_file.txt       0    2015-08-14 18:17:42
```

After move folder test_folder to test_folder1, run dir:

```
-rwx          examples.mfps    18897    2015-08-14 18:17:29
drwx            test_folder1\      0    2015-08-14 18:17:42
-rwx           test_read.txt      33    2015-08-13 14:58:55
-rwx             test_rw.bin      21    2015-08-14 13:04:55
-rwx          test_write.txt     135    2015-08-13 15:37:46
```

Cannot delete a folder with file inside if delete_children_in_folder flag is not set

Can delete a folder with file inside if delete_children_in_folder flag is set to true

After delete folder test_folder1, run dir:

```
-rwx          examples.mfps    18897    2015-08-14 18:17:29
-rwx           test_read.txt      33    2015-08-13 14:58:55
-rwx             test_rw.bin      21    2015-08-14 13:04:55
-rwx          test_write.txt     135    2015-08-13 15:37:46
```

Section 6 An Example of Comprehensive File Operations

This chapter has introduced all of the file operation functions provided by MFP. With them, user is able to do anything on any file if s\he has the authorization. In this section, a comprehensive file operation example is provided to demonstrate the power of MFP.

In Scientific Calculator Plus, function set_array_elem (refer to part 3 in Section 3 of Chapter 3 for details) may assign value to an element in an array and returns the new array. The original array, which is the parameter, may or may not be changed. Even if it is changed, changed value may not be the same as the returned value. In other words, user has to use the returned value as the updated array. This means the right way to call set_array_elem is:

array_to_change = set_array_elem(array_to_change, index, value)

. However, in revision 1.6.6 or earlier, set_array_elem will change the parameter to the new value. In other words, user needs not to retrieve the return value of set_array_elem. Thus many user calls this function in the following way:

set_array_elem(array_to_change, index, value)

, and simply continue to use array_to_change after the function call. However, this will not guarantee correctness after revision 1.7. It is strongly recommended to change code now.

The code change can be performed manually. In this way user has to search all the set_array_elem calls and modify it line by line. This is time-expensive and mistake-prone.

Alternatively user can develop an MFP script to search and change set_array_elem calls. This is much faster and unlikely it will make mistakes.

A question is, will change of source code affect running functions? Fortunately the answer is no because Scientific Calculator Plus automatically loads in all the codes at start. The code will not be reloaded until user restarts Scientific Calculator Plus, or in PC the GUI based Scientific Calculator Plus for JAVA gets the focus back. However, please note that this does not mean Scientific Calculator Plus will never dynamically load code in the future. Thus user still has to be cautious to run script to change code.

To change source code, a script needs to:

1. find all the .mfps files in the scripts folder;

2. search every line of each file and locate the old-style set_array_elem calls;

3. modify some lines in each file and save.

In order to find all the .mfps files in a folder and its sub-folders, functions list_files, is_directory and is_file_normal must be used. List_files lists all the sub-folders and files in the scripts folder, and then script goes through these sub-folders and files. Is_directory is able to tell sub-folders from files. If a file is found, script needs to verify that it is a .mfps file. Then is_file_normal is called to identify if it is a normal file. If a sub-folder is found, script calls list_files again and repeats the above steps.

To achieve the above routine, one solution is to use iteration function, i.e. each time a new sub-folder is found, the iteration function is called. The other way is to use an array to store all the folder and sub-folders. Every time a new sub-folder is found, it is appended to the array until the size of the array no longer increases and all the folders in the array have been searched. The following code chooses the second approach:

```
// all_mfps_files is a 1D array.

// Each element is the path of a source file.

Variable all_mfps_files = []
```

```
// all_folders is a 1D array.
//Each element is a folder or sub-folder.
Variable all_folders = [strScriptsPath]
Variable folder_idx = 0
// go through all_folders. Note that during the procedure
// all_folders is still increasing its size.
While(folder_idx  < size(all_folders)[0])
  // list all the files in a folder
  Variable these_files = list_files(all_folders[folder_idx])
      // go through all the files. Note that these_files is a
      // 1D array so that size(these_files)[0] must be the size
      // of the array. Also note that the array's index starts
      // from 0 to array size – 1.
  For variable idx = 0 to size(these_files)[0] - 1 step 1
    Variable this_file_name = these_files[idx]
    this_file_name = all_folders[folder_idx] _
      + get_file_separator() _
      + this_file_name
    If(is_directory(this_file_name))
      //If this_file_name is a folder, add it to all_folders.
      All_folders = set_array_elem(all_folders, _
                    size(all_folders), _
                    this_file_name)
    Elseif and(stricmp(strsub(this_file_name, _
                strlen(this_file_name)-5), ".mfps")==0, _
```

```
            is_file_normal(this_file_name))
        // if this_file_name is a .mfps source file, add it to
        // all_mfp_files.
        all_mfps_files = set_array_elem(all_mfps_files, _
                    size(all_mfps_files), _
                    this_file_name)
      Endif
    Next
    folder_idx = folder_idx + 1
  Loop
```

After every mfps source file is found, the script starts to read each line of the source file to see if it is an old-style set_array_elem call. Note that old-style set_array_elem call means that, after trimming the blanks from the head and tail, set_array_elem is the beginning of the line, and then round bracket (note that blanks may exist between set_array_elem and bracket), then comma (note that blanks may exist between comma and to-be-changed array name). Because of a clear pattern exists, split function can be used to separate code line into pieces and then identify if the first piece is set_array_elem. The code is shown below:

```
// Assume file has been opened, now read:
Variable strLine = freadline(fd)
// Note that the regex string for "(" is not "(" but "\\(" because
// split function uses "(" as a regex control character so that
// to escape.
Variable strarray1 = split(strline, "\\(")
// If at least 2 sub-strings after stpliting and the first one is
// set_array_elem after trimming the white spaces.
If and(size(strarray1)[0] >= 2, _
      Stricmp(trim(strarray1[0]), "set_array_elem") == 0)
```

```
 // No need to escape comma which is not a regex
 // control character.
 Variable strarray2 = split(strarray1[1], ",")
 If size(strarray2)[0] >= 2
    // Should have at least two sub-strings, otherwise may mean
    // no comma.
  // Now we can confirm that this line needs change.
  ……
 Endif
EndIf
```

Change the old-style set_array_elem call to the new-style is fairly simple. Adding the array name and assignment is enough. The code to do this work is shown below:

```
StrLine = strarray2[0] + " = " + trim(strLine)
```

, where strarray2[0] is the to-be-changed array name (see the above code snip for details).

Then write the line into a new file:

```
Fprintf(fd1, "%s\n", strLine)
```

. After the new file is created, call function move_file to replace old MFP code by the new one. The whole example is shown below. This example can be found in the examples.mfps file in io and file libs sub-folder in the manual’s sample code folder.

```
Help
@language:
 a complicated file operation example
@end
@language:simplified_chinese
  一个复杂的文件操作的例子
```

```
@end
endh
function fileOpr2()
 Variable strConfirm
 // first get current working directory, should be AnMath
 // in Android or the folder where JCmdline.jar is located
 // (for Scientific Calculator for JAVA)
 Variable strOriginalPWD = pwd()
 printf("Current directory is " + strOriginalPWD + "\n")
 // confirm it is the right working folder
 printf("Is this AnMath folder in Android " _
   + "or if you are working on Scientific Calculator for JAVA, " _
   + "is the folder where JCmdline.jar is located?\n")
 strConfirm = input("Y for yes and other for no:", "S")
 if and(stricmp(strConfirm, "Y") != 0, stricmp(strConfirm, "Yes") !=0)
   //exit if not in the right working directory
   print("You are not in the right working directory, exit!\n")
   return
 endif

 // the scripts folder
 Variable strScriptsPath = strOriginalPWD + get_file_separator() _
                         + "scripts"

 // Please back up your source code first
 Print("Please back up your source codes before run the program!!!\n")
 // have you backed up your source codes, if no I cannot continue.
```

```
Print("Have you backed up your source codes?\n")
strConfirm = input("Y for yes and other for no:", "S")
if and(stricmp(strConfirm, "Y") != 0, stricmp(strConfirm, "Yes") !=0)
  // If you haven't backed up your codes, I will do it for you.
  print("If haven't been backed up, I will do it for you!\n")
  pause("Press ENTER to start back up.")
  copy_file(strScriptsPath, strScriptsPath + ".bakup", true)
endif

// ok, now it is the right working directory and source code has been
// backed up. Preparation work has finished, press enter to continue.
pause("Now preparation work has finished, press Enter to continue")

// all_mfps_files is a 1D array with each element path of a .mfps src
Variable all_mfps_files = []
// all_folders is also a 1D array, each element is path of a folder
// including source codes
Variable all_folders = [strScriptsPath]
Variable folder_idx = 0
// Go through all_folders, note that in the procedure all_folders is
// increasing
While(folder_idx  < size(all_folders)[0])
  // list all the files in a folder
  Variable these_files = list_files(all_folders[folder_idx])
  // Go through the files. Note that these_files is a 1D array so
  // size(these_files)[0] must be equal to the length of the array
  // Also note that index is from 0 to array length - 1.
  For variable idx = 0 to size(these_files)[0] - 1 step 1
   Variable this_file_name = these_files[idx]
   this_file_name = all_folders[folder_idx] + get_file_separator() _
    + this_file_name
   If(is_directory(this_file_name))
    // If this file is actually a folder, append it to all_folders
    All_folders = set_array_elem(all_folders, _
              size(all_folders), _
              this_file_name)
   Elseif and(stricmp(strsub(this_file_name, _
            strlen(this_file_name)-5), ".mfps") == 0, _
        is_file_normal(this_file_name))
    // If this file is a .mfps source file, append it to
    // all_mfps_files
    all_mfps_files = set_array_elem(all_mfps_files, _
              size(all_mfps_files), _
              this_file_name)
   Endif
  Next
  folder_idx = folder_idx + 1
 Loop

 // Now all_mfps_files includes all the .mfps files
 For variable idx = 0 to size(all_mfps_files)[0] - 1 step 1
  // create a temporary source file to write the modified code in
  // set encode UTF-8 to ensure that unicode (e.g. Chinese and
  // Japanese characters) is supported
  Variable fd1 = fopen("temporary_src_file","w", "UTF-8")
  print("Now analyse " + all_mfps_files[idx] + "\n")
  Variable fd = fopen(all_mfps_files[idx], "r", "UTF-8")
  Variable idxLine = 0
  while (!feof(fd))
   idxLine = idxLine + 1
   Variable strLine = freadline(fd)
   // Note that the regex string for "(" is "\\(" not "(" because
   // split function uses "(" as a regex control character so that
   // have to escape.
   Variable strarray1 = split(strline, "\\(")
   // If at least 2 sub strings after splitting and the first one
   // is set_array_elem after trimming the white spaces
   If and(size(strarray1)[0] >= 2, _
      Stricmp(trim(strarray1[0]), "set_array_elem") == 0)
    // need not to escape "," for regex, different from "("
    Variable strarray2 = split(strarray1[1], ",")
    If size(strarray2)[0] >= 2
     // Should have at least two sub strings, otherwise may mean
     // no comma.
     // Now we can confirm that this line needs change.
     print("\tset_elem_array calling statement to change at" _
        + " Line " + idxLine + "\n")
     print("\tBefore change is : " + strLine + "\n")
     StrLine = strarray2[0] + " = " + trim(strLine)
     print("\tAfter change is : " + strLine + "\n")
    Endif
   EndIf

   // write strLine into temporary source file

   fprintf(fd1, "%s\n", strLine)

  Loop

  fclose(fd1)

  fclose(fd)

  //move temporary file to replace all_mfps_files[idx]

  move_file("temporary_src_file", all_mfps_files[idx], true)

 Next

 //Done!

 printf("Done!\n")

endf
```

The procedure of the above code is:

1. identify if current working folder is AnMath in Android or the folder where JCmdLine.jar is placed (in PC). If the working folder is wrong, no source code file can be found;

2. remind user to back up. If user hasn't done it, the script will do this for user by copying scripts folder to scripts.bakup;

3. find all the .mfps source files;

4. find all old-style set_array_elem calls;

5. modify and source code files and save the modified files to a new place;

6. copy the modified files to replace the old ones.

The outcome of the program is below. Note that the output varies with user source libs, i.e. different user sees different output. But after the program finishes, user should see all the set_array_elem functions are called in the new way.

```
Current directory is
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_
runtime

Is this AnMath folder in Android or if you are working on Scientific Calculator for
JAVA, is the folder where JCmdline.jar is located?
```

```
Y for yes and other for no:Y

Please back up your source codes before run the program!!!

Have you backed up your source codes?

Y for yes and other for no:n

If haven't been backed up, I will do it for you!

Press ENTER to start back up.

Now preparation work has finished, press Enter to continue

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_
runtime\scripts\examples\math.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_
runtime\scripts\examples\misc.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_
runtime\scripts\userdef_lib\506.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_
runtime\scripts\userdef_lib\biao.mfps

……

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_
runtime\scripts\userdef_lib\cha_ru.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdL
ine_runtime\scripts\userdef_lib\cv100plus测试.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdL
ine_runtime\scripts\userdef_lib\捻系数.mfps
```

```
Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scripts\userdef_lib\支偏计算.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scripts\userdef_lib\断强计算.mfps

Now analyse
C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scripts\userdef_lib\落棉率.mfps

        set_elem_array calling statement to change at Line 35

        Before change is : set_array_elem(luo_mian_lv_s,idx2,luo_mian_lv)

        After change is : luo_mian_lv_s = set_array_elem(luo_mian_lv_s,idx2,luo_mian_lv)

……
```

Summary

Input / output and file operation is an important part of MFP advanced programming. MFP programming language provides a set of C-like I/O functions to read/write console, string and file. Furthermore, MFP programming language includes functions to change directory, list files in a folder and copy, move and delete files. With these functions, as long as user has authorization, any file in OS can be operated.

Please note that file operation in Android sometimes is dangerous. Because files are hidden from users, after program finishes user is not able to easily confirm right files are changed or check result. In this way, it is strongly recommended to create a working folder in AnMath directory and confine all the file operations inside it.

Chapter 7 Time, Date and System Functions

Reading and setting time and date is an important capability in any serious programming language including MFP. Furthermore, MFP provides some APIs to access some operation system functions and commands.

Section 1 Time and Date Functions

Time and date functions provided by MFP programming language are listed as below:

Function Name	Function Info
get_day_of_month	get_day_of_month(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp), get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_day_of_week	get_day_of_week(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp),

	get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_day_of_year	get_day_of_year(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp), get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_hour	get_hour(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp),

	get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_millisecond	get_millisecond(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp), get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_minute	get_minute(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp),

	get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_month	get_month(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp), get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_second	get_second(1) : get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp),

	get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
get_time_stamp	get_time_stamp(1...) : get_time_stamp(string_or_year, ...) returns the timestamp determined by the parameters. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. This function supports two modes. First mode is get_time_stamp(string_time_stamp) where there is only one string based parameter which must be formatted as yyyy-mm-dd hh:mm:ss[.f...]. The fractional second may be omitted. The second mode is get_time_stamp(year, month, day, hour, minute, second, millisecond). All the parameters except year are optional. If omitted, the default value for the parameters hour, minute, second and millisecond is 0, and the default value for the parameters month and day is 1. For example, get_time_stamp("1981-05-30 17:05:06") returns a timestamp at 17:05:06.000 on May 30, 1981. And it works exactly in the same way as get_time_stamp(1981, 5, 30, 17, 5, 6, 0).
get_year	get_year(1) : get_year(timestamp), get_month(timestamp),

	get_day_of_year(timestamp), get_day_of_month(timestamp), get_day_of_week(timestamp), get_hour(timestamp), get_minute(timestamp), get_second(timestamp) and get_millisecond(timestamp) return the year, month, day of year, day of month, day of week, hour, minute, second and millisecond of the timestamp parameter respectively. Timestamp is the difference, measured in milliseconds, between the time it represents and midnight, January 1, 1970 UTC. And day of week is an integer corresponding to Sunday if 0, Monday if 1, ... Saturday if 6. For example, get_day_of_week(get_time_stamp(2014, 12, 21)) returns 0 (Sunday).
now	now(0) : now() returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

As shown above, get_day_of_month, get_day_of_week, get_day_of_year, get_hour, get_millisecond, get_minute, get_month, get_second and get_year convert a timestamp, which is the number of elapsed milliseconds from 12:00 AM on January 1st, 1970 (UTC), to an integer which represents the day of month, day of week (Sunday is 0, Monday is 1, Tuesday is 2, etc), day of year, hour (0 to 23), millisecond, minute, month, second and year respectively. On the other hand, get_time_stamp converts a string based time which can be read by human being to a timestamp. And function now returns the current timestamp.

The following code is an example for the above functions. It can be found in the examples.mfps file in time date and sys libs sub-folder in the manual's sample code folder.

```
Help

@language:

  test time and date functions

@end

@language:simplified_chinese
```

```
  测试日期和时间相关函数
@end
endh
function testTimeDate()
 variable var1
 print("\n\nget_time_stamp(\"1970-01-01 00:00:00.0\") = " _
   + get_time_stamp("1970-01-01 00:00:00.0"))

 // test to convert an invalid time string to a time stamp.
 // Result depends on OS
 try
   print("\n\nget_time_stamp(\"1980-12-71 00:00:00.0\") = ")
   print(get_time_stamp("1980-12-71 00:00:00.0"))
 catch  (var1 = info) == info
   print("throws an exception")
 endtry

 // test now function
 printf("\n\nnow year = %d, month = %d, day of year = %d, " _
   + "day of month = %d, day of week = %d, hour = %d, " _
          + "minute = %d, second = %d, ms = %d", _
          get_year(now()), get_month(now()), get_day_of_year(now()), _
   get_day_of_month(now()), get_day_of_week(now()), _
          get_hour(now()), get_minute(now()), get_second(now()), _
          get_millisecond(now()))

 // test time stamp conversions
 print("\n\nget_millisecond(get_time_stamp(2015, 3, 8, 21, 22, 9, 7)) = " _
   + get_millisecond(get_time_stamp(2015, 3, 8, 21, 22, 9, 7)))
 print("\n\nget_second(get_time_stamp(2015, 3, 8, 21, 22, 19, 700)) = " _
   + get_second(get_time_stamp(2015, 3, 8, 21, 22, 19, 700)))
 print("\n\nget_month(get_time_stamp(2000, 2,29, 16, 58, 9, 700)) = " _
   + get_month(get_time_stamp(2000, 2,29, 16, 58, 9, 700)))
 print("\n\nget_year(get_time_stamp(2014, 12,15, 16, 58, 9, 700)) = " _
   + get_year(get_time_stamp(2014, 12,15, 16, 58, 9, 700)))
 print("\n\nget_day_of_week(get_time_stamp(2014, 12,15, 16, 58, 9, 700)) = " _
   + get_day_of_week(get_time_stamp(2014, 12,15, 16, 58, 9, 700)))
 print("\n\nget_day_of_month(get_time_stamp(2001, 2,29, 16, 58, 9, 700)) = " _
   + get_day_of_month(get_time_stamp(2001, 2,29, 16, 58, 9, 700)))
 print("\n\nget_day_of_year(get_time_stamp(2014, 12,15, 16, 58, 9, 700)) = " _
   + get_day_of_year(get_time_stamp(2014, 12,15, 16, 58, 9, 700)))

 // test conversion of an invalid time. Result depends on OS
 try
   print("\n\nget_hour(get_time_stamp(2014, 12,15, 116, 58, 9, 700)) = " _
              + get_hour(get_time_stamp(2014, 12,15, 116, 58, 9, 700)))
 catch  (var1 = info) == info
   print("\n\nget_hour(get_time_stamp(2014, 12,15, 116, 58, 9, 700)) " _
              + "throws an exception")
 endtry
endf
```

The result of the above example is:

```
get_time_stamp("1970-01-01 00:00:00.0") = -36000000

get_time_stamp("1980-12-71 00:00:00.0") = throws an exception
```

now year = 2015, month = 8, day of year = 228, day of month = 16, day of week = 0, hour = 22, minute = 13, second = 9, ms = 363

get_millisecond(get_time_stamp(2015, 3, 8, 21, 22, 9, 7)) = 7

get_second(get_time_stamp(2015, 3, 8, 21, 22, 19, 700)) = 19

get_month(get_time_stamp(2000, 2,29, 16, 58, 9, 700)) = 2

get_year(get_time_stamp(2014, 12,15, 16, 58, 9, 700)) = 2014

get_day_of_week(get_time_stamp(2014, 12,15, 16, 58, 9, 700)) = 1

get_day_of_month(get_time_stamp(2001, 2,29, 16, 58, 9, 700)) = 1

get_day_of_year(get_time_stamp(2014, 12,15, 16, 58, 9, 700)) = 349

get_hour(get_time_stamp(2014, 12,15, 116, 58, 9, 700)) = 20

Note that if the parameter of get_time_samp is not a valid string based time expression, e.g. "1980-12-71 00:00:00.0" (invalid date), the behavior of get_time_stamp is undefined. In some OSes an error is reported, while in others a timestamp is returned. Therefore it is user's responsibility to keep the parameter right.

Section 2 System Related Functions

MFP provides two system related functions which are sleep and system.

Sleep suspends a running task for a while and then resumes it. The right way to call sleep is sleep(x) where x is a positive value which means the number of milliseconds to sleep. This function returns nothing.

System runs an OS command and returns OS command's returned value. It can be used in two ways. First is simply using the command as its single parameter, which is the only way for Scientific Calculator Plus version 1.6.6 or earlier. Restriction is that the command must be an executable file with its parameters. This means, if using Scientific Calculator Plus for JAVA in windows, calling system("dir") will fail because dir is not an executable file but an internal function of the executable file named cmd.exe. As such, user has to call system("cmd /c dir") instead of system("dir"). Similarly, if running in Linux (not Android), user has to call system("sh -c ls") instead of system("ls").

Though the above way to call system function works on any JAVA platform, it fails in Android. Therefore, from revision 1.6.7, a new way to call system function is introduced. System function still accepts one parameter. However, this parameter must be a string array and each element in the array is a part of the OS command. For example, in order to run "sh -c ls", user may call system(["sh", "-c", "ls"]) in Android. However, to change file1's name to file2, user has to call system(["sh", "-c", "mv file1 file2"]) because "mv file1 file2" is an internal command of sh and it cannot be further separated.

User has to keep in mind that, at this stage, system function cannot accept user's input from command line. Only output can be printed. Also, if the command does not exist, system function throws an exception.

System function works pretty well in Scientific Calculator Plus for JAVA. However, in Android many restrictions and limitations exist because Android does not provide a complete set of sh (shell) functions. Some function calls, e.g. system(["sh", "-c", "echo hello"]), may work well in some mobiles but report error in others. For this reason, if user wants to operate files (e.g. copy, move and delete file or folder) in a mobile, file operating functions listed in Section 5 of Chapter 6 is strongly recommended against calling system(["sh", "-c", "cp file1 file2"]).

Although use of system function is limited in Android, it is still the best approach to start an Android app using MFP programmatically. However, user needs to know the package id of the Android app and its main activity (starting activity)'s name. The usage is:

system("am start –n package_id/main_activity_name")

For example, if user has Smart Photographic Calculator (another app from the same developer of Scientific Calculator Plus) installed, the command to start it is:

system("am start –n
com.cyzapps.SmartMath/com.cyzapps.SmartMath.ActivitySmartCalc")

, where com.cyzapps.SmartMath is the package id of Smart Photographic Calculator and its main activity name is com.cyzapps.SmartMath.ActivitySmartCalc.

Scientific Calculator Plus is able to build APKs. The package id of a generated APK is set by user. However, the main activity name is always com.cyzapps.AnMFPApp.ActivityAnMFPMain (this name may change in a future release of Scientific Calculator Plus). Since package id and main activity name are both known, user is allowed to launch apps created by him or herself.

User is not allowed to terminate a running app by calling system function because this is unsafe.

The following code is an example for the above functions. It can be found in the examples.mfps file in time date and sys libs sub-folder in the manual's sample code folder.

Help

```
@language:
 test sleep and system functions
@end
@language:simplified_chinese
  测试 sleep 和 system 函数
@end
endh
function testSleepSys()
 print("Now sleep 3 seconds\n")
 sleep(3000)
 print("Now wake up!\n")
 // If you have installed Smart Photographic Calculator, this will work
 pause("Now try to start Smart Photographic Calculator. Press Enter to continue")
 system("am start –n com.cyzapps.SmartMath/com.cyzapps.SmartMath.ActivitySmartCalc")
endf
```

Running the above code, user will see an output message printed which is

Now sleep 3 seconds

, and then the program suspends for 3 seconds, then prints another message which is

Now wake up!

. After that, the program tries to start Smart Photographic Calculator. If not successful, an error message is reported.

Summary

Time and date functions are an important component for any programming languages. In OS time is stored as timestamp which is represented as the number of milliseconds elapsed since the mid-night of January 1st, 1970 (UTC). MFP provides a list of API functions to convert between a timestamp and a string based time descriptor readable to human being so that programmers can access and operate time easily.

MFP also includes some system related functions. Function sleep suspends the current running program and function system calls an OS command. Note that in Android directly calling a shell command is not recommended as support to shell may not be completely implemented in some devices. The main use of system function in Android is to start another app if user knows its package id and main activity name.

Chapter 8 Developing Games

Since version 1.7.2, Scientific Calculator Plus has supported 2D game programming using MFP language. Since MFP language is cross platform, an MFP game script can run either in a PC with JAVA support, or in an Android device. Moreover, an MFP game script can be triggered inside Scientific Calculator Plus, or run as an executable file in PC, or be built into a standalone App and installed in every Android device. Because of this feature, MFP developers can program and debug a game in PC and distribute the game to game players using any Android mobiles.

Section 1 Open, Adjust and Close Game Display Window

The first thing in game programming is creating a display window. In an Android device, display window is the whole screen. The size of the display window cannot be adjusted and no title is shown in the display. However, portrait and landscape modes can be switched programmatically. In a PC, a display window's size can be adjusted. Its title is shown. However, there is no portrait or landscape mode. Moreover, whether in an Android device or in a PC, background color and display closing behavior are controlled by program.

The function to create a display window is open_screen_display. It returns a handle of the display window. Here, display window is a concept related to display image (which will be introduced later). This function has six parameters. The first parameter is string based title, which is useless in Android. The second parameter is background color. This is a three or four element array. If it includes four elements, they are [Alpha, R, G, B]; if it includes three elements, they are [R, G, B].In this array, valid value range for every element is from 0 to 255. The third parameter is confirming or not before exit. The fourth parameter is the size of the display window. It is a two array elements. The first one is the width and the second one is the height. Since display window always occupies whole screen, this parameter takes no effect in an Android device. The sixth parameter sets display's position (i.e. portrait or landscape). It is an integer, 0 means landscape, 1 means portrait and -1 means any (depending on the position of the device). This parameter works only in Android. Note that all the six parameters are optional. For example, running the following statement:

variable display = open_screen_display("Hello world", [255, 238, 17], true, [640, 480], true, 0)

would create a display window as below:

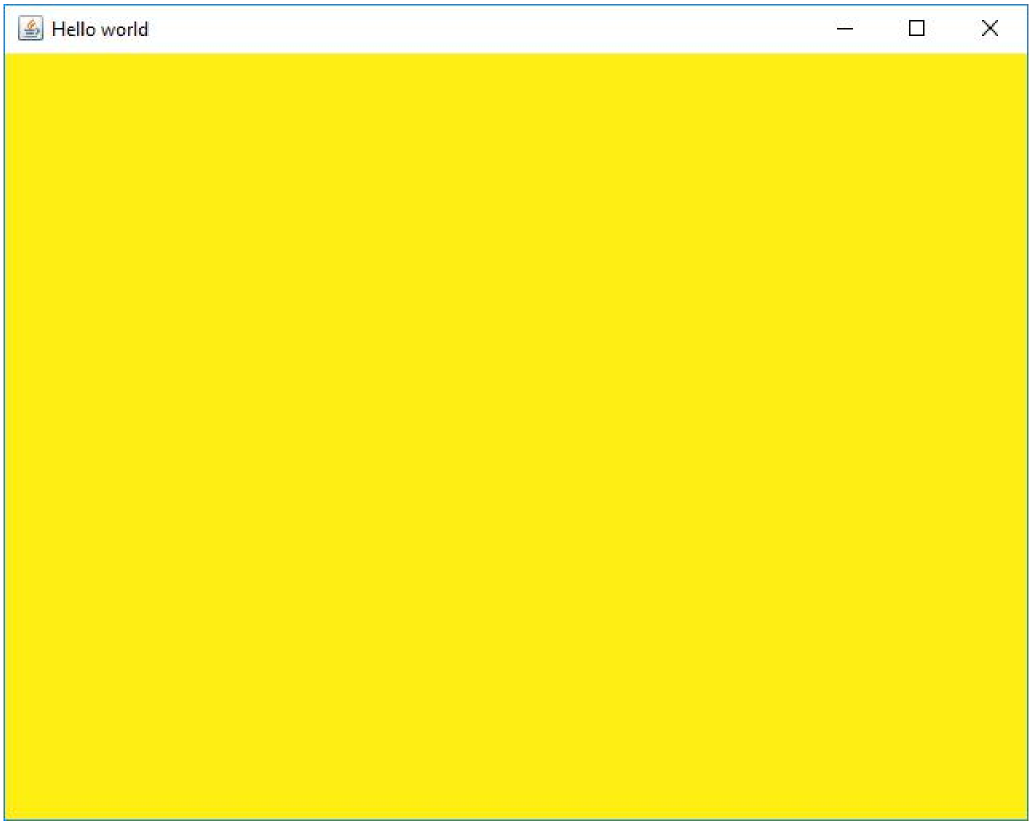


Figure 8.1: Display window shown in PC.

Comparatively, in Android it would simply show a yellow screen.

If user wants to shutdown a display window, shutdown_display function has to be called. This function has two parameters. The first parameter is the handle to the display window. The second parameter is an optional boolean, with true meaning closing without confirmation and false meaning with confirmation if user has already set this flag. And this parameter's default value is false. If user wants to shutdown the above display, simply call shutdown_display(display, true), and the display window is shutdown immediately. The following chart shows how a display window is shutdown with player's confirmation:

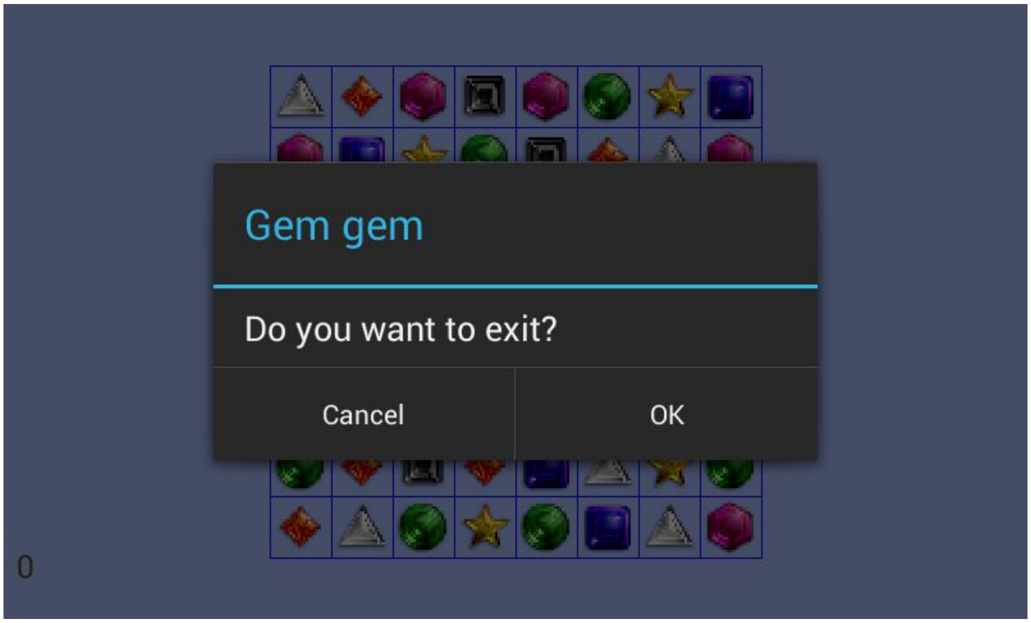


Figure 8.2: Game players need to confirm before exiting a game.

To read and change the above properties of a display window, MFP programming language provides the following get and set functions:

Function Name	Function Info
get_display_caption	::mfp::graph_lib::display::get_display_caption(1) : get_display_caption(display) returns a screen display's caption. If display is an image display or in Android, it always returns an empty string.
set_display_caption	::mfp::graph_lib::display::set_display_caption(2) : set_display_caption(display, caption) sets a screen display's caption on JAVA platform. It does not affect an image display or a screen display in Android.
get_display_bgrnd_color	::mfp::graph_lib::display::get_display_bgrnd_color(1) : get_display_bgrnd_color(display) returns

	background color of a display (whether screen display or image display). Color is a 4-elem array ([Alpha, R, G, B]) or 3-elem array ([R, G, B]) with each elem from 0 to 255.
set_display_bgrnd_color	::mfp::graph_lib::display::set_display_bgrnd_color(2) : set_display_bgrnd_color(display, color) sets background color for a display (whether screen display or image display). Color is a 4-elem array ([Alpha, R, G, B]) or 3-elem array ([R, G, B]) with each elem from 0 to 255.
get_display_confirm_close	::mfp::graph_lib::display::get_display_confirm_close(1) : get_display_confirm_close(display) returns whether confirming is required before closing a screen display. If display is an image display, it always returns false.
set_display_confirm_close	::mfp::graph_lib::display::set_display_confirm_close(2) : set_display_confirm_close(display, confirm_close_or_not) sets whether confirming is required before closing a screen display. It does not affect an image display.
get_display_size	::mfp::graph_lib::display::get_display_size(1) : get_display_size(display) returns a display (whether screen display or image display)'s size. The returned value is a two element array, i.e. [width, height].
set_display_size	::mfp::graph_lib::display::set_display_size(3) : set_display_size(display, width, height) sets a display (whether screen display or image

	display)’s size to be width * height.
get_display_resizable	::mfp::graph_lib::display::get_display_resizable(1) : get_display_resizable(display) tells developer a screen display resizable or not. If display is an image display, it always returns false.
set_display_resizable	::mfp::graph_lib::display::set_display_resizable(2) : set_display_resizable(display, resizable_or_not) sets a screen display resizable or not. It does not affect an image display.
get_display_orientation	::mfp::graph_lib::display::get_display_orientation(1) : get_display_orientation(display) returns orientation of a screen display. In Android, if a screen display is landscape, it returns 0; if a screen display is portrait, it returns 1; if a screen display’s orientation is unspecified, it returns -1. For image display or on a JAVA platform, it always returns -1.
set_display_orientation	::mfp::graph_lib::display::set_display_orientation(2) : set_display_orientation(display, orientation) sets orientation of a screen display in Android. If orientation is -1, it is an unspecified orientation; if it is 0, it is landscape; if it is 1, it is portrait. This function does not affect an image display and has no effect on JAVA platform.

Another important feature of display window is its background image. Function set_display_bgrnd_image(display, image, mode) sets background image and its displaying mode. It has three parameters. The first parameter is the handle of display window; the second parameter is the background image and the third parameter is integer based position mode. Note that background image is different from background color. Background image is on top of background color. There are four position modes for background image. Mode 0 means the background image is placed on the left top of the display window. Mode 1 means the background image is zoomed to exactly fit in the display window. Mode 2 means the background image is tiled on the display window. And mode 3 means the background image is placed in the center of the display window.

To get the background image handle and background image position mode, functions get_display_bgrnd_image and get_display_bgrnd_image_mode should be called respectively. Both of these two functions accept one parameter, which is the handle of the display window.

Section 2 Draw Graph on Display Window

After a display window is started, developer needs to draw and erase graphs on it to show animation. MFP game engine provides a number of functions to draw and erase shapes, plot images and write text based strings.

1. Principle of MFP Game Animation

When developer calls an MFP function to draw something on a display window, MFP does not apply the painting event on the spot. Instead, the painting event is saved in a game painting event scheduler. When the whole or part of the display window needs updating, MFP will first clear the to-be-repainted part of the display window, then draw the background colour and image, and then in order call the painting events in the painting event scheduler. Because each step finishes instantly, game player is only able to identify the difference of display window before and after updating, which seems that the image is moving.

Please note that MFP updates display window at any proper time without noticing developer or game player. If developer intends to update display window, function update_display has to be called. This function has only one parameter which is the handle of display window. As explained above, this function calls every painting event in the painting event scheduler in order. If the scheduler includes many painting events, display updating will take relatively long time. From game player's perspective, long updating time means sluggishness and/or flicker. To avoid this issue, developer may call function drop_old_painting_requests to remove obsolete painting events from painting event scheduler. This function has two parameters. The first parameter is owner_info, telling MFP which painting events should be removed. Each painting event has its own owner_info. MFP compares the owner_info parameter with a painting event's owner info. If the owners are the same and the painting event's creation time is earlier (smaller) than the timestamp defined in drop_old_painting_requests' owner_info parameter, this painting event will be removed. However, sometimes owner_info parameter doesn't include a timestamp. In this case, its timestamp is the time when the owner_info parameter object was created. For example, calling drop_old_painting_requests("my owner", display)

equals to calling

drop_old_painting_requests(["my owner", now()], display)

. In this example, function drop_old_painting_requests will inspect every painting event in the painting event scheduler. If a painting event's owner has name (sometimes a painting event's owner may only have an id but no name), and the name is "my owner", and the painting event was created before function drop_old_painting_requests is called, this event is removed from event scheduler. Otherwise, this event is left in the scheduler.

2. Draw Shapes

MFP provides developers the following functions to draw shapes:

Function Name	Function Info
draw_point	::mfp::graph_lib::draw::draw_point(6) : draw_point(owner_info, display, point_place, color, point_style, painting_extra_info) adds a painting event in the painting event scheduler. This painting event will draw a point in a display when the scheduler calls it. It has 6 parameters. First parameter is owner_info, which tells painting event scheduler who owns this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. The third one is point_place. It is a two element array (i.e. [x, y]). The fourth one is color. It is a 4-elem array ([Alpha, R, G, B]) or 3-elem array

	([R, G, B]) with each elem from 0 to 255. The fifth parameter is point style. At this moment, its format is [point_size, point_shape]. Point size is a positive integer value. Point shape is a string which allows the following values: ″dot″, ″circle″, ″square″, ″diamond″, ″up_triangle″, ″down_triangle″, ″cross″ and ″x″. Note that if point shape is ″dot″, point size has no effect because dot point’s size is always 1. This parameter is optional, by default, it is [1, ″dot″]. The last parameter is painting_extra_info. It tells painting event scheduler what porterduff mode should be selected to draw this point. This parameter is also optional. Because the underlying mechanism of porterduff mode is quite intricate, developer may simply use its default value (i.e. ignore it since it is optional). For detailed information about painting extra info, developer may refer to set_porterduff_mode and get_porterduff_mode functions. For detailed information about porterduff mode, developer may refer to JAVA documentation. An example of draw_point is: draw_point([″my draw″, 0.381], d, [128, 45], [79, 255, 0, 142]) . And another example is: draw_point(NULL, d, [23, 111], [23, 178, 222], [78, ″square″]) .
draw_line	::mfp::graph_lib::draw::draw_line(7) : draw_line(owner_info, display, start_point_place, end_point_place, color, line_style, painting_extra_info) adds a painting event in the painting event scheduler. This painting event will draw a line in a display when the scheduler calls it. It has 7 parameters. First parameter is owner_info, which tells painting event scheduler who owns

	this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. The third one and the fourth one are start_point_place and end_point_place respectively. They both are a two element array (i.e. [x, y]). The fifth one is color. It is a 4-elem array ([Alpha, R, G, B]) or 3-elem array ([R, G, B]) with each elem from 0 to 255. The sixth parameter is line style. At this moment, it is one element array whose element is a positive integer value representing line width. This parameter is optional, by default, it is [1]. The last parameter is painting_extra_info. It tells painting event scheduler what porterduff mode should be selected to draw. This parameter is also optional. Because the underlying mechanism of porterduff mode is quite intricate, developer may simply use its default value (i.e. ignore it since it is optional). For detailed information about painting extra info, developer may refer to set_porterduff_mode and get_porterduff_mode functions. For detailed information about porterduff mode, developer may refer to JAVA documentation. An example of this function is: draw_line(["my draw", 0.381], d, [128, 45], [250, -72], [79, 255, 0, 142]) . And another example is: draw_line(NULL, d, [23, 111], [70, 333], [23,

	178, 222], [7]) .
draw_rect	::mfp::graph_lib::draw::draw_rect(8) : draw_rect(owner_info, display, left_top, width, height, color, frame_or_fill, painting_extra_info) adds a painting event in the painting event scheduler. This painting event will draw a rectangle in a display when the scheduler calls it. It has at least 7 parameters. First parameter is owner_info, which tells painting event scheduler who owns this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. The third parameter is a two element array (i.e. [x, y]). It is the left top corner of the rectangle. The next two parameters are the width and height of the rectangle. Then the parameter is color to draw. It is a 4-elem array ([Alpha, R, G, B]) or 3-elem array ([R, G, B]) with each elem from 0 to 255. The second last parameter is an integer. If it is less than or equal to zero, the rectangle is filled. If it is larger than zero, it is the width of the rectangle's sides. The last parameter is painting_extra_info. It tells painting event scheduler what porterduff mode should be selected to draw. This parameter is optional. Because the underlying mechanism of porterduff mode is quite intricate, developer

	may simply use its default value (i.e. ignore it since it is optional). For detailed information about painting extra info, developer may refer to set_porterduff_mode and get_porterduff_mode functions. For detailed information about porterduff mode, developer may refer to JAVA documentation. An example of this function is: draw_rect(["my draw", 0.381], d, [128, 45], 18, 30, [79, 255, 0, 142], 0) . And another example is: draw_rect(NULL, d, [23, 111], 70, 19, [23, 178, 222], 3) .
draw_oval	::mfp::graph_lib::draw::draw_oval(8) : draw_oval(owner_info, display, left_top, width, height, color, frame_or_fill, painting_extra_info) adds a painting event in the painting event scheduler. This painting event will draw an oval in a display when the scheduler calls it. It has at least 7 parameters. First parameter is owner_info, which tells painting event scheduler who owns this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. The third parameter is a two element array (i.e. [x, y]). It is the left top corner of the rectangle which contains the oval. The next two parameters are the width and

	height of the rectangle which contains the oval. Then the parameter is color to draw. It is a 4-elem array ([Alpha, R, G, B]) or 3-elem array ([R, G, B]) with each elem from 0 to 255. The second last parameter is an integer. If it is less than or equal to zero, the oval is filled. If it is larger than zero, it is the width of the oval's border. The last parameter is painting_extra_info. It tells painting event scheduler what porterduff mode should be selected to draw. This parameter is optional. Because the underlying mechanism of porterduff mode is quite intricate, developer may simply use its default value (i.e. ignore it since it is optional). For detailed information about painting extra info, developer may refer to set_porterduff_mode and get_porterduff_mode functions. For detailed information about porterduff mode, developer may refer to JAVA documentation. An example of this function is: draw_oval([″my draw″, 0.381], d, [128, 45], 18, 30, [79, 255, 0, 142], 0) . And another example is: draw_oval(NULL, d, [23, 111], 70, 19, [23, 178, 222], 3) .
clear_rect	::mfp::graph_lib::draw::clear_rect(5) : clear_rect(owner_info, display, left_top, width, height) adds a painting event in the painting event scheduler. This painting event will clear a rectangle area from a display when the scheduler calls it. It has at 5 parameters. First parameter is owner_info, which tells painting event scheduler who owns this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either

	string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. The third parameter is a two element array (i.e. [x, y]). It is the left top corner of the rectangle. The next two parameters are the width and height of the rectangle. An example of this function is: clear_rect([″my draw″, 0.381], d, [128, 45], 18, 30) . And another example is: clear_rect(NULL, d, [23, 111], 70, 19) .
clear_oval	::mfp::graph_lib::draw::clear_oval(5) : clear_oval(owner_info, display, left_top, width, height) adds a painting event in the painting event scheduler. This painting event will clear an oval area from a display when the scheduler calls it. It has at 5 parameters. First parameter is owner_info, which tells painting event scheduler who owns this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. The third parameter is a two element array (i.e. [x, y]). It is the left top corner

	of the rectangle which contains the oval. The next two parameters are the width and height of the rectangle contains the oval. An example of this function is: clear_oval([″my draw″, 0.381], d, [128, 45], 18, 30) . And another example is: clear_oval(NULL, d, [23, 111], 70, 19) .
draw_polygon	::mfp::graph_lib::draw::draw_polygon(7...) : draw_polygon(owner_info, display, point1_place, point2_place, point3_place, ..., color, frame_or_fill, painting_extra_info) adds a painting event in the painting event scheduler. This painting event will draw a polygon in a display when the scheduler calls it. It has at least 7 parameters. First parameter is owner_info, which tells painting event scheduler who owns this painting event. The owner_info can be a string (i.e. owner name), or an integer (i.e. owner id), or NULL (meaning that system owns it), or a two element array with its first element is either string based owner name, or integer based owner id, or NULL, and its second element is a double value representing a pseudo timestamp (It is not a real timestamp. It can be any double value. It will be used when developer tries to remove this event from painting event scheduler). The second parameter is display. It can be either a screen display or an image display. From the third parameter forward the vertices of the polygon are defined. There should be at least three vertices. All the vertices are a two element array (i.e. [x, y]). The next parameter is color. It is a 4-elem array ([Alpha, R, G, B]) or 3-elem array ([R, G, B]) with each elem from 0 to 255. The second last parameter is an integer. If it is less than or equal to zero, the polygon is filled. If it is larger than

	zero, it is the width of the polygon's sides. The last parameter is painting_extra_info. It tells painting event scheduler what porterduff mode should be selected to draw. This parameter is optional. Because the underlying mechanism of porterduff mode is quite intricate, developer may simply use its default value (i.e. ignore it since it is optional). For detailed information about painting extra info, developer may refer to set_porterduff_mode and get_porterduff_mode functions. For detailed information about porterduff mode, developer may refer to JAVA documentation. An example of this function is: draw_polygon([″my draw″, 0.381], d, [128, 45], [250, -72], [338, 29], [79, 255, 0, 142], 0) . And another example is: draw_polygon(NULL, d, [23, 111], [70, 333], [-239, 89], [66, 183], [23, 178, 222], 3) .

Developer may keep in mind that functions clear_rect and clear_oval simply excavate a rectangle or oval from display window. After these functions are called, the excavated part on the display window is a black color hole.

3. Draw Images

Function draw_image add an image-painting event into painting event scheduler. This function can be called in two different ways. The first one is draw_image(owner_info, display, image_or_path, left, top, width_ratio, height_ratio, painting_extra_info). The second one is draw_image(owner_info, display, image_or_path, srcx1, srcy1, srcx2, srcy2, destx1, desty1, destx2, desty2). In both of the calling approaches, the first parameter is owner_info. Owner_info tells painting event scheduler who owns this painting event. Owner_info can be a string, which means the name of the owner. It can also be an integer, meaning the id of the owner. It can even be NULL, which means system owns the painting event. Owner_info may also be a two element array. The first element of the array is a string (i.e. owner's name) or an integer (i.e. owner's id) or NULL (i.e. system). The second element of the array is a floating value working like a time stamp. However, it is not a real time stamp. It can be any value. It will be used when the scheduler starts to clear painting events. The second parameter of the above calling approaches is display window's handle. The third parameter of the above calling approaches is the handle of a image or a string based image file address. The last parameter, painting_extra_info, tells scheduler what is the porterduff mode to draw the target image. This parameter is optional. The mechanism of porterduff mode is very complicated. So default value (i.e. ignoring this parameter) is strongly recommended. For further details, developer may refer to related JAVA documents.

In the first calling approach, the fourth to the seventh parameters are, respectively, left of the painting destination, top of the painting destination, .scaling ratio along the x-axis (optional, by default it is one), scaling ratio along the y-axis (optional, by default it is one). In the second calling approach, the fourth to the eleventh parameters are, respectively, left of the painting source, top of the painting source, right of the painting source, bottom of the painting source, left of the painting destination, top of the painting destination, right of the painting destination and bottom of the painting destination. Examples of draw_image function are draw_image("image", display, get_upper_level_path(get_src_file_path()) + "gem4.png", 48, 157), draw_image("image", display, gem3Img, 148, 257, 3, 0.5) and draw_image("imagesrc", display, gem3Img, 0, 0, 32, 32, 210, 540, 300, 580, a_painting_extra_info).

4. Draw Texts

Text is a bit different from images or geometry shapes. Origin and alignment must be taken into account. Text origin is unnecessarily the left top corner of the text block. Android platform sees a different origin setting from traditional JAVA platform. Text horizontal alignments include:

1. left alignment, which means left edge of the text block is aligned with a given rectangle's left edge;

2. right alignment, which means right edge of the text block is aligned with a given rectangle's right edge;

3. horizontal center alignment, which means the horizontal center of the text block is aligned with a given rectangle's horizontal center;

The following image illustrates the above horizontal alignments. Each grey rectangle is a line of text. The red shape is a given rectangle. Please note that the text block is not necessarily wider than the given rectangle. Moreover, at this stage, each line inside this text block is always left aligned.

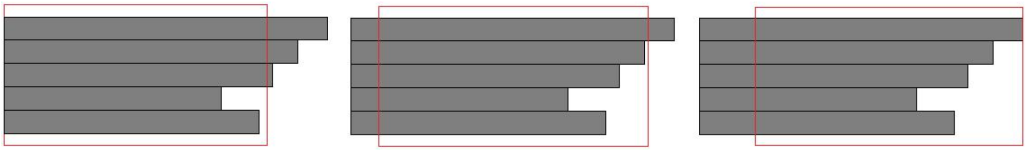


Figure 8.3: Left, horizontal center and right alignments of a text block.

Text vertical alignments are relatively simple compared to horizontal alignment. They include:

1. top alignment, which means top edge of the text block is aligned with a given rectangle's top edge;

2. bottom alignment, which means bottom edge of the text block is aligned with a given rectangle's bottom edge;

3. vertical center alignment, which means the vertical center of the text block is aligned with a given rectangle's vertical center;

MFP programming language provides two functions to calculate the edges of a text block from its origin, and calculate origin from a given border rectangle (Note that a border rectangle is unnecessarily bigger than the text block). The two functions are:

Function Name	Function Info
calculate_text_boundary	::mfp::graph_lib::draw::calculate_text_boundary (4) : calculate_text_boundary(display, string, text_origin, text_style) returns the boundary rectangle of a text block. The format of returned value is a four element array whose elements are [left, top, width, height]. This function's first parameter, display, can be either a screen display or an image display. The second parameter, string, is the multi-line text block. The third parameter is the text block's origin point ([x, y]). An origin point is used by draw_text function as a parameter to draw text. The last parameter, text_style, is a one or two element array. If it is a one element array. The element is a positve integer which is text font size. And the font is system default font. If it is a two element array, the first element is text font size and the second element is string based font name. Note that the last parameter is optional. By default, the system default font with size being 16 is used. Example of this function is: calculate_text_boundary(display, txtStr, [108, 190], [27, "SimSun"]).

calculate_text_origin	::mfp::graph_lib::draw::calculate_text_origin(8) : calculate_text_origin(display, string, boundary_rect_left_top, width, height, horAlign, verAlign, text_style) returns the origin point of a text block given text block's boundary rectangle and alignments. The returned value, origin point, is a two element array (i.e. [x, y]) which will be used in draw_text function. Its first parameter, display, can be either a screen display or an image display. The second parameter, string, is the multi-line text block. The third parameter is the boundary rectangle's left and top. This is a two element array whose first element is left and second is top. The fourth and fifth parameters are width and height of the boundary rectangle respectively. The sixth parameter is the text block's horizontal alignment. -1 means left aligned, 0 means center aligned and 1 means right aligned. The seventh parameter is the text block's vertical alignment. -1 means top aligned, 0 means center aligned and 1 means bottom aligned. The last parameter, text style, is a one or two element array. If it is a one element array. The element is a positve integer which is text font size. And the font is system default font. If it is a two element array, the first element is text font size and the second element is string based font name. Note that the last parameter is optional. By default, the system default font with size being 16 is used. Example of this function is: calculate_text_origin(display, "pei is " + peichoices[idx], [256, 72], peiBndrySize[0], peiBndrySize[1], horAlign, verAlign, [22]) .

Using the above functions, developer may draw a rectangle in any place of display window, and then fill text inside the rectangle, which becomes a button.

The function to draw text is draw_text(owner_info, display, string, origin_place, color, text_style, painting_extra_info). This function adds a painting event in the painting event scheduler. Later on, when this event is called, text string will be drawn in display window. The first parameter of this function is owner_info. Owner_info tells painting event scheduler who owns this painting event. Owner_info can be a string, which means the name of the owner. It can also be an integer, meaning the id of the owner. It can even be NULL, which means system owns the painting event. Owner_info may also be a two element array. The first element of the array is a string (i.e. owner's name) or an integer (i.e. owner's id) or NULL (i.e. system). The second element of the array is a floating value working like a time stamp. However, it is not a real time stamp. It can be any value. It will be used when the scheduler starts to clear painting events. The second parameter is display window's handle. The third parameter is string based text which can be more than one lines. The fourth parameter is text origin. This is a two element array. The first element is x and the second element is y. The fifth parameter is color used in this painting. It is a three or four element array. If four elements, the array is [Alpha, R, G, B]. If three elements, the array is [R, G, B]. In this array, the range of every element is from 0 to 255. The sixth parameter is optional. It defines the font and size of the text. If it is ignored, the font would be system default font and the size of the text is 16. If it is not ignored, it must be an array with one or two elements. If there is only one element, then the element is an integer which means the size of the text. The font of the text, in this case, is system default font. If there are two elements, the first element is the size of the text and the second element is the font's name. Also, developer may keep in mind that this parameter must match the text_style parameter used in calculate_text_origin call before draw_text and the text_style parameter used in calculate_text_boundary call after draw_text. Otherwise, text position calculation cannot be right. The last parameter is painting_extra_info. It tells scheduler what is the porterduff mode to draw the target image. This parameter is optional. The mechanism of porterduff mode is very complicated. So default value (i.e. ignoring this parameter) is strongly recommended. For further details, developer may refer to related JAVA documents.

Examples of draw_text function includes draw_text("image", display, txtStr, [108, 190], [255, 255, 255], [10 + idx, font]) and draw_text("image", display, txtStr, [108, 190], [255, 255, 255], [idx * 2]).

5. Animation Example

An animation example using MFP language is given as below. "Hungry Snake" is a very popular game all over the world. Its logic is quite simple. A snake moves swiftly across the display window. Player can control the moving direction of the snake (i.e. turning up, down, left or right). Also, there is a piece of food in the display window. If the snake eats the food, it will grow up a bit. The snake cannot hit the wall or itself. Otherwise, the game is over.

This section focuses on the animation and drawing player's score. Steering snake movement needs to process player's inputs and will be addressed later.

When implementing the hungry snake game, a diffused approach is to divide snake's moving space into many small square cells. Each cell's width and height exactly match snake body's width. In this way, an elementary movement of the snake is implemented by painting the front cell and end cell of a snake. The CPU cost is very cheap.

In order to practise the above functions, the food will be drawn by calling draw_image function. The body of the snake and the walls are simply made up of squares so that only function draw_rect is needed. The border of snake's moving space is drawn by draw_line. There are also buttons to direct snake's movement. These buttons are implemented by functions draw_rect, calculate_text_origin and draw_text. The score of the player is shown in the left bottom of the screen using function draw_text.

Since MFP hasn't supported global variable, it is strongly recommended to define functions for frequently used constant values, i.e. color, size of snake moving space, size of cell etc. These functions, as shown below, are placed in the beginning of the code so that the constant values are easy to change.

```
// Sleep interval between two updates of screen (ms)
function MOVEINTERVAL()
    if is_running_on_android()
        // if running on android, sleep interval is shorter because
        // MFP takes longer time in calculation than in a PC.
        return 50
    else
        return 500
    endif
endf


// width of the game display window in pixels (for pc only)
function WINDOWDEFAULTWIDTH()
    return 1024
endf
```

```
// height of the game display window in pixels (for pc only)
function WINDOWDEFAULTHEIGHT()
    return 480
endf

// width of button
function BUTTONWIDTH()
    return 80
endf

// height of button
function BUTTONHEIGHT()
    return 60
endf

// font size of button
function BUTTONTEXTFONT()
    return 20
endf

// font size of level information
function LEVELFONTSIZE()
    return 30
endf

// gap between buttons
```

```
function BUTTONGAP()
   return 20
endf

// number of columns in snake's moving space
function GRIDWIDTHDIM()
   return 20
endf

// number of rows in snake's moving space
function GRIDHEIGHTDIM()
   return 16
endf

// size of grid cell in snake's moving space
function CELLSIZE(windowWidth, windowHeight, gridWidthDim, gridHeightDim)
   return round(min(windowWidth/1.2/gridWidthDim, windowHeight/1.2/gridHeightDim))
endf

// the width from the left side of display window to
// the left side of snake's moving space.
function XMARGIN(windowWidth, windowHeight, gridWidthDim, gridHeightDim)
   variable widthEdge = (windowWidth - CELLSIZE(windowWidth, windowHeight, gridWidthDim, gridHeightDim) * gridWidthDim)
   variable heightEdge = (windowHeight - CELLSIZE(windowWidth, windowHeight, gridWidthDim, gridHeightDim) * gridHeightDim)
   if widthEdge > heightEdge
      return round(widthEdge / 3)
```

```
    else
        return round(widthEdge / 2)
    endif
endf

// the height from the top side of display window to
// the top side of snake's moving space.
function YMARGIN(windowWidth, windowHeight, gridWidthDim, gridHeightDim)
    variable widthEdge = (windowWidth - CELLSIZE(windowWidth, windowHeight, gridWidthDim, gridHeightDim) * gridWidthDim)
    variable heightEdge = (windowHeight - CELLSIZE(windowWidth, windowHeight, gridWidthDim, gridHeightDim) * gridHeightDim)
    if widthEdge > heightEdge
        return round(heightEdge / 2)
    else
        return round(heightEdge / 3)
endf

// background color
function BGCOLOR()
    return [170, 190, 255]
endf

// color of the board of snake's moving space
function BOARDERCOLOR()
    return [125, 255, 100, 100]
endf
```

```
// color of the wall

function WALLCOLOR()

    return [125, 100, 100, 255]

endf



// color of the snake's body

function SNAKECOLOR()

    return [155, 100, 255, 100]

endf



// color of the score

function SCORECOLOR()

    return [125, 90, 70, 0]

endf



// color of the text

function TEXTCOLOR()

    return [225, 20, 20, 20]

endf



// color of front edge (facing light edge) in the buttons

function BUTTONFRONTCOLOR()

    return [255, 255, 255]

endf



// color of back edge (not facing light edge) in the buttons

function BUTTONBACKCOLOR()
```

```
    return [0, 0, 0]
endf


// color of game over text
function GAMEOVERCOLOR()
    return [225, 230, 230, 230]
endf


// scaling ratio
function SCALINGRATIO()
    return 0.5
endf
```

However, to save stacking time, a constant value should be saved in a variable instead of calling its function multiple times.

```
    // Initial set up color values
    variable snakeColor = SNAKECOLOR(), scoreColor = SCORECOLOR(), wallColor = WALLCOLOR(), boarderColor = BOARDERCOLOR()
    variable btnFrontColor = BUTTONFRONTCOLOR(), btnBackColor = BUTTONBACKCOLOR()

    // first of all, we store window size, grid dim, x, y margins, button size and gap, and scaling ratio 1/scaling ratio
    // in variables. This avoids calling functions repeatedly so that saves computing time.
    variable windowWidth = get_display_size(DISPLAYSURF)[0], windowHeight = get_display_size(DISPLAYSURF)[1] // window size
    variable gridWidthDim = GRIDWIDTHDIM(), gridHeightDim = GRIDHEIGHTDIM()  // dim of grid (i.e. snake's moving space)
    variable cellSize = CELLSIZE(windowWidth, windowHeight, gridWidthDim, gridHeightDim)
    variable xMargin = XMARGIN(windowWidth, windowHeight, gridWidthDim, gridHeightDim) // left margin
    variable yMargin = YMARGIN(windowWidth, windowHeight, gridWidthDim, gridHeightDim) // top margin

    variable btnW = BUTTONWIDTH(), btnH = BUTTONHEIGHT() // width and height of buttons

    variable btnGap = BUTTONGAP() // gap of button

    variable scalingRatio = SCALINGRATIO() // scaling ratio

    variable oneOverScalingRatio = 1/scalingRatio // 1/scaling ratio

    variable scaledCellSize = cellSize * scalingRatio // scaled cell size
```

Now it is time to calculate the place of the wall(s), snake's starting place, food's starting place and places of the control buttons. Wall(s), snake and food are simple:

```
    // initial place of food

    variable foodPlace = calcFoodInitPlace(gridWidthDim, gridHeightDim, level)

    // initial place of wall

    variable wallPlace = calcWallPlace(gridWidthDim, gridHeightDim, level)

    // initial place of snake

    variable snakePlace = [[3, 5], [3, 4], [3, 3]]

    // moving direction: 1 right, -1 left, i up, -i down

    variable moveDirection = -i

    // calculate the grid cells that snake can move, i.e. the moving space of snake excluding wall and snake body

    variable excludeBarrierPlace = calcExcludeWallPlace(wallPlace, gridWidthDim, gridHeightDim)

    for variable idx = 0 to size(snakePlace)[0] - 1

        excludeBarrierPlace[snakePlace[idx][0], snakePlace[idx][1]] = 2

    next
```

However, the places of the buttons depend on the position (portrait or landscape) of the screen in Android device. If screen is portrait, the buttons are placed in the bottom. If screen is landscape, or the game is running in a PC, the buttons are placed in the right side. If screen is a square (like blackberry) or close to a square, no space for the buttons so that they are not shown. In this case, player can still steer snake's moving using finger to sweep. The following code calculates the places of the buttons:

```
    // right and bottom of snake's moving space(in pixels)

    variable gridRight = xMargin + gridWidthDim * cellSize, gridBottom = yMargin + gridHeightDim * cellSize

    // up, down, left, right buttons' left top positions

    variable upBtnLT = [-1, -1], downBtnLT = [-1, -1], leftBtnLT = [-1, -1], rightBtnLT = [-1, -1]

    variable shouldDrawButtons = true // should we draw the button?

    // are the up, down, left, right buttons pushed?

    variable upBtnPushed = false, downBtnPushed = false, leftBtnPushed = false, rightBtnPushed = false

    if (windowWidth - gridRight > btnW * 3)    // buttons are on right hand side

        upBtnLT = [(windowWidth + gridRight)/2 - 0.5 * (btnW + btnGap), windowHeight / 2 - 1.5 * btnH - btnGap]

        downBtnLT = [(windowWidth + gridRight)/2 - 0.5 * (btnW + btnGap), windowHeight / 2 + 0.5 * btnH + btnGap]

        leftBtnLT = [(windowWidth*0.25+ gridRight*0.75) - 0.5 * (btnW + btnGap), windowHeight / 2 - 0.5 * btnH]

        rightBtnLT = [(windowWidth*0.75+ gridRight*0.25) - 0.5 * (btnW + btnGap), windowHeight / 2 - 0.5 * btnH]

    elseif (windowHeight - gridBottom > btnH * 3)    // bottons are in bottom

        upBtnLT = [windowWidth /2 - 0.5 * (btnW + btnGap), (windowHeight + gridBottom)/2 - 1.5 * btnH - btnGap]

        downBtnLT = [windowWidth /2 - 0.5 * (btnW + btnGap), (windowHeight + gridBottom)/2 + 0.5 * btnH + btnGap]

        leftBtnLT = [windowWidth /4 - 0.5 * (btnW + btnGap), (windowHeight + gridBottom)/2 - 0.5 * btnH]

        rightBtnLT = [windowWidth * 0.75 - 0.5 * (btnW + btnGap), (windowHeight + gridBottom)/2 - 0.5 * btnH]

    else

        shouldDrawButtons = false // buttons are not needed

    endif
```

After the places are determined, it is time to draw the objects (i.e. wall(s), snake and buttons). Since wall(s) and snake are made up of individual cells, simply call draw_rect multiple times. They therefore share the same function called drawPoints:

```
// draw snake or wall, points means the cells' grid coordinates
function drawPoints(drawBGOwner, display, points, color, cellSize, xMargin, yMargin, scalingRatio)
    for variable idx = 0 to size(points)[0] - 1 step 1
        // draw rectangle cell by cell
        draw_rect(drawBGOwner, display, calcTopLeft(points[idx], cellSize, xMargin, yMargin) * scalingRatio, cellSize * scalingRatio, cellSize * scalingRatio, color, 0)
    next
endf
```

In order to draw buttons, the text must be in the middle of the button. To this end, function calculate_text_origin has to be used to determine the starting point of a button. Then function draw_text writes the text. Also, a button has two states, i.e. pressed and released. When a button is pressed down, left and top borders are back to light so that they are dark and right bottom borders are facing light so that bright. When a button is released, left and top borders are bright while right bottom borders are dark. Therefore, function draw_rect cannot be used. Developer has to call draw_line four times to draw the four borders, as shown in the following code:

```
// draw button text which must be horizontally and vertically center aligned with the rectangular button border
function drawButtonText(display, topLeft, width, height, text, isPushed, scalingRatio)
    variable btnTxtFnt = BUTTONTEXTFONT()
    variable textOrigin = calculate_text_origin(display, text, topLeft, width, height, 0, 0, btnTxtFnt)
    draw_text("static element", display, text, textOrigin * scalingRatio, TEXTCOLOR(), btnTxtFnt * scalingRatio)
endf


// draw button's border on screen display. There are two states, pushed or unpushed. If the button is not pushed, left
// and top edges have front light color while right and bottom edges have back light color. Otherwise, left and top have
// back light color while right and bottom have front light color.
```

```
function drawButtonBorderOnScreen(info, display, topLeft, width, height, isPushed, btnFrontColor, btnBackColor)

    variable color1 = btnFrontColor, color2 = btnBackColor

    if isPushed // is button pushed?

        color1 = btnBackColor // back light color

        color2 = btnFrontColor // front light color

    endif

    draw_line(info, display, topLeft, [topLeft[0], topLeft[1] + height], color1, 2) // left border

    draw_line(info, display, topLeft, [topLeft[0] + width, topLeft[1]], color1, 2) // top border

    draw_line(info, display, [topLeft[0], topLeft[1] + height], [topLeft[0] + width, topLeft[1] + height], color2, 2)  // bottom border

    draw_line(info, display, [topLeft[0] + width, topLeft[1]], [topLeft[0] + width, topLeft[1] + height], color2, 2)  // right border

endf
```

Then developer can start to draw the static objects. They are wall(s), border of snake's moving space, text on the buttons and snake's body, food and buttons' borders before game starts. Note that food is drawn by function draw_image. This function loads a picture from hard drive or SD card and pastes it to the screen. In order to load a picture to memory and measure its size, functions load_image and get_image_size are called. These two functions will be addressed in details later on.

```
    // open game display window.

    variable DISPLAYSURF = open_screen_display("Hungry snake", BGCOLOR(), true, [windowWidth, windowHeight], false)

    // draw border for snake's moving space (i.e. grid)

    draw_rect("static element", DISPLAYSURF, [xMargin, yMargin], gridWidthDim * cellSize, gridHeightDim * cellSize, boarderColor, 1)

    // draw the wall

    drawPoints("static element", DISPLAYSURF, wallPlace, wallColor, cellSize, xMargin, yMargin, 1)

    if(shouldDrawButtons)

        // draw text and border of buttons if needed

        // draw up button
        drawButtonText(DISPLAYSURF, upBtnLT, btnW, btnH, "Up", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, upBtnLT, btnW, btnH, upBtnPushed, btnFrontColor, btnBackColor)

        // draw down button

        drawButtonText(DISPLAYSURF, downBtnLT, btnW, btnH, "Down", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, downBtnLT, btnW, btnH, downBtnPushed, btnFrontColor, btnBackColor)

        // draw left button

        drawButtonText(DISPLAYSURF, leftBtnLT, btnW, btnH, "Left", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, leftBtnLT, btnW, btnH, leftBtnPushed, btnFrontColor, btnBackColor)

        // draw right button

        drawButtonText(DISPLAYSURF, rightBtnLT, btnW, btnH, "Right", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, rightBtnLT, btnW, btnH, rightBtnPushed, btnFrontColor, btnBackColor)

    endif

    // draw snake body cell by cell

    variable totalSnakeLen = size(snakePlace)[0]

    for variable idx = 0 to size(snakePlace)[0] - 1

        draw_rect(["snake", size(snakePlace)[0] - 1 - idx], DISPLAYSURF, calcTopLeft(snakePlace[idx], cellSize, xMargin, yMargin), cellSize, cellSize, snakeColor, 0)

    next

    // calculate food left top coordinate (in pixels)

    variable foodPlaceXY = calcTopLeft(foodPlace, cellSize, xMargin, yMargin)

    // load food image

    variable foodImage = load_image(get_upper_level_path(get_src_file_path()) + "food.png")

    // calculate image size

    variable foodImageSize = get_image_size(foodImage)

    // draw food
```

```
draw_image("food", DISPLAYSURF, foodImage, foodPlaceXY[0], foodPlaceXY[1], cellSize/foodImageSize[0], cellSize/foodImageSize[1])
```

Until now, the images are still static. In order to move the snake, first of all developer removes painting events from painting event scheduler which call function draw_rect to draw snake's body. Then developer updates the position of the snake and redraw snake's body, i.e. call draw_rect again to create new painting events. Then developer needs to update screen by calling function update_screen. Moreover, the program has to sleep a while (e.g. 20ms) before next updating. Without sleep, the screen will blink. The detailed code is listed below:

```
variable deltaX = 0, deltaY = 0 // snake head movement

while true

    update_display(DISPLAYSURF) // update game display window

    select (moveDirection) // determine snake head movement from move direction

    case 1

        deltaX = 1

        deltaY = 0

        break

    case -1

        deltaX = -1

        deltaY = 0

        break

    case i

        deltaX = 0

        deltaY = -1

        break

    case -i

        deltaX = 0

        deltaY = 1

        break

    default
```

```
      // do nothing
    ends
    // calculate the new snake head place (in grid coordinate)
    variable newX = mod(snakePlace[0][0] + deltaX, gridWidthDim)
    variable newY = mod(snakePlace[0][1] + deltaY, gridHeightDim)
    variable head2Add = [newX, newY]
    // because snake head has occpied the cell, food or snake body cannot take it again
    excludeBarrierPlace[newX, newY] = 2
    // insert the new snake head place into snakePlace list
    snakePlace = insert_elem_into_ablist(snakePlace, 0, head2Add)
    // remove tail of the snake
    variable tailIdx = size(snakePlace)[0] - 1
    variable tail2Remove = snakePlace[tailIdx]
    // now the cell is available for food and snake's body
    excludeBarrierPlace[tail2Remove[0], tail2Remove[1]] = 0
    // remove the old snake tail place into snakePlace list
    snakePlace = remove_elem_from_ablist(snakePlace, tailIdx)
    sleep(MOVEINTERVAL()) // sleep a while
    // remove all the old painting event requests for snake from the painting request scheduler
    drop_old_painting_requests("snake", DISPLAYSURF)
    // redraw snake body cell by cell
    variable totalSnakeLen = size(snakePlace)[0]
    for variable idx = 0 to size(snakePlace)[0] - 1
      draw_rect(["snake", size(snakePlace)[0] - 1 - idx], DISPLAYSURF, calcTopLeft(snakePlace[idx], cellSize, xMargin, yMargin), cellSize, cellSize, snakeColor, 0)
    next
  loop
```

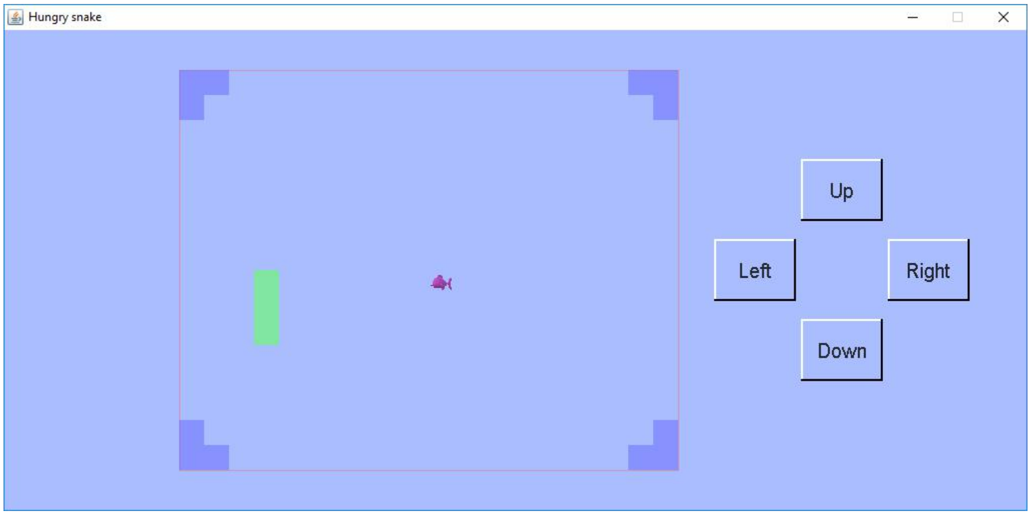


Figure 8.4: Make snake move on the display screen.

Now run the above code, player will see a green "snake"moving swiftly from top to bottom, and then reappear from top. A piece of pink food, which is a fish, is placed in the middle of the screen. Navy blue walls are at the four corners of the snake's moving space. The control buttons are placed on the right of the display window.

Section 3 Process Images and Sounds

In the previous section animation principle has been introduced. However, without optimization, repainting every object on display window in each animation step is very expensive and makes a game sluggish and irresponsive. Provided that accessing RAM is much faster than reading/writing physical screen (i.e. video memory), a better solution would be drawing all objects in an image, and then pasting the image to display window. In this way, image processing replaces video memory reading/writing so that animation is much more swift and responsive.

Moreover, sound effects are always required when playing a game. This section will also address sound processing.

1. Create, Load, Clone and Save Image

The following functions create, load, clone and save image, return image size and validate image handle respectively.

Function Name	Function Info
create_image	::mfp::multimedia::image_lib::create_image(2) : create_image(w, h) returns a new and blank wrapped JAVA image object with width = w and

	height = h.
load_image	::mfp::multimedia::image_lib::load_image(1) : load_image(image_path) returns a wrapped JAVA image object. image_path is a string based path pointing to an image file.
load_image_from_zip	::mfp::multimedia::image_lib::load_image_from_zip(3) : load_image_from_zip(zip_file_name, zip_entry_path, zip_file_type) returns a wrapped JAVA image object loaded from a zipped file. Its first parameter is the path of the zipped file. Its second parameter is the zip entry path of the image. Its last parameter is zip file type. 0 means it is a normal zip file and 1 means it is an Android asset zip file (for MFP app).
clone_image	::mfp::multimedia::image_lib::clone_image(7) : clone_image(image_src, src_left, src_top, src_right, src_bottom, dest_width, dest_height) returns a new wrapped JAVA image object with width = dest_width and height = dest_height. The returned image is a (zoomed) copy of image_src's selected area. The selected area's left, top, right and bottom are src_left, src_top, src_right and src_bottom respectively. Note that src_left, src_top, src_right and src_bottom are optional. By default, they equal 0, 0, width of image_src and height of image_src respectively. dest_width and dest_height are also optional. By default, dest_width equals src_right - src_left and dest_height equals src_bottom - src_top. An example of this function is

	clone_image(img_src, 0, 0, 100, 200, 50, 300) .
save_image	::mfp::multimedia::image_lib::save_image(3) : save_image(image, file_format, path) saves a wrapped JAVA image object to an image file. The first parameter is the wrapped JAVA image object. The second parameter is the format of the file. It is a string and currently only ″png″, ″jpg″ and ″bmp″ are supported. The third parameter is the path of the file. This function returns true if the file is saved successfully, and returns false if any failure. For example, save_image(img, ″png″, ″C:\\Temp\\1.png″) .
get_image_size	::mfp::multimedia::image_lib::get_image_size(1) : get_image_size(image_handle) returns size (i.e. [width, height]) of a wrapped JAVA image object (i.e. image handle).
is_valid_image_handle	::mfp::multimedia::image_lib::is_valid_image_handle(1) : is_valid_image_handle(image_handle) returns true or false, telling developer if a image handle (i.e. a wrapped JAVA image object) is still valid or has been closed.

2. Modify Image

Before pasting an image onto display window, developer generally needs to modify it, i.e. painting on top of the image. In MFP, APIs to paint on image are either the same as or very similar to APIs for display window. Developer calls function open_image_display to create an “image display”. This display’s original size is the same as the image and its size can also be adjusted by set_display_size function. The image itself is the background image of the “image display”. Exactly same as display window, developer is able call functions like draw_rect and draw_image to paint on the “image display”, and call functions update_display and drop_old_painting_requests to control painting time and manage painting events respectively. After drawing finishes, function get_display_snapshot has to be called to obtain the snapshot of the “image display”, i.e. the image after modification. This snapshot can be used by function draw_image to be "pasted" onto a display window. When the image's updated snapshot is no longer used, developer can call shutdown_display to close the "image display".

Developer may keep in mind that, first of all, painting on an "image display" will not change original image. In other words, after updating an"image display" and then closing it, its original image is still unchanged. Second, similar to physical display window, if an "image display"'s painting event scheduler includes too many painting events, updating it will be very slow. A solution is to call function set_display_snapshot_as_bgrnd. This function uses snapshot of a display, whether a display window or an "image display", as the display's background image. Therefore, all the painting events to draw the new background image can be removed after the new background is created.

The details of the above functions are listed as below:

Function Name	Function Info
open_image_display	::mfp::multimedia::image_lib::open_image_display(1) : open_image_display(image_path_or_handle) creates an image display for developer to paint. image_path_or_handle is either a string based path pointing to an image file, or null, or a memory handle of a JAVA image object returned by load_image, load_image_from_zip, create_image or clone_image functions.
get_display_snapshot	::mfp::graph_lib::display::get_display_snapshot(4) : get_display_snapshot(display, update_screen_or_not, width_ratio, height_ratio) returns a display (whether screen display or image display)'s snapshot. Its second parameter, update_screen_or_not, telling MFP whether or not the display should be refreshed so that latest image can be captured. The third and fourth parameters are optional. They tell MFP how to zoom the snapshot. By default, both of them are 1. For example, get_display_snapshot(d, true, 0.5, 3) refreshes display d (i.e. all the painting event

	callbacks take effect) and then takes snapshot of the display and returns a zoomed image. The width of returned image is 0.5 * original snapshot width and the height of returned image is 3 * original snapshot height.
set_display_snapshot_as_bgrnd	::mfp::graph_lib::display::set_display_snapshot_as_bgrnd(3) : set_display_snapshot_as_bgrnd(display, update_screen_or_not, clear_callbacks_or_not) sets a display (whether screen display or image display)'s snapshot as background image. Its second parameter, update_screen_or_not, telling MFP whether or not the display should be refreshed so that latest image can be captured. The third parameter, clear_callbacks_or_not, tells MFP whether or not the painting event callbacks should be cleared. For example, set_display_snapshot_as_bgrnd(d, true, true) refreshes display d (i.e. all the painting event callbacks take effect) and then drops all the painting event callbacks, and then takes snapshot of the display and sets the snapshot to be display's background image.

The following code is copied from the Hungry Snake game. It uses "image display" to draw static objects. And then the snapshot of the "image display" is used as display window's background image. This approach avoids overhead paintings in each animation step.

```
  // open an empty image display

  variable boardImageDisplay = open_image_display(null)

  // adjust it's size to game display window's size times scaling ratio

  set_display_size(boardImageDisplay, windowWidth * scalingRatio, windowHeight * scalingRatio)

  // calculate text origin of level information

  variable textOrigin = [10, 10]

  variable levelFontSize = LEVELFONTSIZE()

  if xMargin > yMargin
```

```
        // text is in the center of left edge rectangle
        textOrigin = calculate_text_origin(DISPLAYSURF, "level " + level, [0, 0], xMargin, windowHeight, 0, 0, levelFontSize)
    else
        // text is in the center of top edge rectangle
        textOrigin = calculate_text_origin(DISPLAYSURF, "level " + level, [0, 0], windowWidth, yMargin, 0, 0, levelFontSize)
    endif
    // draw level information text. Note that the text is scaled down to fit the image.
    draw_text("static element", boardImageDisplay, "level " + level, textOrigin * scalingRatio, scoreColor, levelFontSize * scalingRatio)
    // draw the border of snake's moving space. Note that the rectangle is scaled down to fit the image.
    draw_rect("static element", boardImageDisplay, [xMargin, yMargin] * scalingRatio, gridWidthDim * scaledCellSize, gridHeightDim * scaledCellSize, boarderColor, 1)
    // draw the wall. Note that the wall is scaled down to fit the image.
    drawPoints("static element", boardImageDisplay, wallPlace, wallColor, cellSize, xMargin, yMargin, scalingRatio)
    if(shouldDrawButtons)
        // we draw text only for each button because button text is static while button border is not.
        drawButtonText(boardImageDisplay, upBtnLT, btnW, btnH, "Up", false, scalingRatio)
        drawButtonText(boardImageDisplay, downBtnLT, btnW, btnH, "Down", false, scalingRatio)
        drawButtonText(boardImageDisplay, leftBtnLT, btnW, btnH, "Left", false, scalingRatio)
        drawButtonText(boardImageDisplay, rightBtnLT, btnW, btnH, "Right", false, scalingRatio)
    endif
    // get snapshot of the image display, note that we update the image display before taking snapshot
    variable boardImage = get_display_snapshot(boardImageDisplay, true)
    // shutdown image display
    shutdown_display(boardImageDisplay)
    // set the snapshot of the image display to be game's display window's background image.
    // note that the mode is stretching the background image to fit the whole game's display window
    // as the snapshot image is smaller than the game's display window.
    // 游戏真实显示窗口大小一致。
    set_display_bgrnd_image(DISPLAYSURF, boardImage, 1)
```

3. Play Sound

MFP has the following sound processing functions:

Function Name	Function Info
play_sound	::mfp::multimedia::audio_lib::play_sound(4) : play_sound(source_path, repeat_or_not, volume, create_new_or_not) plays a sound file (can be wave, midi or mp3). This function returns a sound handle which is a JAVA (or Android) media player wrapper. Because the media player resource is scarce, this function will try to reuse previously created sound handle. It has four parameters. The first parameter is the path of the sound file. The second parameter is a boolean telling MFP whether the sound should be repeated to play or not. This parameter is optional and its default value is false. The third one, which is a double from 0 to 1, is the volume of the sound. This parameter is optional and its default value is 1. The fourth one is a boolean flag telling MFP to create a new sound handle compulsorily or not. If it is true, play_sound always creates a new sound handle. This function is optional and its default value is false.
play_sound_from_zip	::mfp::multimedia::audio_lib::play_sound_from_zip(6) : play_sound_from_zip(source_zip_file_path, zip_entry_path, zip_file_type, repeat_or_not, volume, create_new_or_not) plays a sound file (can be wave, midi or mp3) extracted from a zip

	file. This function returns a sound handle which is a JAVA (or Android) media player wrapper. Because the media player resource is scarce, this function will try to reuse previously created sound handle. It has six parameters. The first parameter is the path of the zip file. The second parameter is zip entry path of the zipped sound file. The third parameter is zip file type, 0 for normal zip file and 1 for MFP app's Android asset zip file. The fourth parameter is a boolean telling MFP whether the sound should be repeated to play or not. This parameter is optional and its default value is false. The fifth one, which is a double from 0 to 1, is the volume of the sound. This parameter is optional and its default value is 1. The sixth one is a boolean flag telling MFP to create a new sound handle compulsorily or not. If it is true, play_sound always creates a new sound handle. This function is optional and its default value is false.
start_sound	::mfp::multimedia::audio_lib::start_sound(1) : start_sound(sound_handle) plays a sound referred by sound handle sound_handle. If the sound has been started, this function will do nothing.
stop_all_sounds	::mfp::multimedia::audio_lib::stop_all_sounds(0) : stop_all_sounds() stops all playing sounds.
stop_sound	::mfp::multimedia::audio_lib::stop_sound(1) : stop_sound(sound_handle) stops the playing sound referred by sound_handle. If the sound is not playing, it does nothing.

get_sound_path	::mfp::multimedia::audio_lib::get_sound_path(1) : get_sound_path(sound_handle) returns sound file path of the sound handle.
get_sound_reference_path	::mfp::multimedia::audio_lib::get_sound_reference_path(1) : get_sound_reference_path(sound_handle) returns sound reference file path of the sound handle. If the sound is not extracted from a zip file, sound reference file is the same as sound file. If the sound is extracted from a zip file, sound reference file path is the zip file path followed by the sound’s zip entry path, like ″/folder1/folder2/snd.zip/zipped_folder/snd.wav″, where ″/folder1/folder2/snd.zip″ is path of the zip file and ″zipped_folder/snd.wav″ is zip entry path.
get_sound_repeat	::mfp::multimedia::audio_lib::get_sound_repeat(1) : get_sound_repeat(sound_handle) returns a boolean indicating the sound is repeatedly played or not.
get_sound_source_type	::mfp::multimedia::audio_lib::get_sound_source_type(1) : get_sound_source_type(sound_handle) returns an integer which is sound reference source file type. 0 means normal source file, 1 means zipped source file and 2 means zipped source file in Android asset (for MFP app).
get_sound_volume	::mfp::multimedia::audio_lib::get_sound_volume(1) : get_sound_volume(sound_handle) returns volume of the sound referred by sound_handle. Volume

	is a double value ranging from 0 to 1.
set_sound_repeat	::mfp::multimedia::audio_lib::set_sound_repeat(2) : set_sound_repeat(sound_handle, repeat_or_not) set a sound handle to play repeatedly or not.
set_sound_volume	::mfp::multimedia::audio_lib::set_sound_volume(2) : set_sound_volume(sound_handle, volume) set volume (from 0 to 1) a sound handle.

Because of limited resource, MFP tries to reuse existing media players. MFP compares reference path of a to-be-played sound file with reference path of each existing media player. If there is a match, MFP reuses the existing media player. Otherwise, MFP creates a new media player.

In most cases, reference path of a sound file is the same as its actual path. However, if a sound file is read from a zip file (including a zip file in MFP apps' asset), this sound file will be copied to a temporary place in SD card or hard drive. In this case, the temporary place is the (actual) path of the sound file, the reference path is the zip file path followed by zip entry path of the sound file. For example, in reference path "/folder1/folder2/snd.zip/zipped_folder/snd.wav", zip file path is "/folder1/folder2/snd.zip", zip entry path is "zipped_folder/snd.wav".

4. Load Resource Files On Different Platforms

MFP is a cross-platform programming language. Clearly running an App in Android device is very much different from running an MFP script in PC. In particular, resource accessories (i.e. sound and image files) needed by an MFP game script can be saved in any folder in PC, providing that their paths are correctly referred in the script. Comparatively, in an Android APK, all resource files are saved in APK's asset folder and there is no hard drive path available.

MFP's cross-platform solution is saving all resource files in the same folder as the script, and then call function get_src_file_path() to return the script's folder. From this folder, MFP can find full path of each resource file and then call load_image and play_sound to read them.

For an MFP App, a zip file called resource.zip is created inside the asset folder of the APK package. All the resource files are zipped into resource.zip. When the MFP App is running, functions load_image_from_zip and play_sound_from_zip are called to read resource files.

When packaging an MFP script into an APK, developer has to tell MFP which resource files should be copied into the resource.zip file. For example:

```
...

@build_asset copy_to_resource(get_upper_level_path(get_src_file_path()) + "eatfood.wav", "sounds/eatfood.wav")

if is_mfp_app()

    play_sound_from_zip(get_asset_file_path("resource"), "sounds/eatfood.wav", 1, false)

else

    play_sound(get_upper_level_path(get_src_file_path()) + "eatfood.wav", false)

endif

...
```

. In the above code, resource file name is eatfood.wav. This resource file is placed in the same folder as the script file so that its path is get_upper_level_path(get_src_file_path()) + "eatfood.wav" . If not an MFP App, the script can call function play_sound to read the file from the path. However, if running an MFP App, the script file is packaged inside APK. Consequently, eatfood.wav has to be saved in resource.zip in APK's asset folder. The path of resource.zip is obtained by calling get_asset_file_path("resource"), "sounds/eatfood.wav" means the entry path in the zip file, and 1 means Android asset resource file. Function play_sound_from_zip will extract the eatfood.wav from resource.zip and start to play.

When creating an APK package, annotation @build_asset tells MFP compiler to copy "eatfood.wav" from SD card to the resource.zip file in APK's asset folder. Copy_to_resource is actually an MFP function. Its first parameter is resource file's source path, i.e. where the resource file is located in SD card or hard drive. The second parameter is the test path, i.e. the zip entry in resource.zip file. Different from other MFP functions, function copy_to_resource is located in citingspace ::mfp_compiler. This citingspace is only automatically loaded when building APK package.

Also note that @build_asset must be placed inside a function. It takes no effect if being placed before function statement or after endf statement.

Section 4 Process User Inputs

At this stage, MFP provides four functions to read and analyse player's inputs. They are:

Function Name	Function Info

pull_event	::mfp::graph_lib::event::pull_event(1) : pull_event(display) pull an input event (e.g. mouse or touch pad event) out from screen display’s event list. If there is no event, or the display is not a screen display but an image display, it returns Null. Otherwise, it returns the event.
get_event_type	::mfp::graph_lib::event::get_event_type(1) : get_event_type(event) returns an integer which is type of the event. At this stage it supports the following events: GDI_INITIALIZE (type is 1, when a screen display is created), GDI_CLOSE (type is 10, when a screen display is shutdown), WINDOW_RESIZED (JAVA platform only, type is 21, when a screen display window is resized), POINTER_DOWN (type is 102, when a mouse button is pushed down in PC or user’s finger is tapping down in Android), POINTER_UP (type is 103, when a mouse button bounces up in PC or user’s finger is moving away from touchpad in Android), POINTER_CLICKED (type is 104, when a mouse button is clicked in PC or user’s finger taps touchpad in Android), POINTER_DRAGGED (type is 105, when a mouse or user’s finger is dragging. Different from POINTER_SLIDED event, this event is continously triggered during the dragging process), POINTER_SLIDED (type is 106, when a mouse is dragged to the destination and its button is released in PC, or when user’s finger drags to the destination and starts to leave touchpad in Android. Different from POINTER_DRAGGED, this event is triggered once-off), POINTER_PINCHED (Android only, type is 201, when user pinches to zoom in Android).

get_event_type_name	::mfp::graph_lib::event::get_event_type_name(1) : get_event_type_name(event) returns a string which is type name of the event. At this stage it supports the following events: ″GDI_INITIALIZE″ (when a screen display is created), ″GDI_CLOSE″ (when a screen display is shutdown), ″WINDOW_RESIZED″ (JAVA platform only, when a screen display window is resized), ″POINTER_DOWN″ (when a mouse button is pushed down in PC or user’s finger is tapping down in Android), ″POINTER_UP″ (when a mouse button bounces up in PC or user’s finger is moving away from touchpad in Android), ″POINTER_CLICKED″ (when a mouse button is clicked in PC or user’s finger taps touchpad in Android), ″POINTER_DRAGGED″ (when a mouse or user’s finger is dragging. Different from ″POINTER_SLIDED″ event, this event is continously triggered during the dragging process), ″POINTER_SLIDED″ (when a mouse is dragged to the destination and its button is released in PC, or when user’s finger drags to the destination and starts to leave touchpad in Android. Different from ″POINTER_DRAGGED″, this event is triggered once-off), ″POINTER_PINCHED″ (when user pinches to zoom in Android).
get_event_info	::mfp::graph_lib::event::get_event_info(2) : get_event_info(event, property_name) returns a property of event. Its first parameter is the event and its second parameter is string based property name. The GDI_INITIALIZE and GDI_CLOSE events don’t have any properties. WINDOW_RESIZED event has four integer properties which are ″width″ (current width of the window), ″height″ (current height of the

	window), ″last_width″ (width of the window before this event was triggered), ″last_height″ (height of the window before this event was triggered). POINTER_DOWN event has three properties which are ″button″ (an integer property telling developer which mouse button triggers this event in PC or always zero in Android), ″x″ (x coordinate which is a double) and ″y″ (y coordinate which is a double). Similar to POINTER_DOWN, POINTER_UP event and POINTER_CLICKED event also have three properties which are ″button″, ″x″ and ″y″. POINTER_DRAGGED event and POINTER_SLIDED event both have five properties which are ″button″, ″x″, ″y″, ″last_x″ and ″last_y″. Among them, ″button″ has same meaning as for POINTER_DOWN, ″last_x″ and ″last_y″ are coordinates before the event is triggered, and ″x″ and ″y″ are coordinates after the event is triggered. POINTER_PINCHED event has eight double value properties which store the coordinates of fingers before pinching and after pinching. The properties are ″x″, ″y″, ″x2″, ″y2″, ″last_x″, ″last_y″, ″last_x2″, and ″last_y2″.

When a game starts, player's every input event is saved in a user input event list. By calling function pull_event repeatedly, game player's inputs are extracted and processed. In the Hungry Snake game, the code is

```
do  // looping to read player's input events

    variable giEvent = pull_event(DISPLAYSURF)

    if giEvent == Null

        // no event to handle

        break

    elseif get_event_type_name(giEvent) == "GDI_CLOSE"

        // quit

        return -1
    elseif get_event_type(giEvent) == 106 // mouse or finger slided
        // x1 and y1 are the coordinate when sliding starts, x2 and y2 are the coordinate when sliding finishes
        variable x1 = get_event_info(giEvent, "last_x")
        variable y1 = get_event_info(giEvent, "last_y")
        variable x2 = get_event_info(giEvent, "x")
        variable y2 = get_event_info(giEvent, "y")
        // ensure sliding event to control snake moving direction doesnt happen in button area.
        if or(!shouldDrawButtons, x1 < btnsLeft, x1 > btnsRight, y1 < btnsTop, y1 > btnsBottom)
            // calculate moving direction
            if abs(y2 - y1) > abs(x2 - x1)
                if y2 > y1    // move down
                    moveDirection = -i
                else // move up
                    moveDirection = i
                endif
            elseif abs(y2 - y1) < abs(x2 - x1)
                if x2 > x1    // move right
                    moveDirection = 1
                else // move left
                    moveDirection = -1
                endif
            endif // if y2 - y1 == x2 - x1, do nothing
        endif
    elseif get_event_type_name(giEvent) == "POINTER_DOWN" // mouse or finger tapped down
        if gameIsOver
            // if game is over at this level, identify if player can go to next level based on the score
            if and(level < 2, score >= score_thresh)
                return 1    // can go to next level.
            else
                return 0    // cannot go to next level.
            endif
        elseif (shouldDrawButtons)
            // if game is not over at this level, identify if a button is hit
            xHit = get_event_info(giEvent, "x")
            yHit = get_event_info(giEvent, "y")
            if and(xHit >= leftBtnLT[0], yHit >= upBtnLT[1])
                // this event happens in the button area. If any button is hit,
                // set button pushed state to be true and change snake moving direction.
                if isButtonHit([xHit, yHit], upBtnLT, btnW, btnH)
                    upBtnPushed = true
                    moveDirection = i
                elseif isButtonHit([xHit, yHit], downBtnLT, btnW, btnH)
                    downBtnPushed = true
                    moveDirection = -i
                elseif isButtonHit([xHit, yHit], leftBtnLT, btnW, btnH)
                    leftBtnPushed = true
                    moveDirection = -1
                elseif isButtonHit([xHit, yHit], rightBtnLT, btnW, btnH)
                    rightBtnPushed = true
                    moveDirection = 1
                endif
            endif
        endif
        endif
until false
```

Section 5 Game Example: Hungry Snake

Previous sections have demonstrated how to open display window, how to draw static objects and animation, and how to process player's inputs. Based on the knowledge, a full Hungry Snake game can be easily implemented.

The idea of Hungry Snake game is, a snake is moving in its moving space, and player drags finger or pushes buttons to direct snake's movement. When snake's head hits the food (i.e. snake's head overlaps food's cell), the game plays eating food sound, places the food in a blank cell, updates the score, and increases snake body's length by one cell by drawing snake's head in the new cell. If snake's head hits wall or its body, the game plays hitting wall sound and exits. If the snake hits nothing, it just moves and its body size doesn't change. Clearly, moving the snake simply means removing the its tail and drawing its head. The detailed code is listed as below:

```
    // calculate the new snake head place (in grid coordinate)
    variable newX = mod(snakePlace[0][0] + deltaX, gridWidthDim)
    variable newY = mod(snakePlace[0][1] + deltaY, gridHeightDim)
    variable tail2Remove = null, head2Add = null, newFood = null, newScore = null    // the things to draw.
    if and(newX == foodPlace[0], newY == foodPlace[1])
        // eat the food. play the eat food sound.
        @build_asset copy_to_resource(get_upper_level_path(get_src_file_path()) + "eatfood.wav", "sounds/eatfood.wav")
        if is_mfp_app()
            play_sound_from_zip(get_asset_file_path("resource"), "sounds/eatfood.wav", 1, false)
        else
            play_sound(get_upper_level_path(get_src_file_path()) + "eatfood.wav", false)
        endif
        // snake head is at old food's place
        head2Add = [newX, newY]
        // because snake head has occpied the cell, food or snake body cannot take it again
        excludeBarrierPlace[newX, newY] = 2
        // insert new head's position in the snake position list
        snakePlace = insert_elem_into_ablist(snakePlace, 0, head2Add)
        // calculate new food place
        newFood = foodPlace = getNextFoodPlace(excludeBarrierPlace, wallPlace, snakePlace, gridWidthDim, gridHeightDim)
        // update score
        newScore = score = score + size(snakePlace)[0]
    elseif excludeBarrierPlace[newX, newY] != 0
        // hit the wall or itself. Play the hit wall sound and game is over at this level.
        @build_asset copy_to_resource(get_upper_level_path(get_src_file_path()) + "hitwall.wav", "sounds/hitwall.wav")
        if is_mfp_app()
            play_sound_from_zip(get_asset_file_path("resource"), "sounds/hitwall.wav", 1, false)
        else
            play_sound(get_upper_level_path(get_src_file_path()) + "hitwall.wav", false)
        endif
        gameIsOver = true
    else
        // normal move
        head2Add = [newX, newY]
        // because snake head has occpied the cell, food or snake body cannot take it again
        excludeBarrierPlace[newX, newY] = 2
        // insert the new snake head place into snakePlace list
        snakePlace = insert_elem_into_ablist(snakePlace, 0, head2Add)
        // remove tail of the snake
        variable tailIdx = size(snakePlace)[0] - 1
        tail2Remove = snakePlace[tailIdx]
        // now the cell is available for food and snake's body
        excludeBarrierPlace[tail2Remove[0], tail2Remove[1]] = 0
        // remove the old snake tail place into snakePlace list
        snakePlace = remove_elem_from_ablist(snakePlace, tailIdx)
    endif

    // game hasn't been over so that redraw the snake, food and score
    if head2Add != null
        // remove all the old painting event requests for snake from the painting request scheduler
        drop_old_painting_requests(["snake", totalSnakeLen - size(snakePlace)[0] + 1], DISPLAYSURF)
        // redraw snake body cell by cell
        draw_rect(["snake", totalSnakeLen], DISPLAYSURF, calcTopLeft(head2Add, cellSize, xMargin, yMargin), cellSize, cellSize, snakeColor, 0)
        totalSnakeLen = totalSnakeLen + 1
    endif
    if newFood != null
        // remove the old painting event request for food from the painting request scheduler
        drop_old_painting_requests("food", DISPLAYSURF)
        // calculate food's position and redraw the food
        foodPlaceXY = calcTopLeft(newFood, cellSize, xMargin, yMargin)
        draw_image("food", DISPLAYSURF, foodImage, foodPlaceXY[0], foodPlaceXY[1], cellSize/foodImageSize[0], cellSize/foodImageSize[1])
    endif
    if newScore != null
        // repaint score
        drop_old_painting_requests("score", DISPLAYSURF)
        draw_text("score", DISPLAYSURF, ""+newScore, [10, windowHeight - 32], scoreColor, 25)
        endif

        update_display(DISPLAYSURF) // update game display window
```

The whole example can be found in the hungry_snake.mfps file in 2d games/hungry_snake sub-folder in the manual's sample code folder. A screen snapshot of the Hungry Snake game running in an Android phone is shown as below:

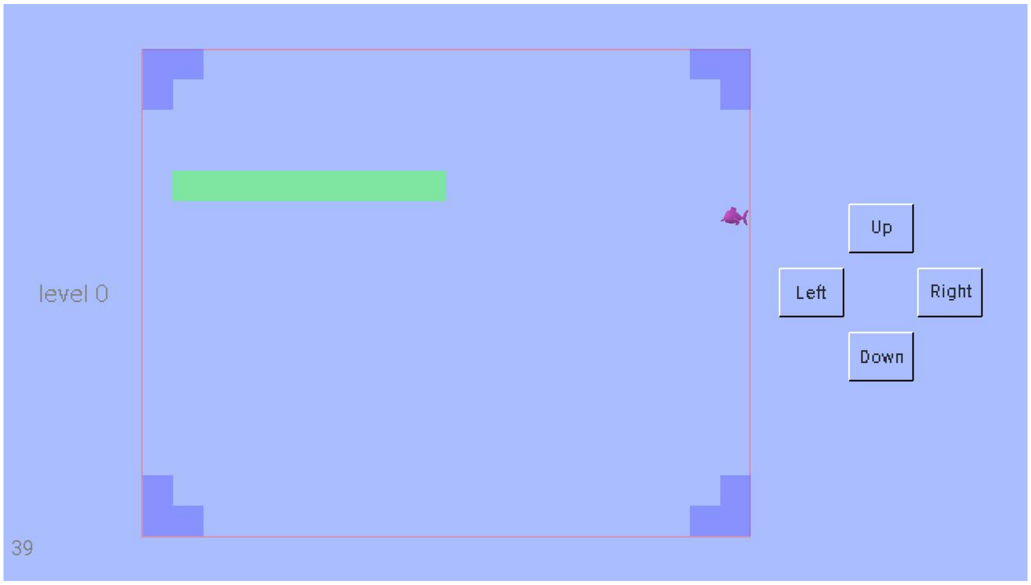


Figure 8.5: Screen snapshot of the Hungry Snake game running in an Android phone.

Section 6 Introduction of the GemGem Game

Besides the Hungry Snake game, Scientific Calculator Plus also provides a GemGem game example. This is a typical match and remove game. Its screen snapshot is below:

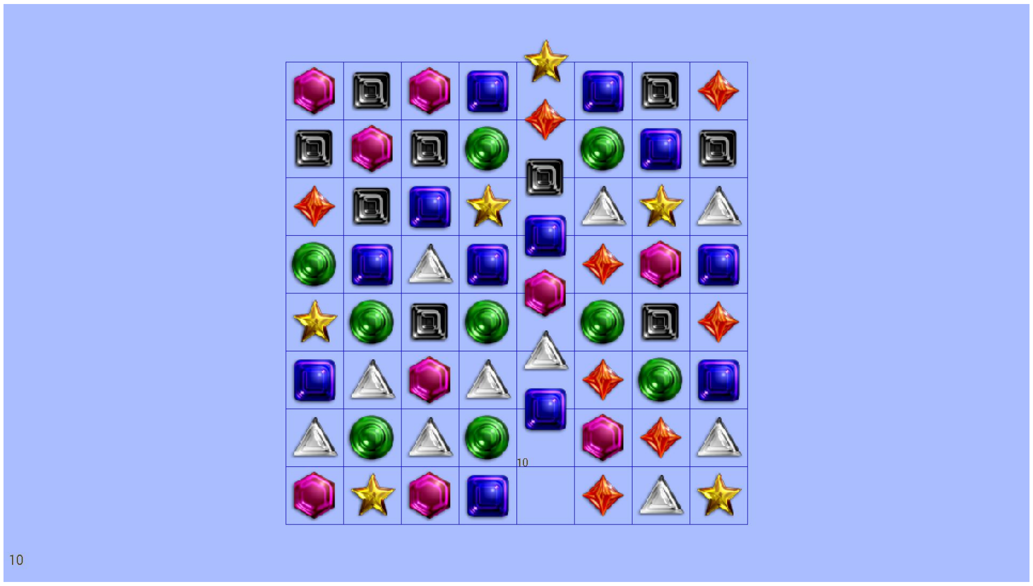


Figure 8.6: Screen snapshot of the GemGem game running in an Android phone.

When the game starts, MFP creates a 8*8 game panel. In each cell of the panel there is a gem. Player selects two adjacent gems to swap. If three or more adjacent gems in the same line or column turn to identical, this swapping is successful. The game plays successful swapping sound. And the identical gems are eliminated. Then above gems drop down to fill the spaces left by the removed gems. However, if no three or more adjacent gems become identical, this swapping is unsuccessful. MFP plays failure sound. And then the swapped gems are swapped back.

The principle of the GemGem game is similar to the Hungry Snake game. First a display window is created. Then animated image(s) are drawn on the display window. Meanwhile, player's inputs will determine what next animation step will be. When building GemGem game's APK, annotation @build_asset will also copy sound and image resource files to resource.zip file and save the zip file in APK's asset folder.

However, the GemGem game is more difficult than the Hungry Snake game from coding's perspective since it needs to store features of each gem. A gem has the following features: the first one is its appearance, e.g. green round or blue square or pink hexagon, etc; the second one is the row of the gem (i.e. y); the third one is the column of the gem (i.e. x); the fourth is gem's moving direction. Clearly, a diffused solution to implement these features is object-oriented programming. However, at this stage MFP doesn't support OO. As such, only dictionary or list can be used.

MFP does provides dictionary functions like get_value_from_abdict. Chosing this way, developer can simply call like get_value_from_abdict(gem_obj_dictionary, "x") to get a feature of the gem, in this case it is the row id. This clearly is very straight forward but not fast enough.

Alternatively, MFP uses array to store the features. In the game, a four element array is allocated for each gem. The first one (index is 0) is the appearance of the gem; the second one is line number; the third one is column number and the last one is moving direction. And because it is an array, the game can read each gem's feature quickly.

Another obstacle is how to effectively emulate the dropping process of gem sequences. Note that when a column of gems drop down, it may include several gem bunches. One bunch has at least one gem. And there are blank cells between two bunches. If there are several falling gem bunches in one column, when the first bunch has landed other bunches are still dropping. In order to cope with this situation, developer has to create a list whose elements are a bunch of dropping gems. When a gem lands, developer needs to remove the first element from the list. And no more element in the list means all the dropping gems have settled.

To handle list and dictionary, MFP provides a set of algorithms to process array-based list and dictionary:

Function Name	Function Info
append_elem_to_ablist	::mfp::data_struct::array_based::append_elem_to_ablist(2) : append_elem_to_ablist(array_based_list, ref_of_elem) appends a reference of value ref_of_elem at idx to an array based list array_based_list. It returns updated array based list. The parameter array_based_list shares elements with returned value.
concat_ablists	::mfp::data_struct::array_based::concat_ablists(2) : concat_ablists(list1, list2) concatenates array based list2 to array_based list1 and returns merged array based list. The parameters list1 and list2 shares elements with returned value.
create_abdict	::mfp::data_struct::array_based::create_abdict(0) : create_abdict() creates an empty array based dictionary.
get_elem_from_ab	::mfp::data_struct::array_based::get_elem_from_

list	ablist(2) : get_elem_from_ablist(array_based_list, idx) returns a reference of the value at idx from an array based list array_based_list. If the idx is invalid, an exception is thrown.
get_value_from_abdict	::mfp::data_struct::array_based::get_value_from_abdict(2) : get_value_from_abdict(array_based_dictionary, key) returns a reference to string based key’s value from array_based_dictionary. If the key doesn’t exist, it throws an exception.
insert_elem_into_ablist	::mfp::data_struct::array_based::insert_elem_into_ablist(3) : insert_elem_into_ablist(array_based_list, idx, ref_of_elem) inserts a reference of value ref_of_elem before idx of array based list array_based_list. It returns updated array list. The parameter array_based_list shares elements with returned value. If idx is not valid, an exception will throw.
remove_elem_from_ablist	::mfp::data_struct::array_based::remove_elem_from_ablist(2) : remove_elem_from_ablist(array_based_list, idx) removes the idxth element from an array based list array_based_list. It returns updated array list. The parameter array_based_list shares elements with returned value. If idx is not valid, an exception will throw.
set_elem_in_ablist	::mfp::data_struct::array_based::set_elem_in_ablist(3) : set_elem_in_ablist(array_based_list, idx, ref_of_elem) set a reference of value ref_of_elem at idx for an array based list

	array_based_list. Note that if the idx is invalid, an exception is thrown.
set_value_in_abdict	::mfp::data_struct::array_based::set_value_in_abdict(3) : set_value_in_abdict(array_based_dictionary, key, value) sets reference of the value to key into array_based_dictionary and returns the updated array_based_dictionary. If the key doesn't exist, it will be created. Note that key must be a string while value can be any data type.

Different from existing array functions, the above functions accept reference instead of value, i.e. whole clone, of parameter(s). This means, for example, when a new element, which is an array, is inserted into a list by calling function insert_elem_into_ablist, and the new element's value is changed later on, then the list is also changed. Clearly, accessing reference of parameter is much faster than cloning parameter.

The calculation of the GemGem game is very instensive. And painting on physical screen is an extremely slow process. To ensure realtime performance, the GemGem game creates "image display" for the whole game panel and every gem column. Painting events are applied to the "image displays", and then the snapshots of the "image displays" are "pasted" onto the display window by calling draw_image. The detailed codes are shown below:

A. The following code create a "image display" for each gem column.

```
......

  // This function creates an image display for each column

  variable thisColumnImgs = alloc_array([boardWidth])

  // initialize column image displays

  for variable idx = 0 to boardWidth - 1

      if size(dropSlots[idx])[0] <= dropSlotsIdx[idx]

          continue

      endif

      movingGemsColumnDisplays[idx] = open_image_display(Null)
```

```
    set_display_size(movingGemsColumnDisplays[idx], gemImgSize, windowHeight)
  next
......
```

B. The following code applies painting events to an “image display”.

```
......
        if and(boardCopy[x][y + 1] == emptySpace, boardCopy[x][y] != emptySpace)
          // The gem in this space drops
          if y + 1 < yLast    // no gem immediately below this gem is falling
            // start a new bunch, not add dropping direction because it is default.
            droppingGems[x]  =  append_elem_to_ablist(droppingGems[x],  [[boardCopy[x][y], x, y]])
          else // a gem immediately below this gem is falling
            // append this gem to existing bunch
            droppingGems[x][size(droppingGems[x])[0] - 1] = append_elem_to_ablist(droppingGems[x][size(droppingGems[x])[0] - 1], [boardCopy[x][y], x, y])
          endif
          yLast = y
          variable theX = xMargin + x * gemImgSize, theY = yMargin + y * gemImgSize
          clear_rect("gemgem", gemsImageDisplay, [theX, theY], gemImgSize, gemImgSize)
          draw_image("gemgem", movingGemsColumnDisplays[x], GEMIMAGES[boardCopy[x][y]], 0, theY, scaledRatio, scaledRatio)
          boardCopy[x][y] = emptySpace
        endif
......
```

C. The snapshots of the “image displays” are pasted onto the physical display window.

```
......
    drop_old_painting_requests("gemgem", DISPLAYSURF)
    draw_image("gemgem", DISPLAYSURF, gemsOnBoardImage, 0, 0, scaledRatio, scaledRatio)
```

```
        variable absProgress = round(0.01*progress*gemImgSize)

        draw_image("gemgem", DISPLAYSURF, movingGemsImage, 0, absProgress, scaledRatio, scaledRatio)

......
```

The whole example can be found in the gemgem.mfps file in 2d games/gemgem sub-folder in the manual's sample code folder.

Summary

MFP programming language is a cross-platform tool. Using this language, developer can build cross-platform games without changing code. Developers will be more productive because they can write / debug code in a PC and deploy code into any Android device.

The workflow of MFP game programming is

1. Open game display window;

2. Read player's inputs;

3. Draw animation;

4. Go back to step 2 and continue.

Two approaches can be chosen to create the animation. First is applying the painting events, i.e. drawing points, rectangles, strings etc., to physical screen. This solution accesses video memory very frequently so that is very slow. Alternatively, painting events can be directed to "image displays", then developer pastes the snapshots of the "image displays" to display window, i.e. physical screen. This needs much less video memory reading/writing so that it is faster. Clearly, the second approach is more feasible for game.

Developer also needs to understand the mechanism of the painting functions, i.e. draw_image, draw_oval etc. These functions do not paint immediately after been called. Instead, they create a painting event and the painting event is appended to a painting event scheduler. The scheduler will call the event later on. If developer wants the scheduler to execute painting events now, function update_screen should be called. Animation, in essence, is applying different painting events in each animation step so that it looks like image being moving. This is achieved by dropping old painting events from painting event scheduler and adding new painting events into it.

So far MFP doesn't support object oriented programming. As such developer has to save features of objects in dictionaries or lists. MFP has provided a set of dictionary/list APIs to do this work.

After game script is developed, developer needs to call annotation @build_asset to copy sound or image files from APK's asset folder to resource.zip file. This ensures the game can run both as an MFP App in Android and as a script in PC.

The best practice is getting hands dirty. Now users of Scientific Calculator Plus can start to develop a small game to experience the capability of MFP programming language.

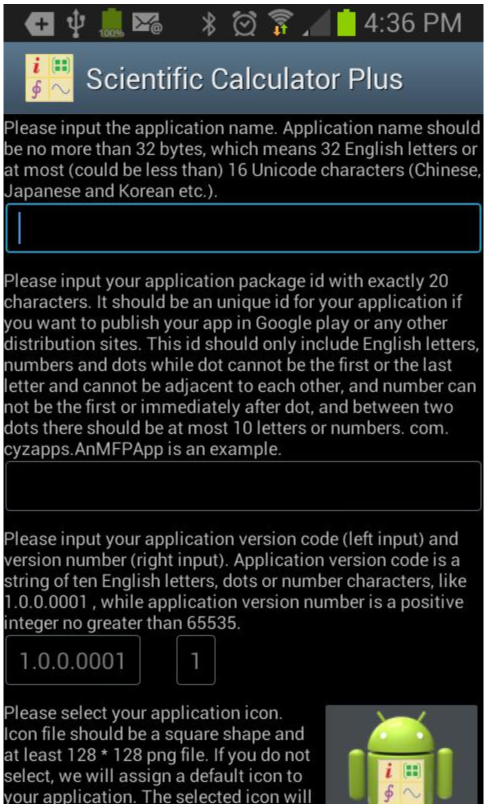
Chapter 9 Building User's App

A major function of Scientific Calculator Plus is creating an independent Android app from an arbitrary MFP function whether it is defined by user or provided by software. The created app can be installed and published. The benefits to use a generated app instead of original MFP source code include:

1. App built by user includes all the capability provided by the original source code but it is much smaller than the whole Scientific Calculator Plus application which is required by MFP source code to run;

2. No need for MFP developer to write separate instructions for the function. The app can embed instructions into its dialog boxes and help page;

3. When building an app, Scientific Calculator Plus compiles the code and does not link unused functions so that the app's loading procedure is much faster.



Please input the application name. Application name should be no more than 32 bytes, which means 32 English letters or at most (could be less than) 16 Unicode characters (Chinese, Japanese and Korean etc.).

Please input your application package id with exactly 20 characters. It should be an unique id for your application if you want to publish your app in Google play or any other distribution sites. This id should only include English letters, numbers and dots while dot cannot be the first or the last letter and cannot be adjacent to each other, and number can not be the first or immediately after dot, and between two dots there should be at most 10 letters or numbers. com. cyzapps.AnMFPApp is an example.

Please input your application version code (left input) and version number (right input). Application version code is a string of ten English letters, dots or number characters, like 1.0.0.0001 , while application version number is a positive integer no greater than 65535.

1.0.0.0001 1

Please select your application icon. Icon file should be a square shape and at least 128 * 128 png file. If you do not select, we will assign a default icon to your application. The selected icon will

Figure 9.1: The first step to build up an MFP app.

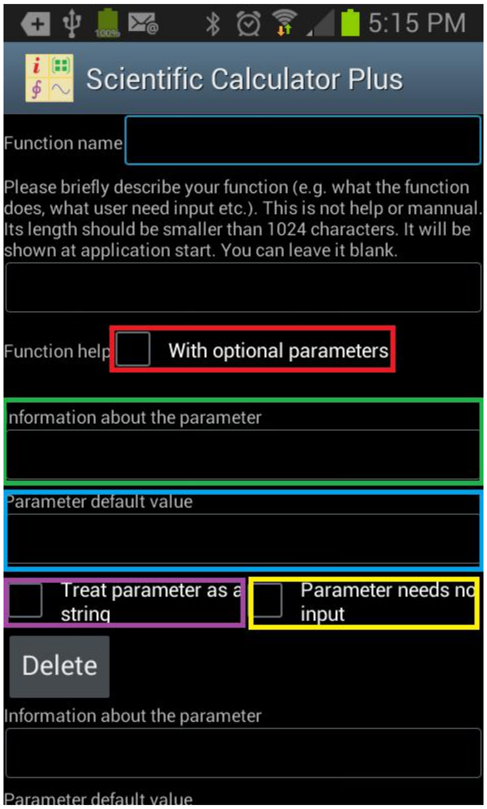
It is very easy to create an app. In Scientific Calculator Plus for JAVA, user needs to click the "tools" menu and select "Create MFP App" sub-menu. In an Android device, user simply taps “Build MFP App” icon. Then user will see the above interface. User simply needs to input the prompts required keeping in mind the following points:

1. Application name is the name shown under the icon in Android’s application tray. This name is also used in Android application manager;

2. Application package id must be identical because it is the “name” which diversifies an app from others. Note that this id is also used in Section 2 of Chapter 7;

3. The default working folder is the working folder when the app starts. The plotted graphs and created files which are based on a relative path are saved in this folder. If user does not set this item, the folder will be the sub-string after the last dot (.) in the app’s package id.

After all the above settings are input, user taps “Next” to select the MFP function to compile, as shown in the below chart. If user does not check the “With optional parameters” box (in the red rectangle), the number of parameters needed by the function should match the number of parameters user added here. User is required to define each parameter added by him/her. In particular, the “Information about the parameter” box (in the green rectangle) is for the parameter prompt, the “Parameter default value” box (in the blue rectangle) is for the default value of the parameter. If the app’s user does not input a value for the parameter, the default value will be used. If the box “treat parameter as a string” (in the purple rectangle) is checked, the value input by the app’s user for the parameter is looked on as a string and double quotes will be automatically added at the beginning and end of the input.If the box “Parameter needs no input” (in the yellow rectangle) is checked, the app’s user will not see the input box of this parameter and the default value will be used when the app starts.


5:15 PM
Scientific Calculator Plus
Function name

Please briefly describe your function (e.g. what the function does, what user need input etc.). This is not help or mannual. Its length should be smaller than 1024 characters. It will be shown at application start. You can leave it blank.

□ With optional parameters

nformation about the parameter
arameter default value

□ Treat parameter as a string

□ Parameter needs nc input

Delete
Information about the parameter
Parameter default value

Figure 9.2: The second step to build an MFP app.

If the "With optional parameters" box is selected, the last parameter input will be used for the optional parameters. This parameter input cannot have any default value (even if user inputs something in the "Parameter default value" box). And input from the app's user for optional parameters is compulsory. After the app starts, user can input multiple lines for the optional parameter input and each line is for one optional parameter. If the box "treat parameter as a string" (in the purple rectangle) is checked, all the lines are treated as string and double quotes will be automatically added (i.e. the app's user shouldn't input double quotes). Otherwise, double quotes cannot be omitted if the app's user wants to input a string as an optional parameter.

Please note that, when an apk is built up, Scientific Calculator Plus does not copy all the mfps source files to the package. Instead, only related source code files are copied and compiled. Sometimes, for example when calling function integrate or plot_exprs, parameter(s) of to be compiled function is a string or a string based variable. In this case Scientific Calculator Plus cannot determine which functions will be called at compiling time because app's user may input a function call as parameter. Thus developer needs to annotate in the source code using keyword @compulsory_link explicitly telling Scientific Calculator Plus which functions are required when building apk. For example:

```
...

@compulsory_link function(::mfpexample::expr1, ::mfpexample::expr2(2))

integrated_result = integrate(expression_str, variable_str)

...
```

. In the above example, ::mfpexample::expr1 and ::mfpexample::expr2 are user defined functions which may be called by integrate. Because of the annotation, all the functions whose name is ::mfpexample::expr1 will be linked into apk. However, only if a function with exactly two parameters or with optional parameter(s) and whose name is ::mfpexample::expr2 will be linked into apk.

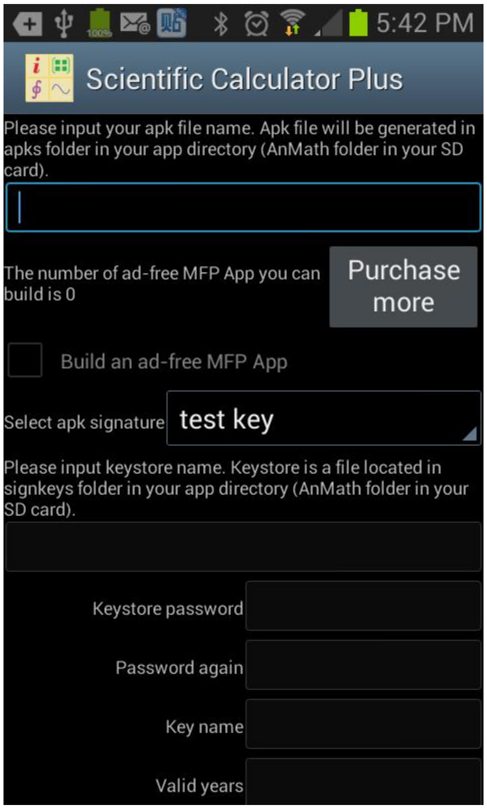
If user wants to link all the functions into apk, the following annotation should be used:

```
@compulsory_link all_functions
```

. However, this means the loading time when apk starts will be much longer. Also user has to ensure that all the functions are defined. Otherwise there will be error reported at compilation time.

The last thing about @compulsory_link annotation is that it has to be placed inside a function, i.e. after function statement and before endf statement. Otherwise it is useless.

The last step to build an apk is to set its name and sign, as shown in the following graph. User needs to select the testing key to sign so that no user name or password is required. However, apk signed by testing key cannot be published which means user can only install the generated apk or share it with friends. If user wants to publish an apk, s\he has to create a public key and input keystore password and key password, as well as personal information. The created key can be reused, but should be only used for (different versions) of a single app.



Scientific Calculator Plus

Please input your apk file name. Apk file will be generated in apks folder in your app directory (AnMath folder in your SD card).

The number of ad-free MFP App you can build is 0

□ Build an ad-free MFP App

Select apk signature: test key

Please input keystore name. Keystore is a file located in signkeys folder in your app directory (AnMath folder in your SD card).

Keystore password

Password again

Key name

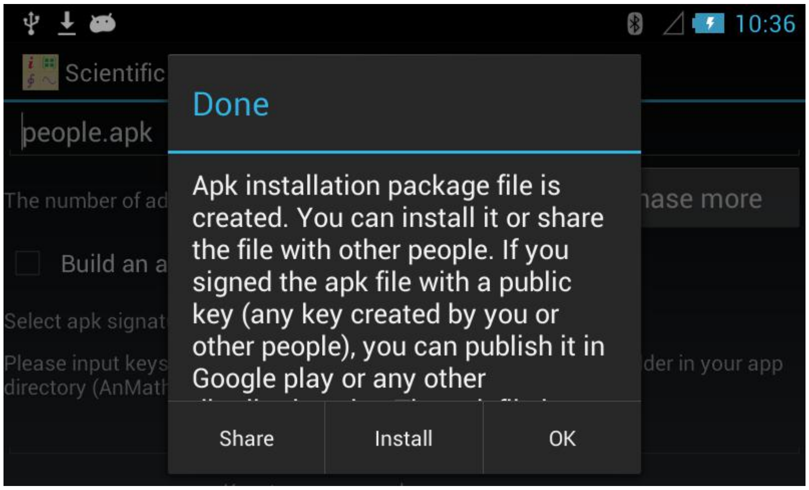
Valid years

Figure 9.3: Last step to create an MFP app.

After everything is done, user taps OK and Scientific Calculator Plus starts to create the app. If everything is OK, user will see the following dialog box. User can select to install or share the created app, or simply record the path of the apk file and tap OK to return to the main panel.

Please note that, if user selects to install the created app, s\he may get a notice to allow installation of apk from unknown sources. User, in this case, needs to enable the "Allow installation of apps from sources other than the Play Store" box in the security settings so that installation can continue.

To publish and distribute the created apk, user needs to register in a distribution website (e.g. Google Play), and then submit the apk file. Because all the apk files created by Scientific Calculator Plus include an MFP interpreter, they are inevitably similar to each other. To avoid any legal issue, user may be inquired about intellectual property of the submitted app. This inquiry will not hinder the apk from publishing after user explains the reason in some detail.



Done

Apk installation package file is created. You can install it or share the file with other people. If you signed the apk file with a public key (any key created by you or other people), you can publish it in Google play or any other

Figure 9.4:   An MFP app is successfully built.

Chapter 10 Using MFP in Your App

Scientific Calculator Plus is based on MFP programming language. This is an object oriented scripting language. It provides plentiful functions for 2D game development, complex number, matrix, (higher level) integration, 2D, polar and 3D chart, string, file operation, JSON data exchange and TCP/WebRTC communication. MFP now has been open sourced based on Apache 2.0 license. Therefore any individuals or companies can take advantage of this programming language. Clearly, if embedded in their projects, MFP could save developers a significant amount of time and resources to achieve their aims.

Github repo for MFP Android lib is located at https://github.com/woshiwpa/MFPAndroLib. Also, from version 2.1.1, Scientific Calculator Plus starts to includes MFP Android lib binary. After upgrade to a new version, the latest binary is copied automatically to local storage at Android/data/com.cyzapps.AnMath/files/AnMath folder. User can also manually copy the binary to local storage by tapping the "Run in PC or MAC" icon.

There are two aar binaries needed to embed MFP library into an Android project. One is MFPAnLib-release.aar. The other is google-webrtc-x.x.xxxxx.aar. Copy both of them from local storage's AnMath folder to the destination Android project. Developer can place them anywhere as long as gradle can find them. Let us assume they are saved in libs folder in a module called app. In this case, developer needs to add the following two lines in app module's build.gradle file:

```
// google-webrtc aar version may change in the future

implementation files('libs/google-webrtc-1.0.19742.aar')

implementation files('libs/MFPAnLib-release.aar')
```

Besides the two binaries, MFP also has some pre-defined scripts which provide developer many useful functions. In Scientific Calculator Plus, the pre-defined scripts are zipped in assets.7z file in local storage's Android/data/com.cyzapps.AnMath/files/AnMath folder. Inside the assets.7z there is a folder named predef_lib. Developer should copy the whole predef_lib folder into assets folder of developer app's project.

In the app's Application implemention, the following code should be added into the onCreate function:

```
public class YourAppImplementationClass extends androidx.multidex.MultiDexApplication {

 @Override

 public void onCreate() {
    super.onCreate();
    ... ...
    MFPAndroidLib mfpLib = MFPAndroidLib.getInstance();
    // initialize function has three parameters. The first one is application's context,
    // the second one is your app's shared preference name, and the last one is a boolean
    // value, with true means your MFP scripts and resources are saved in your app's
    // assets and false means your MFP scripts and resources are saved in your Android
    // device's local storage.
    // The following code is for the situation to save MFP scripts and resources in assets
    // of app. However, if developer wants to run scripts from local storage, uncomment
    // the following line and pass false to the third parameter of mfpLib.initialize function.
    // MFP4AndroidFileMan.msstrAppFolder = "My_App_MFP_Folder";
    mfpLib.initialize(this, "Your_App_Settings", true);
    MFP4AndroidFileMan mfp4AnFileMan = new MFP4AndroidFileMan(getAssets());
    // platform hardware manager has to be early initialized as it is needed to
    // analyze anotations in the code at loading stage.
    // other managers are loaded later at the first command.
    FuncEvaluator.msPlatformHWMgr = new PlatformHWManager(mfp4AnFileMan);
    MFPAdapter.clear(CitingSpaceDefinition.CheckMFPSLibMode.CHECK_EVERYTHING);
    // load predefined libs when app starts
    mfp4AnFileMan.loadPredefLibs();
    }
  ... ...
```

As explained above, there are two ways to save user defined MFP scripts and related resource files. One is saving in module's assets. In this case, developer has to create a zip file named userdef_lib.zip in module's assets. Inside this zip package is a folder named scripts. All the user defined MFP scripts are inside.

Some developers working on MFP scripts may need to load some resources, e.g. image or sound. Developer in this case needs to create another zip file named resource.zip in module's assets. Resource files should be packed inside.

In user defined MFP scripts, developer needs to use logic like the following code to tell MFP interpreter where the script can find its resource.

```
@build_asset copy_to_resource(iff(is_sandbox_session(), get_sandbox_session_resource_path() + "sounds/drop2death.wav", _
        is_mfp_app(), [1, get_asset_file_path("resource"), "sounds/drop2death.wav"], _
        get_upper_level_path(get_src_file_path()) + "drop2death.wav"), "sounds/drop2death.wav")
if is_sandbox_session()
  play_sound(get_sandbox_session_resource_path() + "sounds/drop2death.wav", false)
elseif is_mfp_app()
  play_sound_from_zip(get_asset_file_path("resource"), "sounds/drop2death.wav", 1, false)
else
  play_sound(get_upper_level_path(get_src_file_path()) + "drop2death.wav", false)
endif
```

So basically the if ... elseif ... else ... endif block in the above code tells MFP the current script needs to load a sound file named drop2death.wav. If the current code is running in a sandbox session, the wav file is saved in the sounds folder of the sandbox session's resource path. Here sandbox is a parallel computing concept which means execution environment of call block session sent from a remote MFP instance. If the current code is running in an MFP app, which is also the example we are now focusing on, the wav file is saved in sounds folder in app assets' resource.zip file. Note that the third parameter of function play_sound_from_zip is 1, which means get_asset_file_path function returns an Android app assets path. If the current code is running in local storage, e.g. SD card or PC's hard disk, the wav file is placed in the same folder as the script.

The @build_asset annotation above the if block tells MFP, if need to build an MFP app, or run a call block in a remote sandbox, where the resource file should be saved in the target side. MFP android library doesn't include the function to build an MFP app from MFP script(s). However, developer may need to send some codes and run them in a remote device. So the annotation may still be needed.

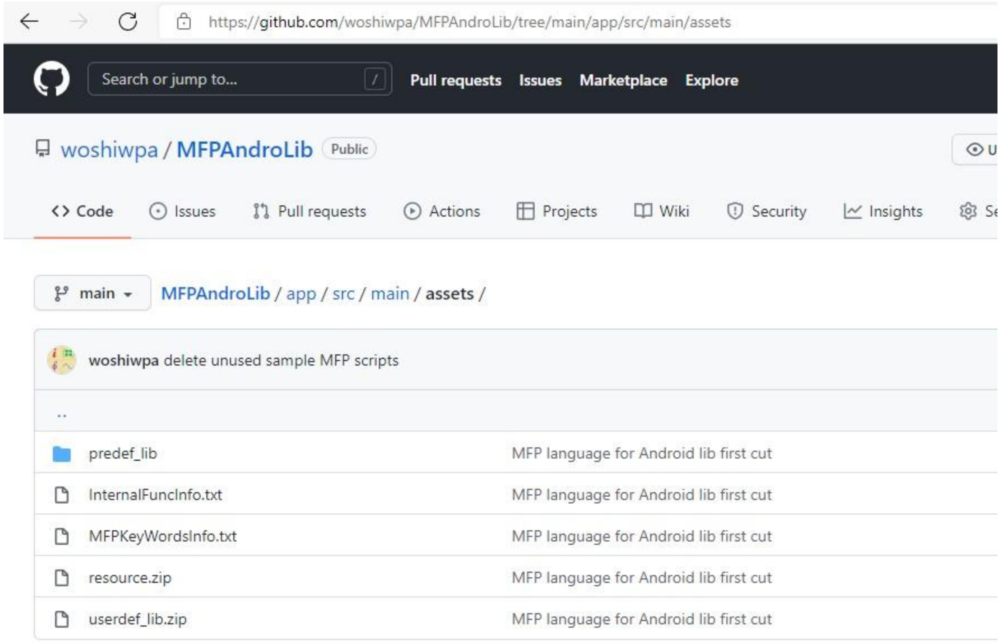
If there is no necessity to send code to remote to run, and developer only wants to save scripts and resources in assets, then only one line in the above code is needed, i.e.

```
play_sound_from_zip(get_asset_file_path("resource"), "sounds/drop2death.wav", 1, false)
```

If developer puts all user defined MFP scripts in local storage instead of assets, MFP needs to be told where the MFP folder is. The MFP folder is located in Android/data/your.app.package.id/files/. And scripts should be placed in the scripts subfolder of the MFP folder. Furthermore, if there is no necessity to send code to remote, only one line of MFP statement is needed to load resource. To the above example, the line of code should be

```
play_sound(get_upper_level_path(get_src_file_path()) + "drop2death.wav", false)
```

As shown in the following snapshot, in MFP Android lib's github repo, the assets folder of the sample app includes the following items: predef_lib, resource.zip, userdef_lib.zip, InternalFuncInfo.txt and MFPKeyWordsInfo.txt. As explained above, resource.zip and userdef_lib.zip are not required if developer decides to put user defined scripts in local storage. Also, InternalFuncInfo.txt and MFPKeyWordsInfo.txt are help info for built-in MFP functions and MFP key words respectively. In general, developer doesn't need them.



woshiwpa delete unused sample MFP scripts

..	
predef_lib	MFP language for Android lib first cut
InternalFuncInfo.txt	MFP language for Android lib first cut
MFPKeyWordsInfo.txt	MFP language for Android lib first cut
resource.zip	MFP language for Android lib first cut
userdef_lib.zip	MFP language for Android lib first cut

Figure 10.1: An MFP app is successfully built.

After all the binaries and MFP scripts and resources are copied to right place, developer needs to load user defined MFP scripts. If the scripts are saved in app assets, call MFP4AndroidFileMan.loadZippedUsrDefLib function to load the scripts. Otherwise, MFP4AndroidFileMan.reloadAllUsrLibs should be called to do the job:

```
// Now start to load functions

MFP4AndroidFileMan mfp4AnFileMan = new MFP4AndroidFileMan(am);

// if we repeatedly run this function, we have to call the following statement to ensure

// MFP citingspace is clear. However, if we only run this function once, MFP citingspace

// is clear anyway so the following line can be removed.

// MFPAdapter.clear(CitingSpaceDefinition.CheckMFPSLibMode.CHECK_USER_DEFINED_ONLY);

// load user defined lib.

if (mfp4AnFileMan.isMFPApp()) {

  MFP4AndroidFileMan.loadZippedUsrDefLib(MFP4AndroidFileMan.STRING_ASSET_USER_SCRIPT_LIB_ZIP, mfp4AnFileMan);

} else {

  // use the following line if you put your MFP scripts in local storage

  MFP4AndroidFileMan.reloadAllUsrLibs(ActivityAnMFPMain.this, -1, null);

}
```

The last step before running MFP scripts is initializing MFP interpreter environment. This cannot be done immediately after initializing MFP android lib because initializing android lib requires application's context while here activity's context is needed.

```
// now initialize MFP interpreter environment

MFPAndroidLib.getInstance().initializeMFPInterpreterEnv(ActivityAnMFPMain.this, new CmdLineConsoleInput(), new CmdLineLogOutput());
```

Note that here developer needs to pass CmdLineConsoleInput and CmdLineLogOutput into the initializeMFPInterpreterEnv function besides activity's context. CmdLineConsoleInput and CmdLineLogOutput, derived from MFP lib abstract classes ConsoleInputStream and LogOutputStream respectively, tell MFP how to read input of MFP's input and scanf functions from the app and how to show MFP's print output strings in the app. As such the implementation of two classes really depends on the developer. For example, developer may want to swallow all the outputs so the outputString function is overridden to do nothing. And if MFP's input function is never called, developer may just throw an exception in CmdLineConsoleInput's inputString function although this behavior is not recommended. MFP Android lib project in github has already provided implementation examples for these two classes.

Now everything is ready so that developer can run MFP code. The code to run should be stored in a JAVA string. Use '\n' to separate lines. For example, "\n\nplot_exprs(\"x**2+y**2+z**2==9\")\ngdi_test::game_test::super_bunny::run( )\n" includes two valid MFP lines, one calls plot_exprs function and the other runs a super bunny game.

The final step would be straight forward. Depending on the number of statments included in str2Run, developer simply calls MFPAndroidLib.processMFPStatement or MFPAndroidLib.processMFPSession to run the MFP statements. strOutput includes final text shown after the run. If the single statement doesn't return anything or the session doesn't include a return statement, strOutput for MFP session run is empty. Otherwise, it is the returned value of the statement or session run or exception stack if the run failed. varAns is the variable storing the returned value. It keeps its original value, i.e. an MFP null, if there is no returned value. varAns is very important to developer because the raw returned value, i.e. the returned value with original type, can be extracted from it.

```
String[] statements = str2Run.trim().split("\\\n");

String strOutput;

Variable varAns = new Variable("ans", new DataClassNull());  // this variable stores returned value

if (statements.length == 1) {  // run single line

  strOutput = MFPAndroidLib.processMFPStatement(str2Run.trim(), new LinkedList(), varAns);

} else { // run multiple lines

  strOutput = MFPAndroidLib.processMFPSession(statements, new LinkedList(), varAns);

}// don't add this line if developer doesn't want to show the final output in app

new CmdLineLogOutput().outputString(strOutput);
```

To find out implementation details of the above codes, please go to github, download the repo for MFP Android lib and start to play by yourself.

Epilogue

Scientific Calculator Plus is a very powerful Android app. User will realize this and find many functions which are not included in any other calculator apps after reading this manual. The most appealing capability of Scientific Calculator Plus is plotting 3D graphs and game development. Actually Scientific Calculator Plus can partially replace the widely used graphing software tool GNUPlot in academia and industry. Also Scientific Calculator Plus's programming language, i.e. MFP, is powerful enough to gain an advantage over any hardware based programmable calculator sold in the market.

However, regretfully Scientific Calculator Plus is still far from being widely used. The reasons include:

1. The interface of Scientific Calculator Plus is not cool. Actually it is implemented by Android 2.2 APIs and its style has been obsolete;

2. Without detailed manual, Scientific Calculator Plus is hard to use even if the user him/herself is a programmer;

3. Marketing resources are very limited. In fact no resource has ever been fully allocated to marketing. Because of no marketing, downloading count in Google Store is very low.

As the developer of Scientific Calculator Plus, I fully understand the importance of the above three points which determines the app's life and the size of user group. Therefore, I have started a google group named MFP programming language (https://groups.google.com/forum/#!forum/mfp-programming-language). User can discuss and share their ideas and codes in this web site. I will also read and answer questions posted by user in this website frequently. Moreover, I suspended development work and started to write this manual half a year ago. I hope this manual is able to answer most of the questions from user.

As MFP programming language shares similar syntax as BASIC, it is very easy to use. But it is much more than BASIC. This manual, therefore, can be an initiation book for computer programming for users without any programming background. The interesting API functions will be a studying bonus for them.

The ugly GUI of Scientific Calculator Plus has been complained by users many times. However, this needs time to polish because Android devices need a group of icon files at different resolutions and screen sizes. Redrawing all the icons takes significant time and any commercial icon-drawing software is expensive. But I promise users that I will endeavor to make it looks better.

The last thing is marketing. I need users' help. I hope existing users could introduce this software to their friends, classmates and workmates if they think it is good. As such more people can try it.

This year sees more than 3000 calculator apps in Android market. Most of them have arrived at an innovation bottleneck because no more major functionality can be introduced and the development is now mainly focused on detail polishing. However, as a programming language, MFP possesses a very bright future. At this moment this language can only operate files and plot charts. If MFP can control web-browsers, send/receive emails, access MSG, USB and Bluetooth, and develop complicated user interfaces， its application will be boundless. As its developer, I have some novel ideas to plan MFP's next step. I hope I could share my findings and plan of MFP with other people, especially the existing and potential MFP users. I welcome any emails, whether complaints, suggestions or questions from you. My email address is cyzsoft@gmail.com.