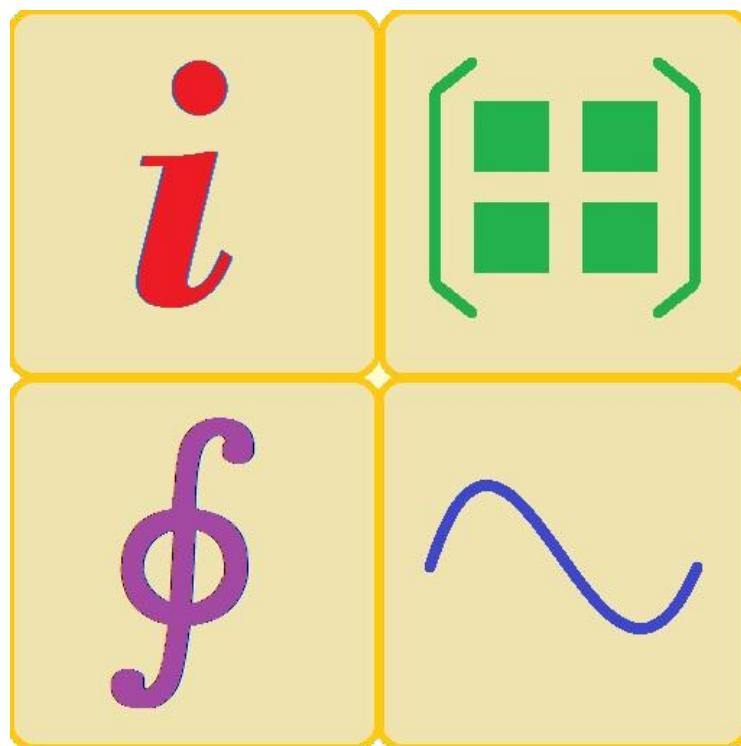


可编程科学计算器使用指南及 MFP程序开发语言用户手册



适用于可编程科学计算器 2.1.0.92 及其以上版本

可编程科学计算器使用指南及 MFP 程序开发语言用 户手册

© CYZSoft

湖北省武汉市洪山区华中师范大学

430000

电话: 13260560268 (中国) +61-4-25388821 (澳大利亚)

敬告:

以上联系方式仅用于商务合作, 如用户有任何关于软件的技术问题, 请发 email 给 cyzsoft@gmail.com 或者在百度 MFP 吧中发帖讨论。

目录

前言及开发者的话.....	1
第 1 章 可编程科学计算器初级用户使用指南.....	10
第 1 节 安装与启动.....	10
第 2 节 智慧计算器的使用.....	12
第 3 节 命令提示符的使用.....	22
第 4 节 绘制图形工具.....	25
第 5 节 计算微积分工具.....	41
第 6 节 设置输入键盘.....	41
第 7 节 管理文件和编写程序.....	42
第 8 节 设置工具.....	43
第 9 节 在电脑上运行可编程科学计算器工具.....	43
第 10 节 创建 MFP 应用工具.....	54
第 11 节 帮助工具.....	55
小结.....	56
第 2 章 MFP 编程语言基础.....	58
第 1 节 MFP 编程语言概述.....	58
第 2 节 MFP 编程语言基本数据类型.....	60
第 3 节 MFP 编程语言所支持的操作符.....	63
第 4 节 function, return 和 endf 语句.....	66

第 5 节 variable 语句.....	68
第 6 节 if, elseif, else 和 endif 语句.....	69
第 7 节 while, loop; do, until 和 for, next 语句.....	71
第 8 节 break 和 continue 语句.....	74
第 9 节 select, case, default 和 ends 语句.....	75
第 10 节 try, throw, catch 和 endtry 语句.....	77
第 11 节 solve 和 slvreto 语句.....	81
第 12 节 help 和 endh 语句以及@language 标注.....	88
第 13 节 citingspace 以及 using citingspace 语句.....	90
第 14 节 class 和 endclass 语句.....	109
第 15 节 call 和 endcall 语句.....	120
第 16 节 @compulsory_link 标注.....	126
第 17 节 @build_asset 标注.....	127
第 18 节 @execution_entry 标注.....	129
第 19 节 如何部署用户创建的 MFP 函数程序.....	132
小结.....	134
第 3 章 MFP 编程语言对数、字符串和数组的操作.....	136
第 1 节 MFP 对数的操作函数.....	136
1. MFP 对整数的操作.....	136
2. MFP 进制转换函数.....	139
3. MFP 逻辑操作函数.....	143
4. MFP 对复数的操作函数.....	146
第 2 节 MFP 对字符串的操作函数.....	148
第 3 节 MFP 对数组和矩阵的操作函数.....	153

1. MFP 创建数组的函数.....	153
2. 获取数组的尺寸和判断数组的特征.....	155
3. 对数组赋值.....	158
小结.....	162
第 4 章 MFP 编程语言对于各种数学和科学计算的支持.....	163
第 1 节 内置的数学常用变量.....	163
第 2 节 单位转换函数和返回物理化学常量值的函数.....	163
第 3 节 三角函数双曲三角函数.....	167
第 4 节 指数, 对数和次方函数.....	171
第 5 节 矩阵相关函数.....	175
第 6 节 表达式和微积分函数.....	181
第 7 节 统计、随机和排序函数.....	189
第 8 节 信号处理函数.....	194
第 9 节 阶乘求值函数、判断质数函数和多项式求根函数.....	197
小结.....	200
第 5 章 用 MFP 编程语言绘制图形.....	201
第 1 节 绘制表达式的图像.....	201
第 2 节 绘制常规坐标系下的二维图像.....	210
第 3 节 绘制极坐标系下的二维图像.....	222
第 4 节 绘制三维图像.....	228
小结.....	253
第 6 章 输入输出和文件操作.....	255
第 1 节 在命令提示符中输入输出.....	255
第 2 节 对字符串输入输出.....	268

第 3 章 文件内容读写及其相关函数.....	273
1. 打开和关闭文件.....	275
2. 文本文件的读写.....	278
3. 二进制文件的读写.....	283
第 4 章 文件属性操作函数.....	288
第 5 章 类似 Dos 和 Unix 命令的文件整体操作函数.....	300
第 6 章 进行复杂文件操作示例.....	308
小结.....	323
第 7 章 日期和时间以及系统函数.....	324
第 1 章 日期和时间的函数.....	324
第 2 章 系统相关函数.....	332
小结.....	335
第 8 章 用 MFP 编程语言进行游戏开发.....	336
第 1 章 创建，调整和关闭游戏显示窗口.....	336
第 2 章 在游戏显示窗口上作图.....	341
1. 游戏动画原理.....	341
2. 绘制几何图形.....	342
3. 绘制图像.....	350
4. 绘制文字.....	351
5. 动画例子.....	355
第 3 章 处理图像和声音.....	374
1. 创建，装载，克隆和保存图像.....	374
2. 修改图像.....	376
3. 声音的播放.....	381

4. 在不同的平台上装载声音图像附件.....	384
第 4 节 玩家输入的事件处理.....	386
第 5 节 游戏示例贪吃蛇的编程逻辑.....	392
第 6 节 削宝石游戏简介.....	397
小结.....	404
第 9 章 构建用户自己的应用.....	405
第 10 章 在你自己的应用中使用 MFP 安卓库.....	411
后记.....	418

前言及开发者的话

可编程科学计算器的第一版是在 2012 年发布的，从第一版开始，这个软件就支持编程。基于十几年的各种各样编程语言的使用经验和心得，我决定，这个编程语言必须如 Basic 一样简单，但又可以在日后扩展为像 JAVA 和 C++ 那样面向对象，甚至比面向对象更进一步。不同于普通的编程语言，它必须内建对数学的一些基本元素，比如矩阵，复数，的支持。我把它命名为 MFP，意思是面向并行计算的数学语言。

随着第一版的发布，后续版本和功能一步一步地添加进来。1.0.4 版实现了矩阵的除法；1.1 版增加了基于 JAVA 的电脑上运行的可编程科学计算器；1.2 版实现了自定义用户键盘；1.3.1 版实现了求多项式的根；1.4 版实现了解方程；1.5 版支持 3D 绘图；1.5.5 版支持极坐标绘图；1.6.0 版优化了效率；1.6.2 版完善了积分功能；1.6.3 版实现了文件操作；1.6.4 版添加数学公式识别的功能；1.6.6 版实现将用户自定义函数打包成为 APK 安装文件；1.7 版实现对微分的支持和引入引用空间的概念；1.7.1 版 MFP 能够如其他任何脚本语言一样执行 mfps 脚本文件；1.7.2 版可以开发游戏；1.8.0 版增加了对并行计算的支持；2.0.0 版引入 class 语句开始支持面向对象编程。每一次升级，我都争取给用户带来新的东西，对于我而言，这都是一次艰苦的努力。

那么，我的这些努力是否值得？可编程科学计算器这个软件，究竟能在哪些方面为用户发挥作用呢？

首先，当然是普通的日常生活。可编程科学计算器自带的智慧计算器组件，打开就是一个普通的计算器界面，购物消费的时候用它计算花销，再容易不过了。比如，去商场购物，猪肉一斤 18 块 7 毛 9，买了 2 斤 6 两，梳子一把五块四毛九，蜂蜜一瓶 41 块 5，其他蔬菜花了 19 块，计算总花销，输入 $18.79*2.6 + 5.49 + 41.5 + 19$ ，然后点击那个桔黄色圆中间有白色三角的开始按钮，就会得到总花费为 114 块 8 毛 5。参见下图。



图 0.1： 可编程科学计算器进行用于日常生活的普通四则运算。

第二，用于数学教育，可编程科学计算器具有解方程的功能，比如，有一道数学题是，求 x 的解。那么在智慧计算器中输入 $x^{**2}-4*x==9$ （注意智慧计算器的次方符号是 $**$ 而等号为 $==$ ），然后点击开始按钮，就会轻松得到 x 的解，参见下图。



图 0.2： 可编程科学计算器解一元二次方程组。

为了提高学生对数学学习的兴趣，可编程科学计算器还可以显示一些有趣的数学表达式的图形。比如，在启动可编程科学计算器之后，点击绘制图形，选择绘制极坐标曲线图，然后直接运行例子，不作任何修改，就可以用极坐标绘制出爱心，花瓣和圆形。参见下图。



图 0.3： 可编程科学计算器绘制有趣的图形。

第三，用于金融工程。在金融工程中，有时候需要绘制波动率曲面（Volatility Surface），并且对波动率曲面进行分析，比如观察波动率微笑（Volatility Smile），传统的桌面软件比如 Excel 和 Matlab，虽然可以绘制三维图形，但是图形是静止的，而可编程科学计算器不但可以绘制出三维数据图形，还可以对图形进行缩放，平移和旋转。不仅如此，可编程科学计算器不仅可以手机和平板上绘制三维曲面，还可以在桌面个人电脑上绘制同样的图形，大大方便了金融分析师对数据随时随地对风险数据进行观察研究。

下图是在手机端和在桌面电脑上从不同的角度显示东南亚某大型证券交易
所的波动率曲面图象。

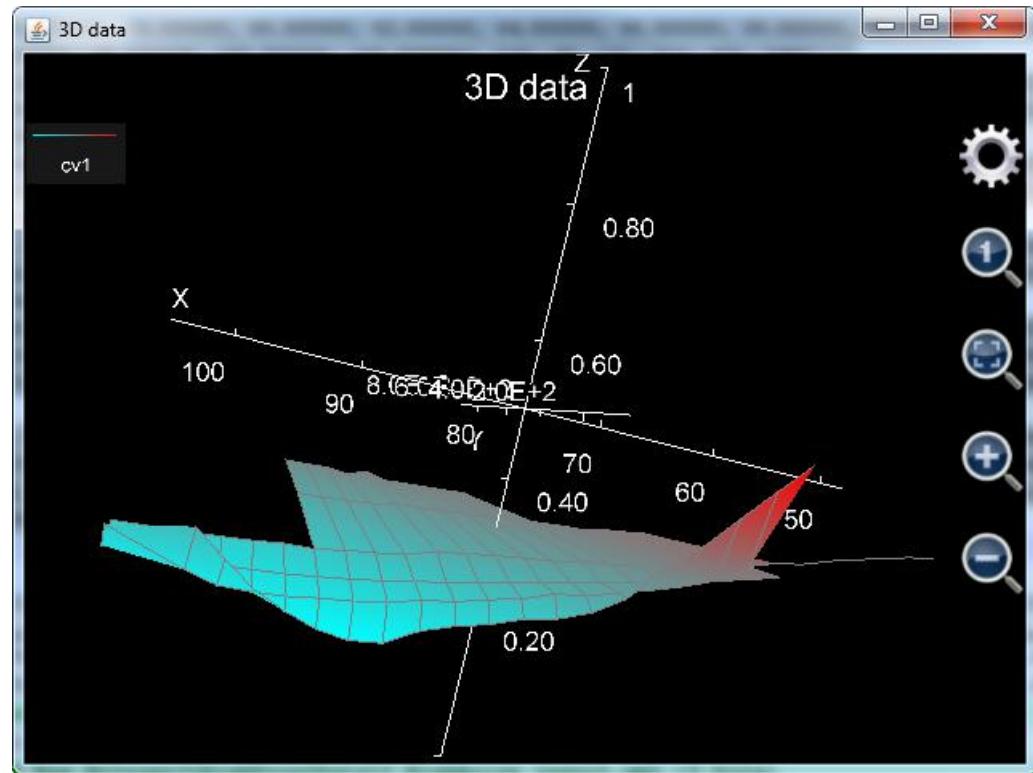


图 0.4: 在桌面电脑上显示波动率曲面。

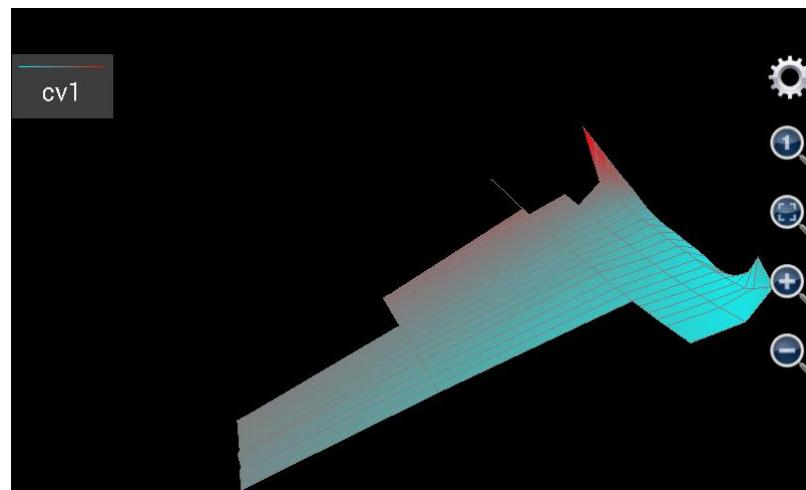


图 0.5: 在手机上从另一个角度显示同一个波动率曲面，并隐藏坐标轴。

第四：用于工业现场分析计算。比如测量工程，路桥工程以及机器的现场调试。在这些时候，往往需要运行某些特定的小程序进行计算。这些小程序，用大型软件开发工具开发费时费力，而且也没有必要，用脚本编写非常方便。但往往，在野外，或者在工作现场，携带一个笨重的笔记本电脑既不方便，也没有电源。这个时候，用手机里的可编程科学计算器提供的MFP编程语言实现现场计算，再合适不过了。

更妙的是，可编程科学计算器提供了把用户编写的MFP函数打包生成安卓APK安装文件的功能。这样一来，一位工程师可以轻松地把自己编写的计算程序发送给自己的同事，免去了重复开发的麻烦。用户甚至还可以把自己的APK发布到谷歌商店和其他任何网站，和全世界分享自己的创意。



图 0.6： 用可编程科学计算器创建自己的 MFP 应用并和他人共享。

第五：实现用脚本进行游戏开发。现在的游戏，无论是在PC端，还是在手机端，大部分都是用C/C++，C#和JAVA这些编译语言开发出来的。编译语言的速度快，对于要求很强实时性的大型的复杂的视频游戏来讲不可替代。但是这些语言开发需要专门的工具，语言编写复杂，而且基本上没有

跨平台的能力（要想跨平台就必须重新开发，至少重新编译）。还有一些非常简单的小游戏，可以用 html 或者 JAVA Script 开发，这些语言通过网络浏览器实现了跨平台，但是速度很慢，而且不太容易操作本地资源，所以也只能编写非常简单的，实时性不高的小游戏。大量对实时性要求处在这两者之间的游戏，用脚本语言开发是最合适的。

现在很多脚本语言都可以开发游戏了，比如 Python，Lua 等等。但是这些语言开发出的游戏脚本，还是无法实现跨平台。比如 Python 语言开发的 Py Game 游戏，在 PC 上的 Python 环境中运行毫无问题，但是在安卓系统上，没有一个官方的 Python 解释器，所以还无法在安卓上运行，而反过来，Lua 语言的 LuaJAVA 分支可以自由调用安卓和传统 JAVA 的 API，编写游戏脚本毫无压力。但问题是，安卓和传统 JAVA 的绘图 API 不同，所以 LuaJAVA 为传统 JAVA 平台编写的游戏脚本，无法在安卓上运行，为安卓平台编写的游戏脚本，也无法在传统 JAVA 平台上运行。

统一安卓和 PC 上脚本编程，实现一次编写，处处运行，具有重大意义。安卓平台是未来的方向，不是人人都会有 PC，但人人将来都会随身携带安卓设备。但是安卓是为用户进行简单的触摸操作设计的，不适合编程这种需要精细输入的工作。如果开发人员能够在 PC 上编写程序并调试运行（直接调试运行，而不是通过安卓模拟器），然后在安卓设备上进行一些简单的测试之后就可以发布代码或者打包好的 APK，直接让用户在安卓上使用，将大大降低安卓应用和游戏的开发难度，实现生产力质的提升。

可编程科学计算器已经朝这个方向迈出了重要的一步。从 1.7.2 版开始，MFP 语言提供了一个游戏开发引擎，该引擎实现了安卓和传统 JAVA 平台上的统一接口，开发人员编写一次游戏脚本，就可以在任何支持 JAVA 的平台上运行，还可以打包成 APK 发布到安卓游戏应用网站供全世界下载。比如以下削宝石的游戏，就是用统一的脚本在 PC 和安卓上运行：

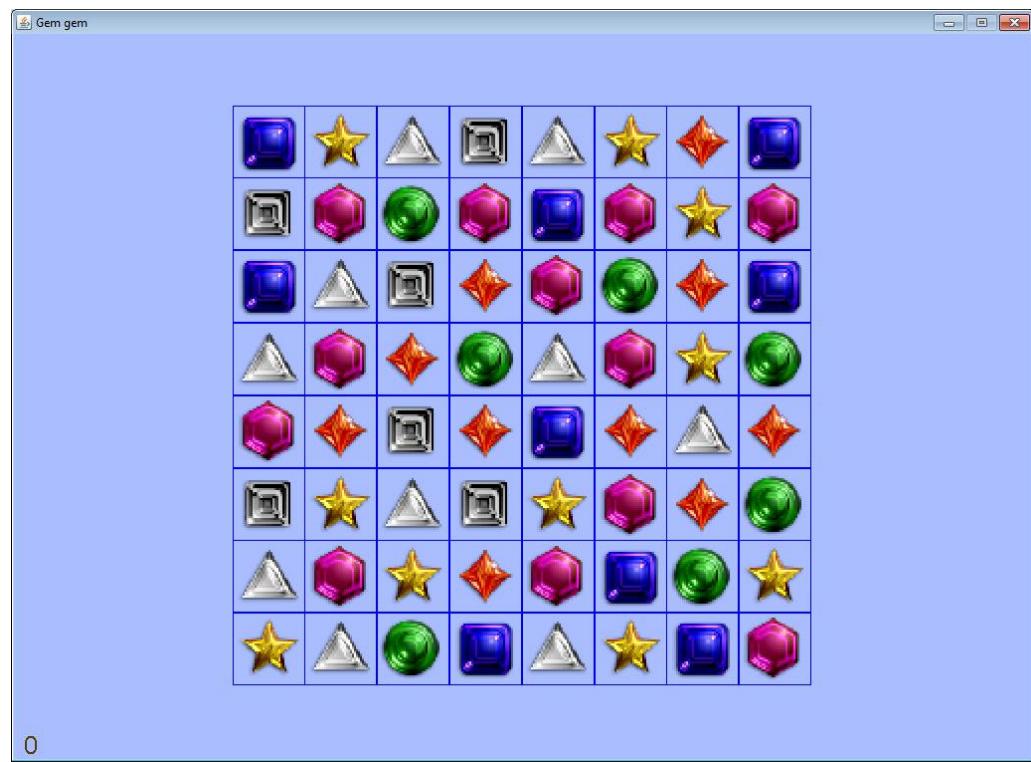


图 0.7：用安卓和传统 JAVA 平台上的统一的脚本编写削宝石游戏并在 PC 上运行。

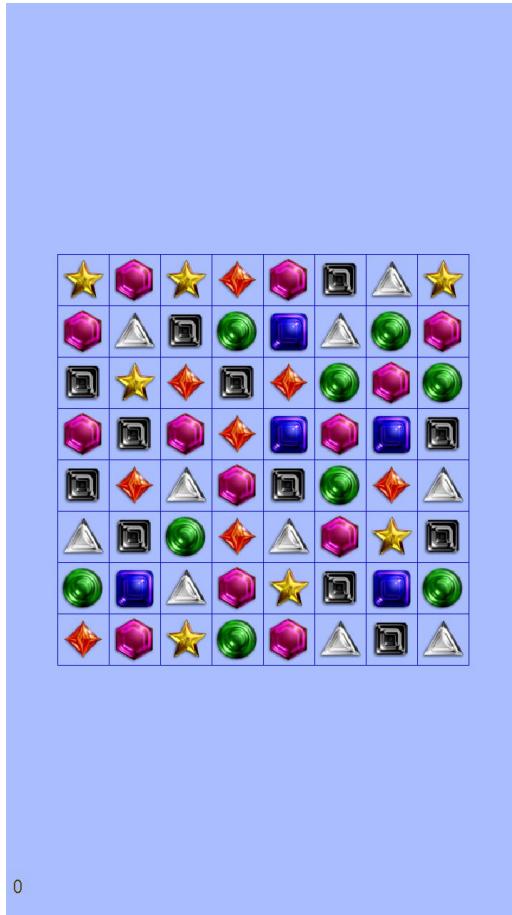


图 0.8：用安卓和传统 JAVA 平台上的统一的脚本编写削宝石游戏并在安卓上运行。

第六：未来用于智能设备和工控机的控制。可编程科学计算器现在只可以进行数学计算，支持基本的用户输入和输出，读写文件，日期以及绘制图形，但是我的目标是在不远的将来让它成为一个能对智能设备进行全面编程的脚本工具，包括更复杂的与用户交互的界面，对万维网，电子邮件和短信的访问，蓝牙 USB 传输，摄像头的控制等等。届时，可编程科学计算器将不再是一个普通的计算工具，而是安卓上的脚本开发平台，推动安卓在人们的生产和生活中得到更广泛的应用。

为了让 MFP 能够得到更加广泛的应用，我已经将 MFP 语言在 GitHub 上开源。MFP 语言代码遵循 Apache 2.0 协议，任何个人和公司均可自由地修改和使用。运行于安卓上的 MFP 语言库的代码网址位于 <https://github.com/woshiwpa/MFPAndoLib>，运行于 JAVA 虚拟机上的 MFP 语言解释器代码网址位于 <https://github.com/woshiwpa/MFPLang4JVM>，此

外，百度的的 MFP 吧（<http://tieba.baidu.com/f?kw=mfp&ie=utf-8>）是 MFP 大本营论坛，欢迎广大用户前往发问和讨论。

第1章 可编程科学计算器初级用户使用指南

可编程科学计算器是为所有需要用到数学的用户开发，毕竟，绝大部分用户都不会编程。但是，即便不会编程，也完全可以使用这个软件所带来的各种方便。

第1节 安装与启动

可编程科学计算器可以在谷歌商店以及国内各大安卓应用下载网站，包括百度手机，小米商店，联想商店，酷安网等上下载。用户登录网站，搜索可编程科学计算器，点击下载安装即可。

在安装完毕启动软件之后，用户看到如下界面：



图 1.1： 可编程科学计算器启动界面。

其中，智慧计算器模块是专门为不会编程的普通用户和数学爱好者设计的。它可以实现所有的数学计算功能，还可以绘制数学表达式的图形（包

括二维，三维和极坐标图形），还具有拍照识别数学公式和保存历史记录的功能。

命令提示符是为具有一定编程基础的科研人员以及理工科大学生设计的。它和 Matlab 很相似，用户敲入一个命令，实际上也就是用户编写的或者系统提供的一个函数，然后运行，得到结果，在屏幕上打印显示出来。

绘制图形是用于绘制制定范围的二维，三维或者极坐标图形。这个和智慧计算器中的绘制图形功能略有不同。智慧计算器中，绘制的是数学表达式的图形而不制定范围，比如 $\text{Log}(x)$ ，这样，随着用户平移缩放图形，图像的绘制范围会动态地变化，一部分点需要重新计算。这样好处是用户可以看到数学表达式在各个区间的完整面貌；坏处是由于需要重新计算绘制点，用户对图形进行缩放平移操作时会有时延和迟滞。而这个独立的绘制图形工具由于在绘制图形之前就制定了范围，所以平移缩放不会有重新计算的问题，所以图形生成之后，操作特别平滑。

计算微积分是用于计算一阶，二阶和三阶导数的表达式和值以及计算不定积分和一次，二次和三次定积分。

设置输入键盘是为了帮助用户输入特别长的函数名而设计的一项自定义键盘的功能。用户可以把常用的函数放在用户键盘上，打开智慧计算器或者命令提示符之后，不再需要冗长的输入，用户点一次就可以输入完整的函数名。

文件管理程序用于管理用户生成的 MFP 代码，在智慧计算器或者绘制图形模块中创建的图形以及编译出的应用 APK 文件。用户可以长按任何一个文件的图标运行（如果是 MFP 代码脚本文件），打开（如果是图形文件）或者安装（如果是 APK 文件）。

编写程序就是一个 MFP 代码编辑器。

设置集成了可编程科学计算器所有的设置参数，供用户调整。

在电脑上运行用于在任何一台支持 JAVA 的台式机上启动基于 JAVA 的可编程科学计算器。这将在后面详细说明。

创建 MFP 应用用于将用户编写的，或者软件提供的任意一个 MFP 函数编译打包为一个安卓应用并安装，共享或者发布。

帮助提供了基于 HTML 网页格式或者 PDF 文件格式的用户帮助手册，包括全部的函数信息和使用方法。在帮助中，用户还可以将本手册中所有的示例代码拷贝到 scripts 目录下便于阅读和运行。

第 2 节 智慧计算器的使用

智慧计算器是可编程科学计算器中的一个重要模块，用于方便大多数不会编程的用户使用，启动智慧计算器，看到的界面如下：

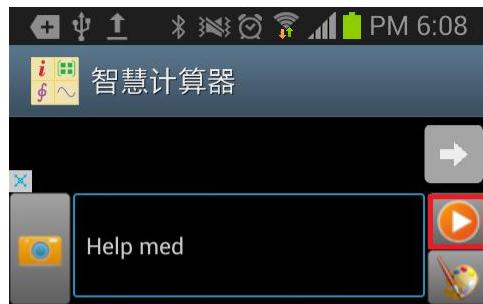


图 1.2：智慧计算器界面。

2.1 智慧计算器的输入

用户可以左右滑动输入键盘进入输入字母模式，输入数字模式和输入函数名模式，注意，在输入字母模式时，键盘提供了自带的函数名字典的功能，不用输入完整的函数名，就可以看到所有的函数名选项。

如果用户不确定某个函数的用法，可以点击输入数字模式和输入函数名模式时的救生圈图标按钮，或者输入字母时的“？”按钮，再点击开始计算按钮，获得函数的在线帮助。参见下图：



med(0...) :

函数med(...)返回任意数目参数的中位数。

and	or	xor	sum
rand	avg	gavg	havg
max	med	min	nCr
nPr	std	stdP	qsort
$\Sigma f(x)$		$\prod f(x)$	
$\int f(x)dx$			

图 1.3： 获得函数的在线帮助。

如果用户觉得可编程科学计算器自带的键盘还是不太顺手，或者想输入一些可编程科学计算器自带的键盘上没有的字符，则需要回到系统原有的键盘，这时，用户需要点击菜单键（在老的安卓设备中，菜单键位于屏幕下方，标志类似于☰；在新的安卓设备中，菜单键位于屏幕顶部应用的图标名称条的右侧，三个点的按钮），选择“激活软键盘”，则会弹出系统的输入键盘。如果用户想回到可编程科学计算器自带的键盘，则要再次点击菜单按钮，选择“隐藏软键盘”，下次用户输入，可编程科学计算器自带的键盘就会弹出。参见下图：



图 1.4：选择输入键盘。

2.2 用智慧计算器进行数值计算

在进行计算时，用户需要输入诸如 $3 + \log(4.1 / \text{avg}(1, 5, -3))$ 或者 $4*x**2 + x == 3$ 的表达式，或者一组表达式比如

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

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

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

（注意每一个表达式是一行），然后点击开始计算按钮进行计算。

使用本计算器时，用户需要注意几点，第一，本计算器中等于符号是“ $==$ ”，而“ $=$ ”是赋值符号。比如，计算满足 $x+3$ 等于 5 的 x 值，用户需要输入 $x+3==5$ 而不是 $x+3=5$ 。但要注意给一个变量赋值依然是合法的，比如 $x=7$ 。

第二点，本计算器中次方运算符是“ ** ”而不是“ \wedge ”。比如，计算满足 x 平方等于 7 的 x 值，应该输入 $x \text{ ** } 2 == 7$ 而不是 $x \wedge 2 == 7$ 。

第三点，本计算器中用户最多可以一次输入 6 个表达式。每一个表达式占据一行。智慧计算器将输入框中所有的表达式作为一个表达式组进行处理。处理过程和 MFP 语言中的 solve 程序块比较类似。但要注意的是，由于一些移动设备的硬件性能不够强大，智慧计算器仅仅依次分析每一个表达式，而不会如 MFP 语言中的 solve 程序块一样，在分析完所有表达式之后，再回过头去从头开始分析还没有能够解出的表达式。这样一来，一些 solve 程序块能够解决的方程组在智慧计算器中未见得能够解决。比如，如果用户输入

$x \text{ ** } 2 + 2*x == y$

$y + 1 == 2$

，智慧计算器仅仅能够解出 y 的值为 -1，而不能够得出 x 的值。这是因为智慧计算器先尝试解含有 x 的方程式，但是由于里面 y 值未知，所以 x 无法解出，然后智慧计算器分析第二个表达式解出 y 值。但这时智慧计算器不会再回头去解第一个方程式，所以无法得到 x 的值。但是，如果用户编写一个脚本程序，在其中嵌入 solve 程序块，然后在命令提示符中，或在拥有 JAVA 环境的个人电脑中运行基于 JAVA 的可编程科学计算器，就可以同时解出 x 和 y 的值。基于这个原因，当用户在智慧计算器中输入表达式的时候，应尽可能地将独立的表达式置为顶部，而将依赖其他变量的值以求解的表达式置于底部。

2.3 用智慧计算器进行表达式绘图

用户可以输入表达式以绘制 2 维，极坐标或者 3 维图形。表达式可以是一个等式（ $==$ ），比如 $y \text{ ** } 2 == \sin(x)*x$ 或 $t1 + t2 == t3$ 。除了等式，左侧为单一变量的赋值（ $=$ ）表达式也可以接受，比如 $a = b + 20$ 。如果一个输入的表达式既不是等式，也不是赋值式，那么可编程科学计算器将假设表达式的值等于另外一个变量。比如，如果用户输入 $2 * x + 5$ ，可编程科学计算器将自动将其转化为 $2 * x + 5 == f_x$ ，这里， f_x 是另外一个单一变量。

用户一次可以输入最多 4 个表达式用于绘图。如果这些表达式中总共包括 3 个不同的变量，计算器就会绘制 3 维图形。如果总共包括 2 个不同的变量，并且没有任何一个变量是希腊字母 α 、 β 、 γ 或者 θ ，计算器就会绘制 2 维图形。如果总共包括 2 个不同的变量，并且至少有一个变量是希腊字母 α 、 β 、 γ 或者 θ ，计算器就会绘制极坐标图形。比如，

$y=x+2*z$

$z==\sin(x)*y$

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

$4==x$

将会被绘制为 3 维图形而

$x + abs(x) - 3$

$4 + y == x$

将会是 2 维图形 ($x + abs(x) - 3$ 将会被自动转换为 $x + abs(x) - 3 == y$)，而

$\log(r)$

θ

将会被绘制成以 θ 为角度的极坐标图形。

图形绘制完成后会显示在屏幕上，用户可以注意到图形上面有若干个操作按钮，用于放大，缩小，调整 x 轴 y 轴比例为 1: 1 以及回到图形刚生成时的状态，用户还可以点击齿轮按钮对图形的绘制范围，计算采样点的数目，以及是否侦测奇异点进行设置。除了使用按钮，用户还可以用手势对图形进行拖动和缩放。参见下图：

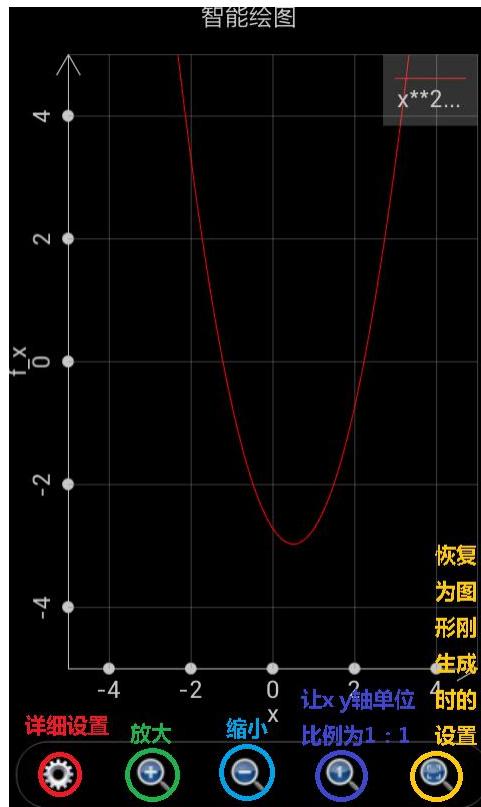


图 1.5：生成的图形。

需要注意的是智慧计算器中的绘图功能不同于本软件中的独立的绘制 2 维，极坐标或 3 维图形的功能。智慧计算器是绘制表达式而不是数值。这样一来，如果用户通过缩放或平移来改变绘制图形的范围，智慧计算器将自动重新计算在新的范围内的表达式的值。而本软件中的独立的绘制图形程序只会使用已有的数值。

此外，如果输入的表达式是一个隐函数，计算器将会尝试解出隐函数的根，然后绘制根表达式的图形。如果绘制包含两个变量的隐函数，最多 4 个根表达式将会被绘制出。如果是 3 个变量的隐函数，根据情况计算器可能会对每个变量求根，然后为每个变量绘制出最多 2 个解表达式。这样一来，一个单一的 3 变量的隐函数，可能 意味着绘制出 6 条图形，整个求解绘图过程将会花费比较长的时间。

2.4 数学公式拍照识别输入

如果用户的安卓设备拥有后置摄像头，可以使用数学公式拍照识别输入的功能快速输入表达式。注意在现阶段只支持打印体数学公式的识别。手写体数学公式的识别在我们另外一个应用，智慧拍照计算器，中支持，但仍然不完善，处于测试开发阶段。

拍照识别步骤如下：

1. 点击智慧计算器界面输入文本框左侧的拍照识别按钮；
2. 在智慧计算器显示拍照预览后，点击预览屏幕下方按钮中的一个进行拍照识别。用户可以调整绿色长方形的大小和位置以选定拍照识别的范围。用户需要注意两点：第一用户需尽可能避免拍摄出来的表达式位置不正（倾斜或者倒立）；第二，在绿色长方形的识别范围内，背景的颜色必须比表达式的颜色要浅，比如，背景颜色为白色，而表达式为黑色，此外，背景必须尽可能的单调。光投影在背景上造成背景色深浅不一是可以接受的，但是，如果背景有很明显的颜色区别，软件会误认为颜色的变化为表达式的笔画，而造成识别错误；

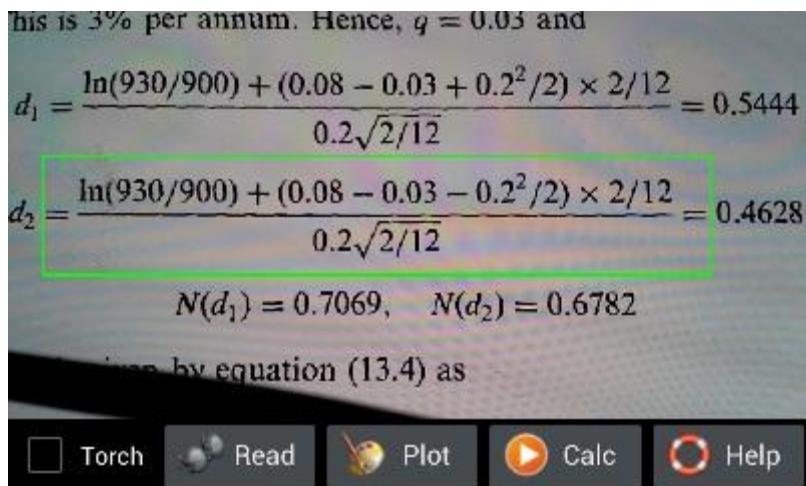


图 1.6： 拍摄数学公式准备识别。

3. 耐心等待智慧计算器对所拍得的照片预处理。识别成功后，软件会自动退回智慧计算器主屏幕对识别出的表达式进行处理。如果用户选择的是绘图，那么一个 2 维，或者 3 维，或者极坐标图会被绘制出。如果用户选择的是计算，则会给出计算结果。通常情况下，整个过程处理时间不会超过一分钟。如果表达式很简单（也就是不包

括积分和其他复杂计算），但是花费的时间很长。这通常意味着识别有误。在这种情况下，用户可以点击回退键取消正在进行的操作。如果识别已经完成，用户可以修改识别后的表达式文本重新进行计算。如果识别还未完成，用户可以重新拍照识别。如果所有的处理和计算都完成后用户仍然对结果不满意，可以发 email 给我们；

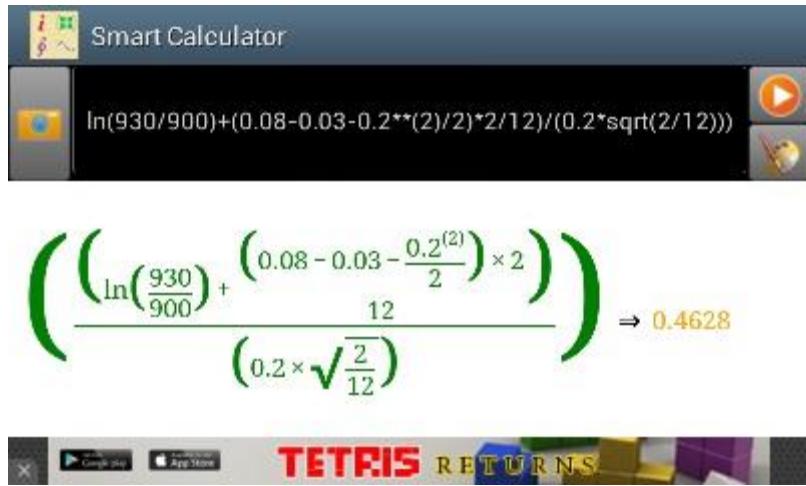


图 1.7： 数学公式识别计算结果。

拍照识别的注意事项和要求如下：

1. 智慧计算器致力于识别印刷体表达式。显然，印刷体表达式越清晰越好，识别激光打印表达式的效果肯定大大好于识别喷墨打印的效果。注意，现在我们的软件还不支持手写识别。
2. 当智慧计算器对白纸照相时，请确保背景光线不要太暗。否则，智慧计算器无法看清公式识别会失败。用户可以在预览窗口下选中“开闪光灯”选择框，将闪光灯打开。但要注意，如果闪光灯打开，白纸上的黑色墨水可能会反光造成笔画不连续，影响识别效果。
3. 当智慧计算器对白纸照相时，请确保照相机离白纸 10-30 厘米远（如果白纸是平放在桌子上）。如果太远，照出来的字太小，看不清；如果太近，过强的手机的阴影会投射到纸上，造成背景光线过暗，也会严重影响识别效果。
4. 对电脑屏幕照相比对白纸照相更复杂。这是由于电脑屏幕并非是一个完整的平面，它是由像素点矩阵构成，而且屏幕也在很快

地刷新，刷新的速率和拍照的曝光时间 接近。这样一来，用户在处理过的图像上会看到很多噪音点（实际上就是处理过的像素点），或者很多条纹（实际上就是拍摄下来的刷新线）。在这种情况下，用户 必须注意不要让摄像头离屏幕太近，并且如果一次识别不成功，可以多试几次。

拍照识别对数学表达式的支持范围如下：

1. 加减乘除；
2. 开任意次方；
3. 连加 (Σ)，连乘 (Π)；
4. 不高于 6 次的多项式；
5. 线形多元一次方程组；
6. 基本的代数函数比如三角，对数等；
7. 复数；
8. 简单的积分和求导；
9. 矩阵。

可编程科学计算器的拍照识别功能还在不断地改进中。希望用户能够把识别不准确的结果发给我们。也希望用户能够多给我们一些鼓励，一些支持，让我们把它做得更好。

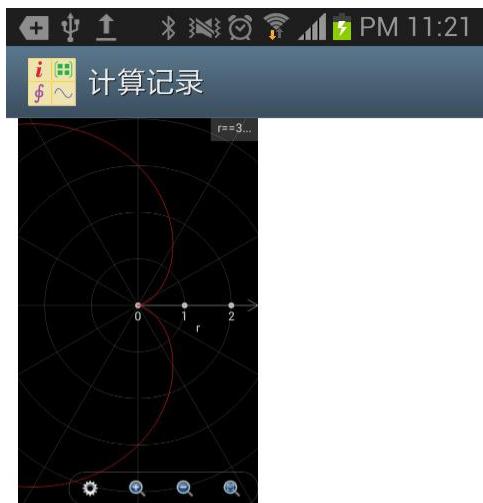
2.5 智慧计算器的输出

计算器界面的输出框向使用者显示计算结果。如果表达式的语法不正确，输出框将显示错误信息。使用者也可以点击计算器界面的输出框将上一次的计算中的任意一条表达式或者结果拷贝到计算器界面的输入框中。但要注意一点，打印输出（也就是调用 `print` 或者 `printf` 函数的输出）在计算器界面的输出框中不会被显示出来。

如果用户选择的是绘制图形，在图形绘制完成之后，退回智慧计算器界面，用户将会看到绘制出来的图形的缩略图。用户点击缩略图，会重新显示图形。

2.6 历史纪录

点击菜单按钮，选择历史纪录，则会进入历史纪录界面。历史纪录记录了用户过去一段时间在智慧计算器上所有的计算和绘图的行为。如果点击历史纪录中用户输入的表达式和结果，使用者可以将被点击的内容拷贝回计算器界面的输入框中。这样可以有效地节省输入时间。如果点击的是历史纪录中的图像缩略图，则会重新显示该图像。参见下图：



$$\begin{aligned}x^2 - 4 \times x &= 9 \\ \rightarrow \\ -9 - 4 \times x + x^2 &= 0 \\ \rightarrow \\ x &= 5.6056\end{aligned}$$

图 1.8： 显示历史纪录。

2.7 计算助手

用户可以通过点击安卓菜单按钮，选择“计算助手”菜单来进入计算助手屏幕。计算助手提供两个工具。一个是常数值，第二个是单位转换。通过常数值工具（通过底层函数 `convert_unit` 来实现），用户可以选择一个数学或者物理的常数值插入输入。通过单位转换工具，用户可以将基于一个单位的数值转换为基于另外一个单位的数值，并将转换结果或者转换表达式插入输入框。注意在启动计算助手时，如果用户在输入对话框中输入了一个合法的实数，被转换值将会被初始化为输入对话框中的数值，否则，计算助手将不会初始化被转换数值。参见下图：



图 1.9：计算助手之单位转化。

第 3 节 命令提示符的使用

命令提示符是可编程科学计算器所提供的除了计算器界面之外的另外一个有用的工具。和 Windows 的命令提示符, Unix 的终端以及 Matlab 的命令对话框类似,使用者输入命令,按运行按钮(下图中红色方框中的按钮,注意不是回车按钮),然后命令提示符将输出和返回值打印在输入行的下面。命令提示符相对于计算器界面的优势在于命令提示符可以显示所有的 print 函数的打印输出而计算器界面只能显示表达式的返回值和出错信息。

和智慧计算器一样,命令提示符也带有自己的输入键盘,用户可以左右滑动输入键盘以选择输入数字,输入字母或者输入关键字。在输入字母时,软件提供了函数名字典的功能,用户敲入函数名前面若干字母,所有符合条件的函数名选项都会列出来。命令提示符自带的输入键盘上不仅带有运行按钮,还带有回车按钮(下图紫色方框中的按钮)。和运行按钮不同,回车按钮用于调用 scanf 和 input 函数时用户结束一段输入,或者在输入多行命令时(多行命令指的是,一个命令中包含多个语句,多行命令支持

出了 function 和 endf 语句之外的所有语句，参见后面章节“在电脑上运行可编程科学计算器工具”对多行命令的说明），点击回车键换行。如果用户在命令提示符自带的输入键盘找不到想要输入的字符，则可以和在智慧计算器里面一样，通过点击菜单按钮，选择“弹出系统软键盘”菜单来激活安卓系统的输入法进行输入。

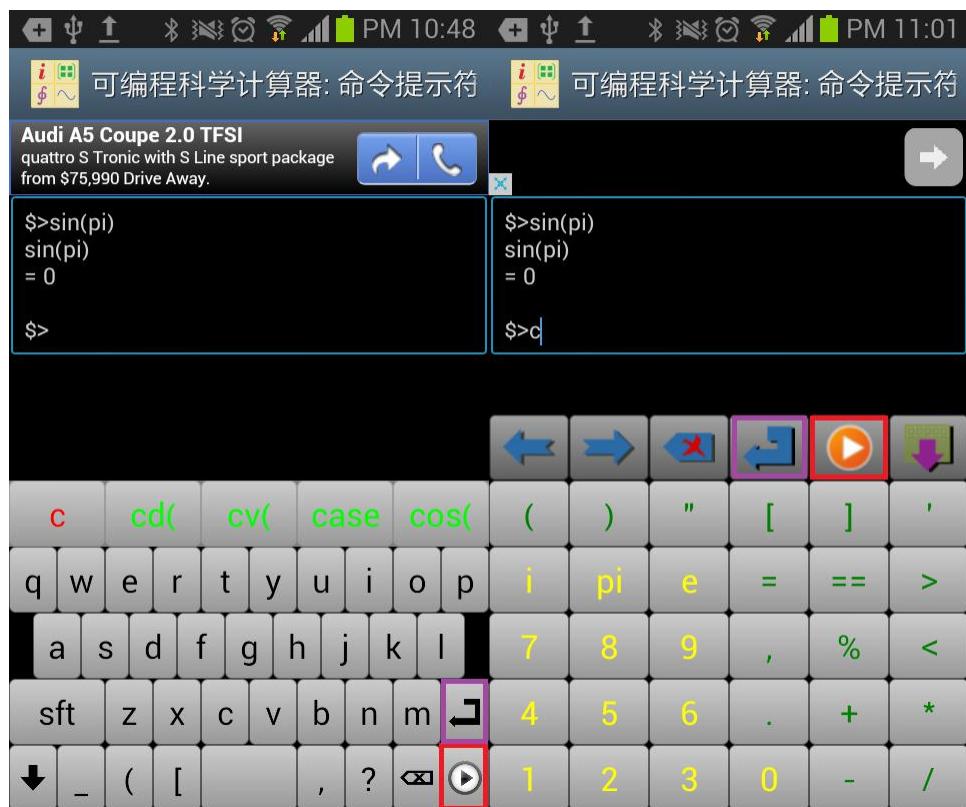


图 1.10：命令提示符的输入输出。

从 1.7.1 版开始，命令提示符工具提供了一个额外的输入键盘，该键盘保存用户的历史输入记录，大大方便了一些需要快速重复输入的用户。该键盘包含 3 列。红色字体的第一列保存了过去的输入命令，绿色字体的第二列保存了过去的输入结果，蓝色字体的第三列保存了 input 函数过去的输入数值。用户只需要点击相应按钮就可以迅速重复过去的输入。历史记录的长度取决于设置工具中的历史记录长度的设置值。



图 1.11：命令提示符的输入历史纪录

需要注意的是，基于 JAVA 的图形界面的可编程科学计算器同样也是一个命令提示符工具。在基于 JAVA 的图形界面的可编程科学计算器中，一条命令同样可以包括一行或者多行。使用者可以通过拷贝和粘贴的办法将一行或者多行拷贝到命令提示符中，或者通过键入 Shift-Enter 在一个命令中开始新的一行。和基于安卓的命令提示符工具不同，键入 Enter（回车）键（没有同时按下 Shift）会触发该命令被执行。

命令提示符支持全局变量。但要注意和 Matlab 有所不同的是，使用者需要先声明变量。比如，使用者输入命令 Variable a, b, c 以定义 a, b 和 c 三个变量，然后给这三个变量赋值（使用普通的赋值语句例如 $a = "hello world"$, $b = 4$ 以及 $c = 5 + 3.7i$ ），然后将这三个变量用于其它命令比如 $print(a)$ 以及 $exp(c)$ 。

通过点击菜单，使用者在命令提示符中可以轻松输入上一条指令或者上一次的结果。和 Matlab 类似，命令提示符有一个预先定义好的全局变量“ans”用于存储上一次计算结果（上次计算没有返回值的情况除外）。但

要注意在 MFP 编程语言中赋值语句也有返回值（就是被赋予的值）。比如语句 $c = 5 + 7.3i$ 返回 $5 + 7.3i$ 。这样一来，如果使用者在命令提示符中输入语句 $c = 5 + 7.3i$ ，“ans” 变量将被赋值为 $5 + 7.3i$ 。

和计算器界面一样，使用者已可以通过菜单启动计算助手工具并将一个常数值或者单位转换值或者单位转换表达式插入命令行。注意启动计算助手时，如果上一次的计算结果是一个合法的实数，被转换值将会被初始化为该数值，如果上一次的结果不是一个合法实数，但本次即将运行的指令是一个合法的实数，被转换值将会被初始化为该实数的数值，否则，计算助手将不会初始化被转换数值。

第 4 节 绘制图形工具

用户可以在智慧计算器中绘图，但是，智慧计算器只能对独立的数学表达式绘图，如果用户需要在指定数值范围绘制一些复杂的图形，则需要使用可编程科学计算器中独立的绘制图形工具。

绘制图形工具支持三种绘图方式：绘制二维图形，绘制极坐标图形和绘制三维图形，他们的绘图模式实际上是类似的，以绘制二维图形为例。首先用户需要对图形本身进行设定，包括图像的名字（也就是图形文件名），标题，x、y 坐标轴的名称，和是否显示网格，然后用户点击添加曲线按钮，可以在图像中添加一条曲线（图象最多显示 8 条曲线）。

定义一条曲线是使用此工具的关键，曲线的标题，颜色，曲线上每个采样点的形状，以及是否显示采样点间的连接线都易于理解，关键什么是 t ， t 的定义范围以及 $X(t)$ 和 $Y(t)$ 的定义。

我们知道，任何一个常规坐标上的不分叉二维曲线，都可以看作是一个点随时间运动的轨迹，而这个轨迹在 x 轴位置上的投影，就是 x 坐标对于 t 的函数，也就是 $X(t)$ ，同理，而这个轨迹在 y 轴位置上的投影，就是 $Y(t)$ 。

为什么用这样的方式来定义一条曲线呢？原因在于，有时候一条曲线在一个 x 的坐标对应多个 y 值，或者在同一个 y 的坐标对应于多个 x 的值。比如，一个圆形。用智慧计算器中绘图功能，这样的曲线有时无法用单个的表达式描述，而在这里，则可以用 t 分别描述 x 和 y。比如，圆形的半径为 2.5，圆心位置为 $(1.3, -1.7)$ ，那么，设定 $X(t)$ 为 $2.5\cos(t) - 1.3$ ，设

定 $Y(t)$ 为 $2.5 \sin(t) + 1.7$ ，让 t 从 0 到 2π 变化，则这个圆可以精确地绘制出来。

用上述办法，可以绘制出各种更为复杂的图形。但是也可以用它绘制出常规的简单图形，办法就是将 $X(t)$ 设置为 t （也就是用 t 来代替 x ）， $Y(t)$ 设置为一个关于 t 的表达式，比如为了绘制最普通的抛物线，可以将 $Y(t)$ 设置为 t^{**2} （也就是 t 平方），然后任选 t 的变化范围，比如让 t 从 -5 到 5 变化（也可以取其他变化范围），就绘制出了抛物线 y 等于 x 平方 x 从 -5 到 5 的图像。



图 1.12：用独立的绘制图形工具绘制二维图形。

以下是从用二维图形工具绘制图形的一些其它例子。比如，想要绘制从 $(3, 5)$ 到 $(3, 15)$ 的线段，可以设定 t 从 5 到 15，步长设为自动， $X(t)$ 设为 3， $Y(t)$ 设为 t ，即可。

也可以用此工具绘制包含奇异点的图形，比如想绘制 $y = \tan(x)$ ，可以设定 t 的范围从 -2π 到 2π ，间隔设置为自动（空出不填写就是自动）， $X(t)$ 设置为 t ， $Y(t)$ 设置为 $\tan(t)$ ，则绘制出的图形如下左。

但是如果 t 的间隔设置不是自动（也就是用户填入指定的间隔，比如 0.1），可编程科学计算器将不会自动侦测奇异点，绘制出来的图形如下图右。

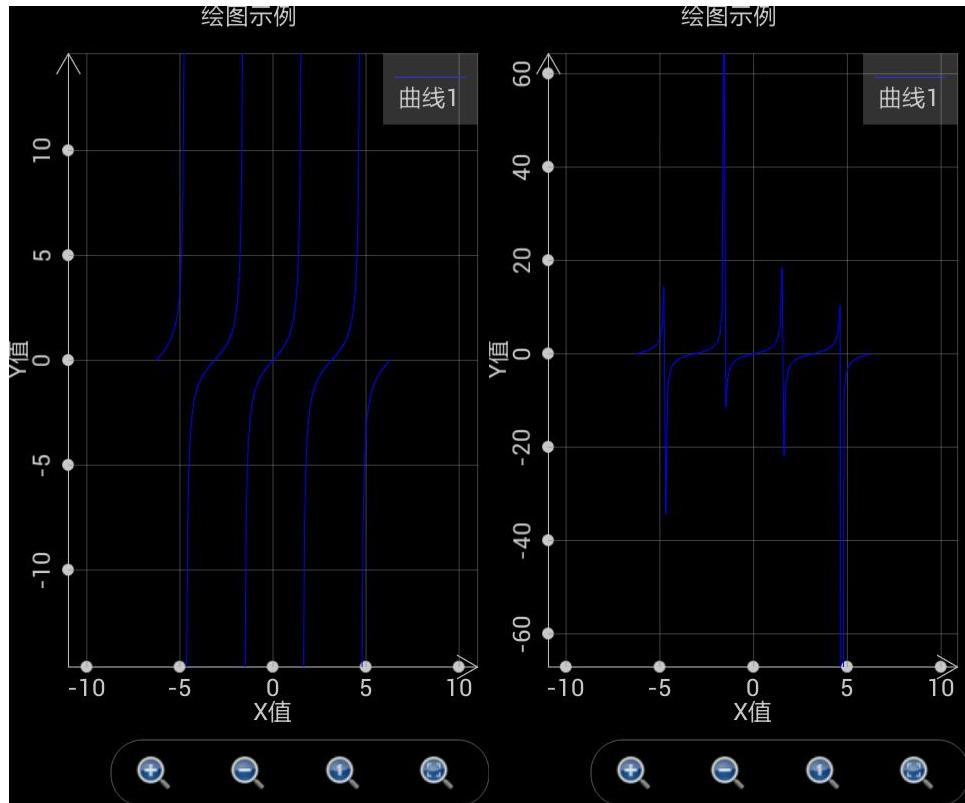


图 1.13： T 的间隔设置为自动时，软件能够自动侦测奇异点。

绘制极坐标图形和绘制二维图形比较类似，但要注意这时就不再是设置 $x(t)$ 和 $y(t)$ 而是设置 $r(t)$ 和 $\theta(t)$ 。这里 r 是轨迹点到极坐标的距离， θ 是轨迹点的幅角。

以下是绘制极坐标图形的一些例子。当 t 从 0 到 2π ，间隔为自动， $r(t)$ 为 $\cos(t)$ ， $\theta(t)$ 为 t 时，绘制出来的是一个圆形。参见下图的绿色曲线。

当 t 从 -2π 到 2π ，间隔为自动， $r(t)$ 为 $2\sin(4t)$ ， $\theta(t)$ 为 t 时，绘制出来的是一个花瓣形。参见下图的蓝色曲线。

当 t 从 -1.5π 到 1.5π ，间隔为自动， $r(t)$ 为 t ， $\theta(t)$ 为 t 时，绘制出来的是一个心形。参见下图的紫红色曲线。

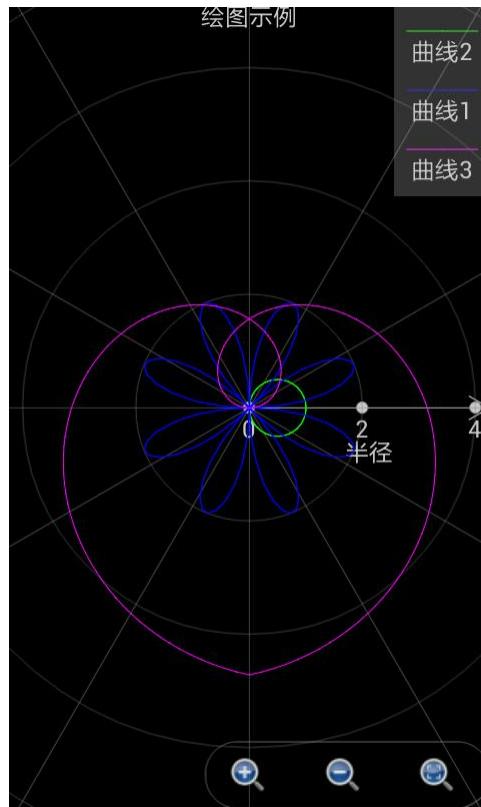


图 1.14： 极坐标绘图工具绘制出各种图形。

绘制三维图形则略为复杂，设置界面参见下图。图像整体的设置和曲线的标题都易于理解。仅绘制网格复选框指的是仅仅绘制曲面上个网格而不加以填充。最大值和颜色值指的是当 Z 的值大于等于该最大值时，曲面正面的颜色和反面的颜色。最小值和颜色指的是当 Z 的值小于等于该最小值时，曲面正面的颜色和反面的颜色。位于最大值和最小值之间的部分，其颜色处于最大值颜色和最小值颜色之间的过渡。

和二维图形不同，这里不存在 t ，而是存在 u 和 v ，变量 X , Y 和 Z 都是 u 和 v 的函数。之所以设置两个而不是一个内部变量是因为大部分情况下我们需要画出的是曲面而不仅仅是一条曲线， u 和 v 可以理解为轨迹点在曲面经度和纬度上的变化。

由于存在两个内部变量，这时 X , Y 和 Z 均为基于内部变量 u 和 v 的表达式。通过设定 $X(u, v)$, $Y(u, v)$ 和 $Z(u, v)$ ，我们可以绘制出各种有趣的曲面。我们依然可以绘制出三维曲线，办法是将 X , Y 和 Z 设置为仅仅基于 u 的表达式，和 v 的变化无关。



图 1.15：用独立的绘图工具绘制出三维图形。

以下是绘制三维图像的例子。如果想绘制一个在 x 等于 10 的平面，可以设定 u 从 0 到 10，间隔为自动， v 从 0 到 10，间隔为自动， x 设置为 10， y 设置为 v ， z 设置为 u ，绘制出来的图形如下图：

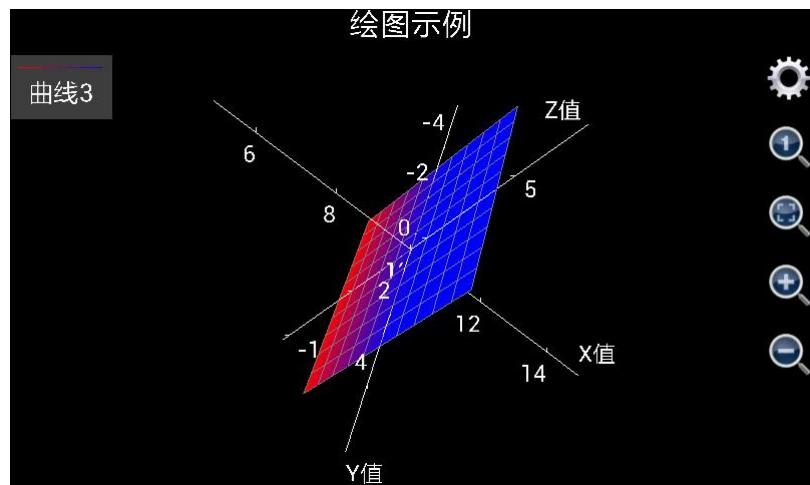


图 1.16：用独立的三维绘图工具绘制出 x 等于 10 的平面。

如果想绘制出从三维空间中一个点到另外一个点的线段，比如从 $(x, y, z)=(1, 5, 6)$ 到 $(x, y, z)=(10, 3, 9)$ ，可以将 u 设置为从1到10，间隔为自动，因为 x ， y 和 z 将和 v 无关， v 的值可以任意设置。但要注意，由于图像中所有需要计算的点的数目等于 u 的变化步数乘以 v 的变化步数，如果想要计算尽可能地快，需要计算的点的数目必需尽可能地少，由于 u 的变化间隔已经设置为自动，这样一来，用户需要使 v 的变化步数尽可能的少，所以，可以设置 v 从0到1变化间隔为1（也就是 v 的变化步数为1步）。然后将 x 设为 u ， y 设为 $(u-1)/(10-1)*(3-5)+5$ 以保证 x 和 y 线性相关， z 设为 $(u-1)/(10-1)*(9-6)+6$ 以保证 x 和 z 线性相关，还要注意，为了看清线段的颜色，必须选择“仅绘制网格”复选框，否则绘制出的线段将是和坐标轴接近的灰色。绘制出来的图形如下图：

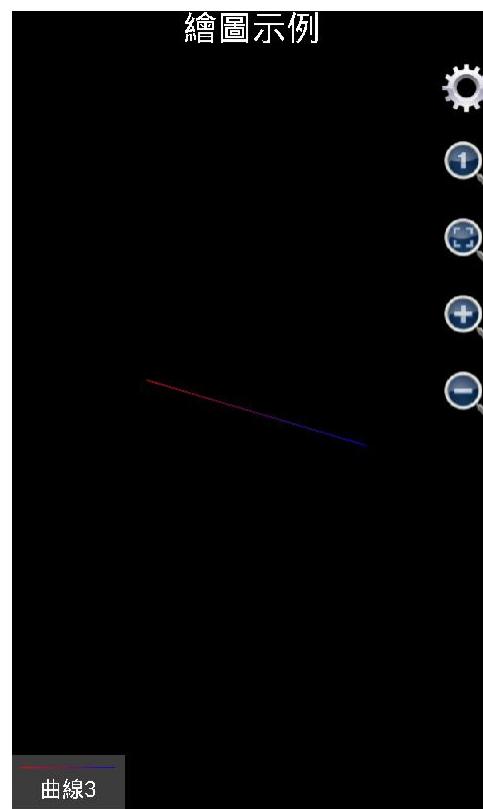


图 1.17：用独立的三维绘图工具绘制出线段。

需要注意的是，在1.6.7版之前，可编程科学计算器绘制三维图像时是自动显示坐标轴的，如果用户觉得坐标轴在图形中显得特别碍眼，可以点击齿轮按钮，选中“不显示轴和标题”复选框，把坐标轴和标题隐藏起来。但是，从1.6.7版开始，可编程科学计算器绘制三维图像时是自动隐藏坐

标轴的，如果用户想显示坐标轴，可以点击齿轮按钮，不选中“不显示轴”复选框，然后点击确定，则坐标轴将会被显示出来。使用类似的办法，用户还可以显示或者隐藏标题。



图 1.18：三维图形显示或者隐藏坐标轴和标题。

如果是想绘制出更复杂的图形，比如球形，可以想象 u 是球形的经度，而 v 是球形的纬度。那么应该 x 设置为 $3\cos(v)\cos(u)$ ，这里 3 是球形的半径， y 设置为 $3\cos(v)\sin(u)$ ， z 设置为 $3\sin(v)$ ， u 的取值为 0 到 2π ，间隔自动， v 的取值为 $-\pi/2$ 到 $\pi/2$ ，间隔自动，绘制出来的图形如下图左（由 1.6.6.51 版可编程科学计算器绘制出）。

如果用户使用的是 1.6.6 或者以前版本的可编程科学计算器，就会发现，似乎绘制出的不是一个球，而是一个椭球体。这是因为， x ， y 和 z 轴的坐标单位长度不一致。用户可以点击那个放大镜中有一个小 1 的按钮，将 x ， y 和 z 轴的坐标单位长度设为 1: 1: 1，得到一个标准的球体。参见下图右。如果用户使用的是 1.6.7 版或者更高版本的可编程科学计算器，绘制三维图形的时候， x 、 y 和 z 的比例已经被自动调整为 1: 1: 1，所以，用

1.6.7 或者更高版本的可编程科学计算器绘制上述图像，将会直接得到下图右的效果。

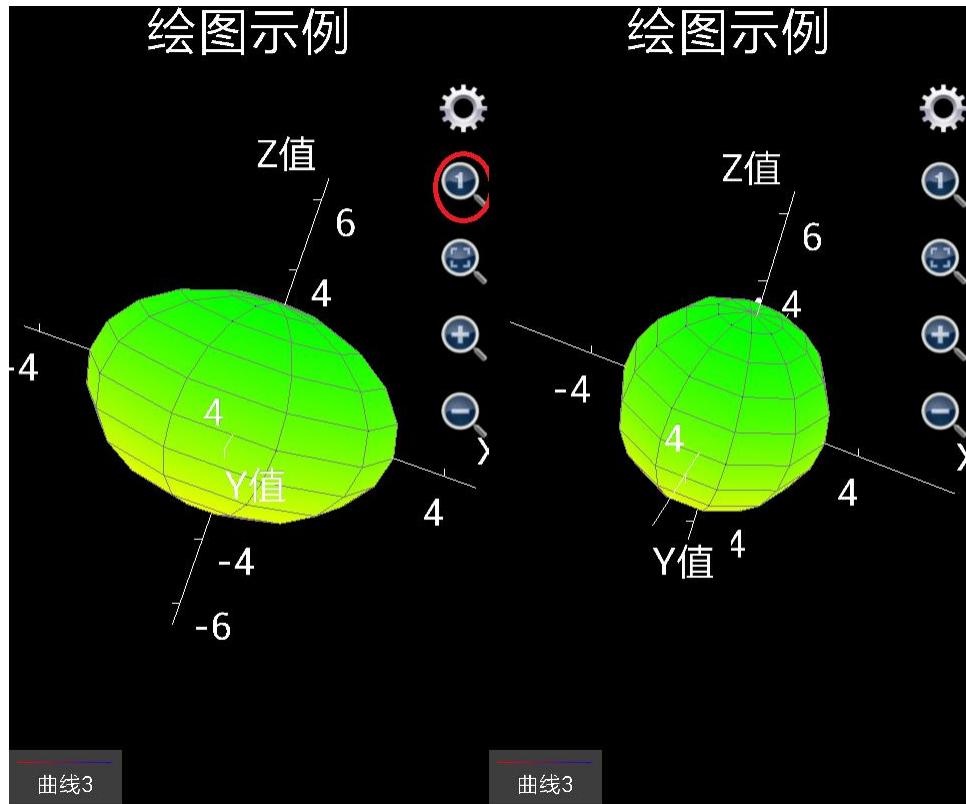


图 1.19：用三维图形工具绘制出球形。

如果用户想绘制出一个类似地球的形状，南北极是白色，中间是绿色，则要绘制出两个半球，第一个半球，也就是第一条曲面，标题设置为“北半球”，不选择“仅绘制网格”复选框，最大值和颜色选择为自动（也就是不填任何内容），白色，白色（表示北极），最小值和颜色选择为最大值和颜色选择为自动，绿色，绿色（表示位于赤道的绿色森林）。x 设置为 $3*\cos(v)*\cos(u)$ ，这里 3 是球形的半径，y 设置为 $3*\cos(v)*\sin(u)$ ，z 设置为 $3*\sin(v)$ ，u 的取值为 0 到 2π ，间隔自动，v 的取值为 0 到 $\pi/2$ ，间隔自动。

第二个半球，也就是第二条曲面，标题设置为“南半球”，不选择“仅绘制网格”复选框，最小值和颜色选择为自动，白色，白色（表示南极），最大值和颜色选择为最大值和颜色选择为自动，绿色，绿色（表示位于赤道的绿色森林）。x 设置为 $3*\cos(v)*\cos(u)$ ，y 设置为 $3*$

$\cos(v) * \sin(u)$, z 设置为 $3 * \sin(v)$, u 的取值为 0 到 $2 * \pi$, 间隔自动, v 的取值为 $-\pi/2$ 到 0, 间隔自动。

最后绘制出来的图形如下:

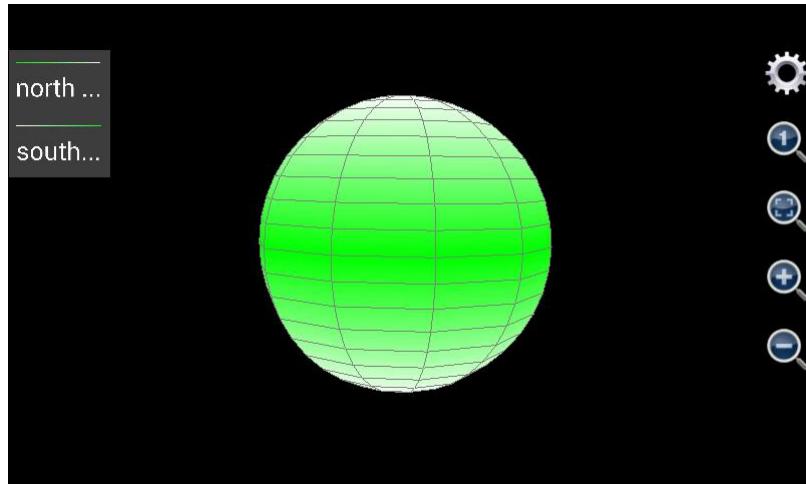


图 1.20: 用三维图形工具绘制出类似地球形状。

如果是想绘制圆柱体, 则和绘制球体的办法略有不同。球体用一个表面即可绘制出来, 而圆柱体有三个表面——上表面, 下表面和侧表面。不妨假设待绘制的圆柱体半径为 5, 圆柱体下表面的高度为 0, 上表面的高度为 20, 则对于柱体的下表面 (是一个实心圆), 设定 u 为幅角, 变化范围从 0 到 2 步长为 0.05 (表示从 0 到 $2 * \pi$ 变化, 每一步为 $0.05 * \pi$), v 为半径, 变化范围从 0 到 5 步长为 5, x 应该设置为 $v * \cos(u * \pi)$, y 应该设置为 $v * \sin(u * \pi)$, z 是高度, 等于 0, 最大值和颜色分别为自动, 红色, 红色, 最小值和颜色也是自动, 红色, 红色 (上表面的 z 方向厚度为 0, 所以最大值颜色和最小值颜色应该保持一致)。

对于柱体的上表面 (也是一个实心圆), 设定 u 为幅角, 变化范围从 0 到 2 步长为 0.05 (表示从 0 到 $2 * \pi$ 变化, 每一步为 $0.05 * \pi$), v 为半径, 变化范围从 0 到 5 步长为 5, x 应该设置为 $v * \cos(u * \pi)$, y 应该设置为 $v * \sin(u * \pi)$, z 是高度, 等于 20, 最大值和颜色分别为自动, 蓝色, 蓝色, 最小值和颜色也是自动, 蓝色, 蓝色 (下表面的 z 方向厚度为 0, 所以最大值颜色和最小值颜色应该保持一致)。

对于柱体侧表面, u 应该设置为柱体横截面的幅角, 变化范围从 0 到 2 步长为 0.05 (表示从 0 到 $2 * \pi$ 变化, 每一步为 $0.05 * \pi$), v 为柱体表面每一点

的垂直高度，所以 v 的变化范围从 0 到 20 步长为 20，由于柱体的侧表面上每一点的半径均为 5， x 的应该设置为 $5*\cos(u*pi)$ ， y 应该设置为 $5*\sin(u*pi)$ ， z 应该设置为 v 。为了和上下表面的颜色匹配，侧表面的最大值和颜色分别为自动，蓝色，蓝色（和上表面保持一致），最小值和颜色分别为自动，红色，红色（和下表面保持一致），则最后绘制出的图形如下图：

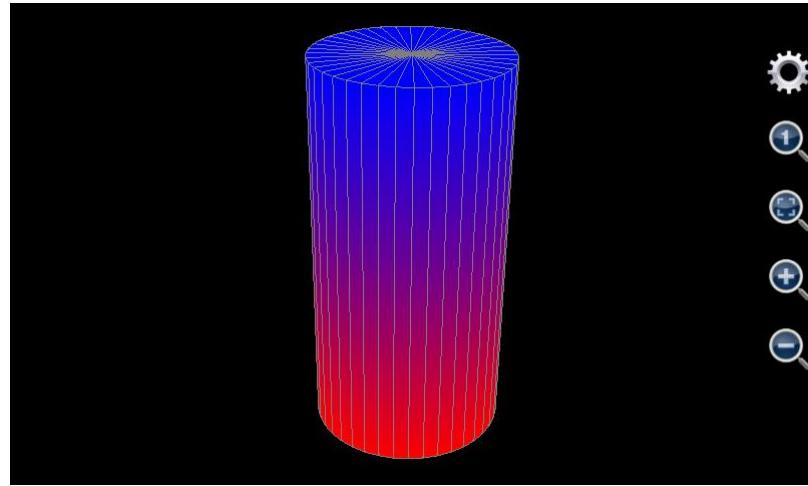


图 1.21：用三维图形工具绘制出圆柱体。

用三维绘图工具也可以绘制出圆锥体。和圆柱体有所不同的是，圆锥体只有两个表面——下表面和侧表面。下表面绘制方法和圆柱体完全一样，对于侧表面，其上每一点的半径随着高度的增加而减小，所以，其半径不再是一个常数，而是一个和高度线形相关的变量。

不妨假设待绘制的圆锥体的下表面的半径为 5，下表面高度为 0，圆锥体的高度为 20，下表面由于和上述圆柱体的下表面完全一样，绘制方法参见上面圆柱体下表面的绘制方法。对于侧表面， u 应该设置为锥体横截面的幅角，变化范围从 0 到 2 步长为 0.05（表示从 0 到 $2*pi$ 变化，每一步为 $0.05*pi$ ）， v 为锥体表面每一点的垂直高度，所以 v 的变化范围从 0 到 20 步长为 20，由于锥体的侧表面上每一点的半径为和高度反向线形相关，所以半径表达式应该写为 $5*(20-v)/20$ ， x 的应该设置为 $5*(20-v)/20*\cos(u*pi)$ ， y 应该设置为 $5*(20-v)/20*\sin(u*pi)$ ， z 应该设置为 v 。侧表面的最大值和颜色分别为自动，蓝色，蓝色，为了和下表面的颜色匹配，最小值和颜色分别为自动，红色，红色，则最后绘制出的图形如下图：

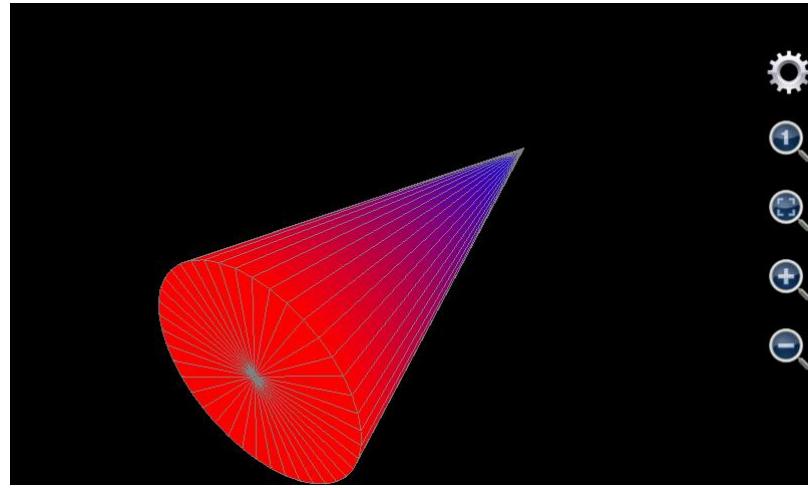


图 1.22：用三维图形工具绘制出圆锥体。

用这个三维绘图工具还可以生成其他一些有趣的形状。比如 u 设置为 0 到 2π , v 设置为 0 到 10, x 设置为 $v \cos(u)$, y 设置为 $v \sin(u)$, z 设置为 $6 \cos(v) \exp(-v/10)$, 我们得到如下形状:

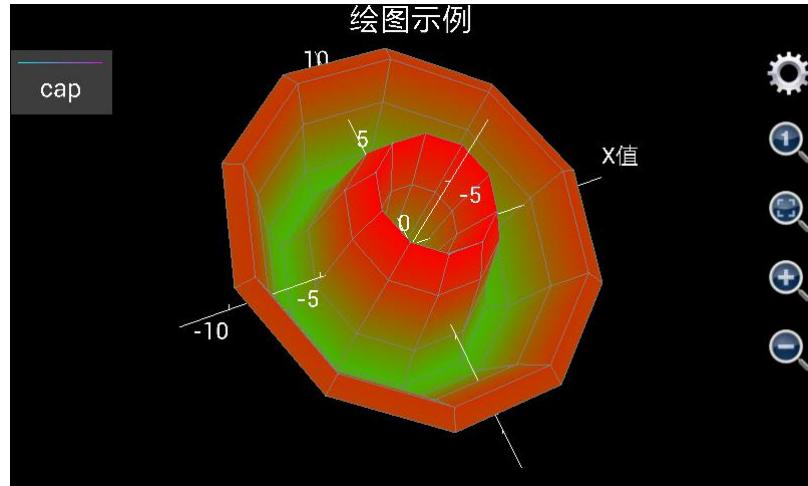


图 1.23：用三维图形工具绘制出类似花朵的形状。

再比如, 设置 v 从 0 到 10 步长 0.1, 设置 x 为 $v \cos(v)$, y 为 $v \sin(v)$, z 为 v , u 设置为其他任何值 (如上所述, 为了加快计算速度, 最好设置 v 从 0 到 1 步长为 1), 并选中“仅绘制网格”复选框, 我们得到如下螺旋曲线:

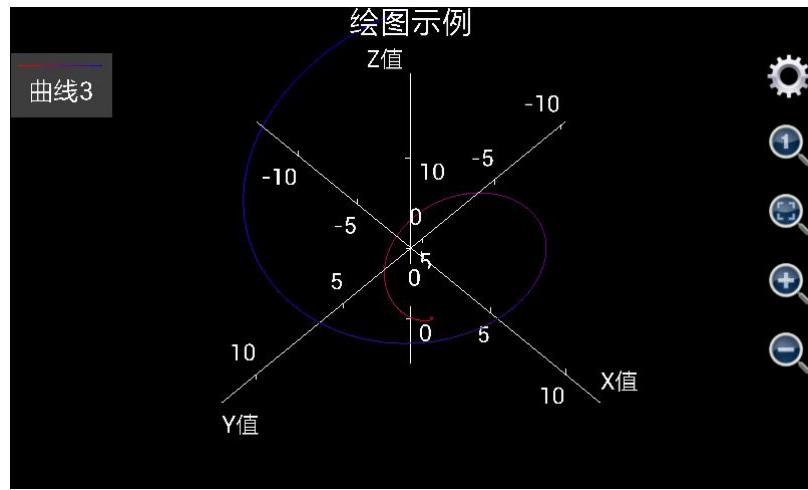


图 1.24：用三维图形工具绘制出螺旋线。

最后，本节列出一个使用三维图形绘图工具绘制出复杂图形的例子。众所周知，中国最大的城市上海的地标建筑为东方明珠电视塔，其照片如下：



图 1.25：上海地标东方明珠电视塔。

那么，有没有可能利用三维绘图工具在手机中画出东方明珠电视塔呢？答案是肯定的。

为了绘制出东方明珠电视塔，首先需要分析出东方明珠电视塔由哪些几何图形构成（从底部到顶部）：

最下端是电视塔底部三根倾斜的支撑柱。对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为红色（red），最大值颜色为黄色（yellow），最小值和最大值不设置（也就是设置为软件缺省值）。假设柱体的半径为 3，倾斜度为 45 度，高度为 20，则三根柱体的底部中心坐标在水平面上构成了一个正三角形，其坐标分别为 $(-20\sqrt{3}/2, -10, 0)$, $(0, 20, 0)$, $(20\sqrt{3}/2, -10, 0)$ ，设置 u 为幅角， v 为半径， u 从 0 到 8（表示从 0 到 8π ），步长为 0.25（表示 $1/4\pi$ ）， v 从 0 到 20，步长为 20，则 x 的表达式为 $\text{iff}(u \leq 2, 3\cos(u\pi) - (20-v)\sqrt{3}/2, \text{and}(u \geq 3, u \leq 5), 3\cos(u\pi), u \geq 6, 3\cos(u\pi) + (20-v)\sqrt{3}/2, \text{Nan})$ ， y 的表达式为 $\text{iff}(u \leq 2, 3\sin(u\pi) + (20-v)/2, \text{and}(u \geq 3, u \leq 5), 3\sin(u\pi) - (20-v)\sqrt{3}/2, u \geq 6, 3\sin(u\pi) + (20-v)/2, \text{Nan})$ ，由于海拔为随 v 变化，所以 z 的表达式就是 v 。

这里需要注意几点，首先，三根倾斜的支撑柱实际上包括 3 个曲面，但是这里用一组表达式绘制出来。为什么这样做而不是用一组表达式绘制一个曲面呢？原因在于，上海东方明珠电视塔所需要绘制的曲面个数超过 8 个，而使用绘制图形工具最多只能使用 8 组表达式，所以，必须使用一组表达式绘制出多个曲面。

使用一组表达式绘制多个曲面，其窍门在于使用了 iff （也就是如果）函数。这里， u 的值是从 0 到 8（也就是幅角变化范围从 0 到 8π ），但是一个柱体横截面的幅角变化范围是从 0 到 2π ，这样，我们通过调用 iff 语句，在 u 从 0 到 2（也就是幅角从 0 到 2π ）变化时绘制第一个柱体，在 u 从 3 到 5（也就是幅角从 3π 到 5π ）变化时绘制第二个柱体，在 u 从 6 到 8（也就是幅角从 6π 到 8π ）变化时第三个柱体， u 在不同的范围，通过 iff 函数所提供的 x 和 y 的表达式不同。

但是，为什么不设置 u 的值从 0 到 6 变化？换句话说，为什么需要引入 u 从 2 到 3 和从 5 到 6 时 x 和 y 的值为 Nan ？这是因为，虽然可以同时绘制 3 个柱体，但是必须保证这三个柱体并不相连，引入 u 从 2 到 3 和从 5 到 6 时 x 和 y 的值为 Nan ，就是起到断开这三个曲面的连接的作用，毕竟， Nan 是无法被绘制出来的。

至于 x 和 y 的表达式中的 $(20-v)*sqrt(3)/2$ 和 $(20-v)/2$ 部分，则是用来实现柱体的倾斜效果，也就是说，随着 v 的增大，x 和 y 的坐标出现漂移。

绘制了三根支撑柱体之后，注意到三根支撑柱体之间还有一根立柱起到上下通路的作用。绘制直立柱体很简单，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为绿色（green），最大值颜色为黄色（yellow），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，可以设置 u 从 -1 到 1，步长为 0.25，v 从 0 到 20，步长为 20，柱体半径为 2，则 x 的表达式为 $\cos(u*\pi)*2$ ，y 的表达式为 $\sin(u*\pi)*2$ ，由于海拔为 v，所以 z 的表达式为 v；

然后就是一个很大的球体，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为红色（red），最大值颜色为青色（cyan），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，假设其半径为 10，球心位于 (0, 0, 20)，u 从 -pi 到 pi 步长为 $\pi/10$ ，v 从 $-\pi/2$ 到 $\pi/2$ 步长为 $\pi/10$ ，x 的表达式为 $10*\cos(u)*\cos(v)$ ，y 的表达式为 $10*\sin(u)*\cos(v)$ ，z 的表达式为 $10*\sin(v)+20$ ；

球体之上是三根垂直的立柱，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为绿色（green），最大值颜色为兰色（blue），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，假设立柱的半径为 1.5，三个柱心的 x 和 y 的坐标分别为 $(-2, 2/\sqrt{3})$ ， $(0, 4/\sqrt{3})$ 和 $(2, 2/\sqrt{3})$ ，和绘制三根斜柱一样，设置 u 为幅角，v 为半径，u 从 0 到 8（表示从 0 到 $8*\pi$ ），步长为 0.25（表示 $1/4*\pi$ ），v 从 20 到 70，步长为 50，则 x 的表达式为
$$\text{iff}(u \leq 2, 1.5*\cos(u*\pi)-2, \text{and}(u >= 3, u \leq 5), 1.5*\cos(u*\pi), u >= 6, 1.5*\cos(u*\pi)+2, \text{Nan}),$$
 y 的表达式为
$$\text{iff}(u \leq 2, 1.5*\sin(u*\pi)+2/\sqrt{3}, \text{and}(u >= 3, u \leq 5), 1.5*\sin(u*\pi)-4/\sqrt{3}, u >= 6, 1.5*\sin(u*\pi)+2/\sqrt{3}, \text{Nan}),$$
 z 的表达式为 v。

然后绘制立柱上部的较小的球体，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为紫红色（magenta），最大值颜色为白色（white），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，设置球心为 $(0, 0, 70)$ ，半径为 6，u 从 -pi 到 pi 步长为 $\pi/10$ ，v 从 $-\pi/2$ 到 $\pi/2$ 步长为 $\pi/10$ ，x 的表达式为

$6*\cos(u)*\cos(v)$ ， y 的表达式为 $6*\sin(u)*\cos(v)$ ， z 的表达式为 $6*\sin(v)+70$ ；

小球上面还有一根立柱，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为黄色（yellow），最大值颜色为绿色（green），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，假设高度为 15，柱心位于 $(0, 0)$ ，半径为 1.5，设置 u 为幅角， v 为半径， u 从 0 到 2（表示从 0 到 $2*\pi$ ），步长为 0.25（表示 $1/4*\pi$ ）， v 从 70 到 85，步长为 15，则 x 的表达式为 $\cos(u*\pi)*1.5$ ， y 的表达式为 $\sin(u*\pi)*1.5$ ， z 的表达式为 v ；

柱上面还有一个更小的球，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为红色（red），最大值颜色为青色（cyan），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，设置球心位于 $(0, 0, 85)$ ，半径为 2，设置 u 从 $-\pi$ 到 π 步长为 $\pi/10$ ， v 从 $-\pi/2$ 到 $\pi/2$ 步长为 $\pi/10$ ， x 的表达式为 $2*\cos(u)*\cos(v)$ ， y 的表达式为 $2*\sin(u)*\cos(v)$ ， z 的表达式为 $2*\sin(v)+85$ ；

最后是圆锥形的天线，对于标题和颜色，设置曲面标题为空，不选择“仅绘制网格”，设置最小值颜色为红色（red），最大值颜色为浅灰色（ltgray），最小值和最大值不设置（也就是设置为软件缺省值）。对于曲面本身，设置椎底半径为 0.5，椎高度为 30，椎底中心坐标为 $(0, 0, 85)$ ，设置 u 从 $-\pi$ 到 π 步长为 $\pi/5$ ， v 从 85 到 115 步长为 10，设置椎顶部最小半径为 0.2 倍的锥体底部最大半径，则 x 的表达式为 $0.5*\max(0.2, (115-v)/30)*\cos(u*\pi)$ ， y 的表达式为 $0.5*\max(0.2, (115-v)/30)*\sin(u*\pi)$ ， z 的表达式为 v 。

以上设置相当繁复，用户在手机上输入比较困难。考虑到这一点，在可编程科学计算器 1.6.7 及其以上版本中，用户进入“绘制三维图像”工具后，点击安卓系统的菜单按钮，选择“填充示例”菜单，所有上述输入将自动被填入，用户只需再点击观看按钮就可以开始绘制图形。

由于东方明珠电视塔是一个很复杂的图形，如果手机性能不好，绘制这个图形可能需要 2-3 分钟（手机性能好可能会很快），所以需要耐心等待，如果用户使用的是可编程科学计算器 1.6.6 版及其以下版本，最后绘制出来的图形参见下图左边部分。

显然，上图左边部分的图形长宽高不成比例，所以，如果使用的是可编程科学计算器 1.6.6 版及其以下版本，用户需要点击红色圆圈中的放大镜里面有一个小 1 的按钮，将长宽高调整为 1: 1: 1，然后，用户需要点击绿色圆圈中的齿轮按钮，选择隐藏坐标轴和标题，最后调整过的图形参见下图右边部分。

如果用户使用的是可编程科学计算器 1.6.7 版及其以上版本，则不需要做上述调整，绘制出的图形的长宽高单位比例自动设置为 1: 1: 1，并且坐标轴自动隐藏（标题会自动显示），图像绘制出来就达到下图右边部分的效果。对比东方明珠的照片，可以看到绘制出来的图形非常相似和完美。

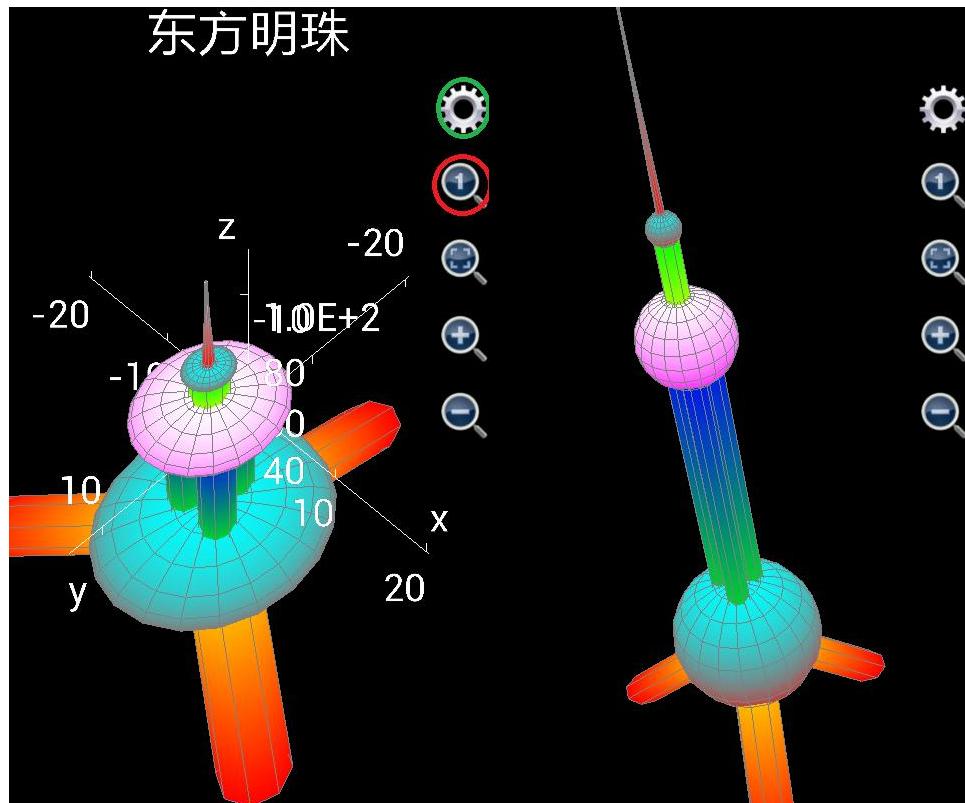


图 1.26：用绘制图形工具绘制上海东方明珠电视塔。

当然，无论如何，在手机上输入上述 8 组复杂的表达式是一件非常不容易的事情，这个时候，调用 MFP 的 `plot3d` 或者 `plot_3d_surfaces` 函数直接编程的威力就显现出来。如果用户会使用编程功能，可以将上述输入在一个自定义函数中用一个 MFP 调用语句实现，将该自定义函数保存在一个代码

的文本文件中，任何时候，用户只需要在命令提示符中输入函数名并执行就可以绘制出精美的东方明珠电视塔图形。代码参见本教程的第 5 章第 4 节。

第 5 节 计算微积分工具

可编程科学计算器计算微分（求导）的工具非常简单易用，它支持计算一阶，二阶和三阶倒数表达式或者导数值。如果用户没有输入变量的数值，可编程科学计算器就给出导数表达式，否则就给出导数值。可编程科学计算器计算微分的工具本质上是调用的 derivative（求取导数表达式）和 deri_ridders（求取导数值）函数。

可编程科学计算器计算积分的工具使用起来也比较简单，它支持不定积分和从一次到三次定积分。在计算定积分时，积分的起始值和终止值可以是实数，复数，甚至正负无穷（inf 或者 -inf）。积分的步数必须是正整数或者为 0。当积分步数为 0，或者积分的起始值或者终止值为无穷或者负无穷时，会自动调用高斯克朗得法进行积分，速度会比较慢，精确度会大大提高，甚至也可以处理某些有奇异点的情况。也正是因为高斯克朗得法的速度比较慢，所以当进行二次或者三次积分时，强烈不建议使用高斯克朗得法。

可编程科学计算器计算积分的工具本质上是调用的 MFP 编程语言的 integrate 函数。比如，计算不定积分时，输入被积分表达式为 $x+3$ ，第一个积分变量的名字为 x ，得到结果显示为 Integrate("x+3", "x") = "3*x+0.5*x**2"，表示不定积分结果是 3 乘以 x 加上 0.5 乘以 x 平方。再比如，计算 $\exp(x)$ 从负无穷到 0 的定积分，输入被积分表达式为 $\exp(x)$ ，第一个积分变量的名字为 x ，积分起始值为 -inf，终止值为 0，积分步数任意填写（因为无论积分步数设置是否为零，积分起始终止范围包含无穷就肯定会调用高斯克朗得法），得到结果为 1。

第 6 节 设置输入键盘

用户可以自定义若干输入键盘以便在计算器中快速输入自定义的函数。每一个输入键盘均有横置和竖置模式对应于计算器的横置和竖置模式。用户可以通过点击菜单按钮来增加或删除输入键盘。

每一个输入键盘均有一个独一无二的名字以方便系统识别。此外每个键盘还拥有一个长名字，双行名字和短名字。如果用户不选中可见框，键盘将在计算器中隐藏。

用户可以在一个输入面板上增加（点击输入面板最底部的+按钮）或删除按键（按住按键按钮不放，就会出现上下文菜单让用户选择）。如果用户点击某个按钮，将对按钮的显示文字，显示文字的颜色，输入文字，和点击按钮后输入光标的位子进行编辑。注意，这里的输入光标的位子是从后开始计算，比如某一个按钮，输入文字为 `my_func()`，我们希望在点击按钮后，输入光标位于括号和反括号之间，则输入光标的位置应该为 1，表示向前数一个字符。

输入键盘设置保存在移动设备的 SD 卡的 `AnMath/config` 目录中，名字叫 `inputpad.cfg`（用于智慧计算器）或者 `inputpad_c1.cfg`（用于命令提示符）。如果这个文件不存在，计算器将自动载入默认键盘。`inputpad.cfg` 和 `inputpad_c1.cfg` 实际上是基于 `xml` 的文本文件，对此感兴趣的用户可以尝试自己编辑。如果想恢复默认键盘，用户删除此文件即可。

第 7 节 管理文件和编写程序

可编程科学计算器能够将用户的数据文件保存在外部存储设备（比如 SD 卡）中。当可编程科学计算器启动的时候，它会检查是否有数据文件夹。如果没有，可编程科学计算器会尝试创建该文件夹。如果创建不成功会有错误提示信息。文件夹路径将会在设置窗口中显示。

用户能够通过一个内置的文件管理器浏览和使用文件。在文件管理器中，用户点击图标以选中一个文件或文件夹。如果一个文件或文件夹被选中，它的背景颜色变成橘红色。如果想运行一个程序文件或打开一个文件夹，用户可以按住该文件或文件夹的图标不放或者先选中该文件或文件夹，然后点击菜单按钮选择“运行”或者“打开”菜单。用户也可以通过选择菜单来删除或者重命名一个文件或文件夹。

用户可以打开一个程序源文件进行浏览或修改。可编程科学计算器有一个内置的编辑器。该编辑器提供基本的程序编辑功能。用户也可以打开图像文件进行浏览。如果用户想研究图像文件的格式，可以在 PC 上用普通的文本编辑器打开它看里面是什么。

第8节 设置工具

可编程科学计算器提供了以下设置：

1. 数值精度。这项设置决定了小数点之后显示多少位有效位数。例如，如果数值精度是 4 位，0.003204876 将会被显示为 0.003205。注意这一项设置同时应用于命令提示符界面。
2. 科学计数。这项设置决定了使用科学计数方式（例如 2.1e-37）显示数值的数值范围。注意这一项设置同时应用于命令提示符界面。
3. 历史记录长度。这项设置决定了多少项历史计算记录将会被保存。注意这一项设置同时应用于命令提示符界面。但命令提示符和智慧计算器界面并不共享历史记录。
4. 绘制公式图形时变量变化范围。缺省范围是从 -5 到 5。
5. 开启应用时自动启动程序。这项设置决定了当用户启动应用时的自动启动功能。在缺省状态下，没有程序功能会自动启动，用户将会看到程序功能列表。
6. 按计算器按钮时震动。用户选择这项选项后点击计算按钮设备将会振动。
7. 当命令提示符在运行命令时，不隐藏输入键盘。用户选择这个选项后，在命令提示符中每次点击运行按钮，输入键盘不会消失。这样虽然命令提示符输出窗口的可视面积变小，但用户不用再次点击命令提示符输出窗口以弹出输入键盘，大大方便了需要高频快速输入的用户。
8. 文件夹路径。用户可以选择在哪一个存储卡上存储软件生成的数据。这里还有几项只读的文件夹路径。第一个是应用的数据文件夹；第二个是程序文件夹；第三个是图片文件夹；第四是 APK 安装包所在文件夹；第五是 APK 签名文件所在文件夹。这些设置在以后的版本中将可以被用户编辑。

第9节 在电脑上运行可编程科学计算器工具

为了方便用户在移动设备和不同的电脑中使用本软件，从 1.1 版开始，安卓版可编程科学计算器包括了一个基于 JAVA 的图形界面的可编程科学计算器，可应用于任何安装了 JAVA（1.6 或者以上版本）的 Windows、MacOSX

和 Linux 的电脑。从 1.7.1 版开始，安卓版可编程科学计算器又提供了一个基于命令控制台（也就是 Windows 中的 Dos/Cmd/Powershell 窗口或者 Unix/MacOSX/Linux/Cygwin 中的终端）的 JAVA 版 MFP 语言解释器，使用该解释器，`mfps` 脚本可以用和 Python, Perl 以及其他任何脚本语言一样的方式在个人电脑上运行。

每一次安卓上的可编程科学计算器软件升级的时候，基于 JAVA 的可编程科学计算器的最新版会被自动拷贝到 SD 卡上，所以，一般不需要用户运行“在电脑上运行”这个工具。但是，有时候，比如在升级安卓上的可编程科学计算器软件时，用户的 SD 卡被拔出或者接触不良；或者在升级安卓上的可编程科学计算器软件之后，用户不小心将 SD 卡上的软件误删，这时，就需要运行这个工具将基于 JAVA 的可编程科学计算器的最新版重新拷贝到 SD 卡上以便在电脑上使用。

“在电脑上运行”这个工具仅包含两个步骤，第一步是拷贝基于 JAVA 的可编程科学计算器至 SD 卡，第二步是确认并提示用户下一个步骤。在执行完成之后，用户需要使用一条 USB 电缆，将可编程科学计算器和一台装有 JAVA 的个人电脑连接起来，参见下图：

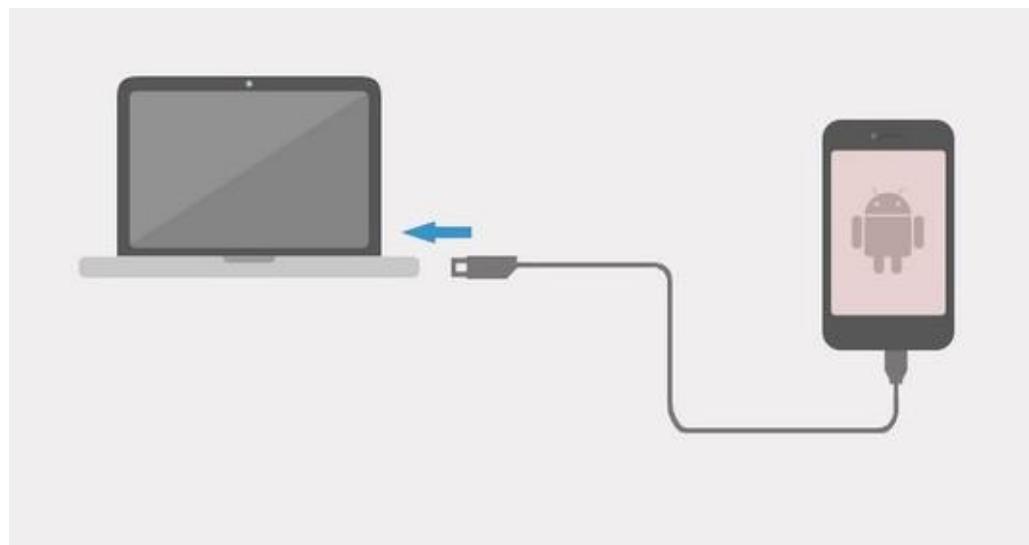


图 1.27：用 USB 电缆连接安卓设备和装有 JAVA 的个人电脑。

在安卓设备连接上电脑之后，有的安卓设备的 SD 卡能够自动被电脑找到，有的安卓设备则会提示用户，准备和电脑交换文件，用户需点击确认后，电脑才能够找到该安卓设备的 SD 卡，参见下图：



图 1.28： 用户确认将在电脑上打开 USB 存储设备，也就是 SD 卡（适用于部分安卓设备）。

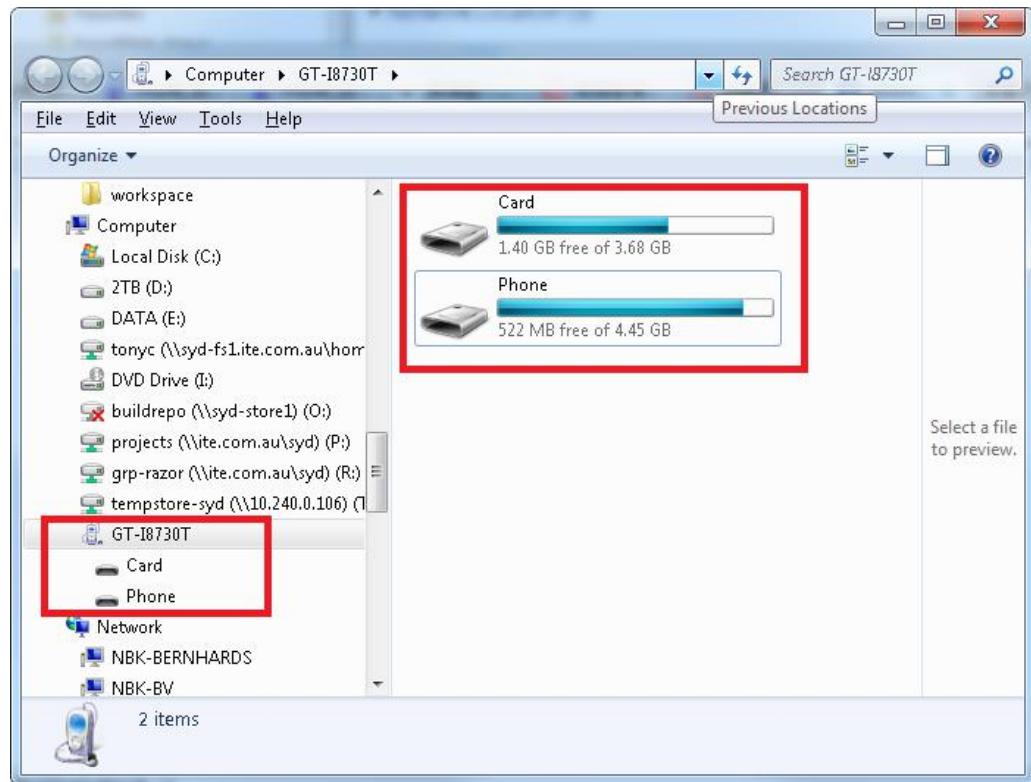


图 1.29： 用户个人电脑找到安卓设备的 SD 卡。

以上步骤完成之后，用户打开我的电脑，找到安卓设备所对应的移动磁盘，参见上图。

需要注意的是，有的安卓设备拥有多个 SD 卡，其中一个是自带的内部存取器，其他的是用户插入的扩展 SD 卡。可编程科学计算器在一开始都是将基于 JAVA 的可编程科学计算器组件拷贝到第一个 SD 卡，也就是自带的内部存取器上，但是用户可以在设置工具中重新设置为其他的 SD 卡。在上图例子中，第一个 SD 卡对应的名字叫 Phone，用户扩展的 SD 卡对应的名字为 Card。打开第一个 SD 卡，在其中找到 AnMath 子目录，进入子目录用户可以发现有一个 scripts 文件夹（下图蓝色方框中），在该文件夹内保存有所有的用户脚本代码；还有一个 JMathCmd. jar 文件（下图红色方框中），该文件就是 JAVA 版基于图形界面的可编程科学计算器；此外另有三个文件， JMFPLang. jar， mfplang. cmd 和 mfplang. sh（下图绿色方框中），用于基于系统控制台的 MFP 语言解释器。用户可以在个人电脑中编写和测试程序并存入移动设备中（保存在 AnMath\scripts 目录下），以便以后在移动设备中使用。整个流程参见下图：

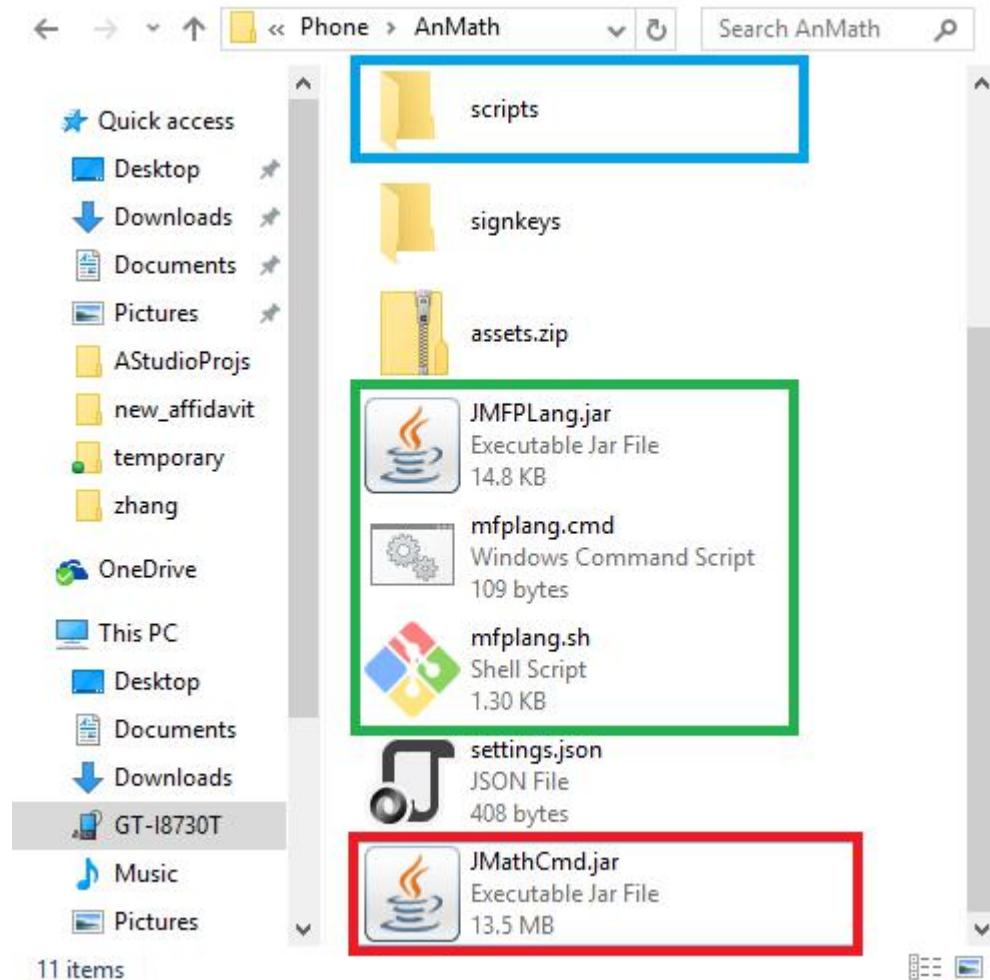


图 1.30： 用户打开 SD 卡找到 JMathCmd. jar 文件， JMFPLang. jar 文件， mfplang. cmd 文件， mfplang. sh 文件和 scripts 目录。

有一些安卓设备，比如三星 Galaxy Express，在连接到个人电脑上后，不允许用户从个人电脑上直接在手机存储卡文件夹中执行文件或创建新的文件。解决办法是，把位于手机存储卡中的 AnMath 目录整个拷贝到个人电脑的一个可读写的位置，在那里创建新的.mfps 函数程序文件，并且在那里启动基于 JAVA 的可编程科学计算器以调试编写的代码。调试完成后，再将 AnMath 文件夹整个拷贝到手机存储卡中的原来位子以覆盖原来的 AnMath 目录。

JAVA 版的图形界面的可编程科学计算器的启动屏幕如下（用户在装有 JAVA 的个人电脑上双击 AnMath 目录下的 JMathCmd. jar 文件即可启动）：



图 1.31： JAVA 版的可编程科学计算器启动屏幕。

基于 JAVA 的图形界面的可编程科学计算器实际上还是一个命令提示符工具。使用者输入命令，按回车，然后命令提示符将输出和返回值打印在输入行的下面。在基于 JAVA 的图形界面的可编程科学计算器中，一条命令可以包括一行或者多行（也就是多个语句组成的程序块），但命令内部不能包括对函数的定义。比如下述 6 条语句组成一个程序块，用户可以将它们一起拷贝粘贴到命令提示符中运行：

```
Variable a
```

```
if 3 > 2
```

```
a = 10
```

```
else
```

```
a = 9
```

```
endif
```

，但是，如果有函数的定义，比如：

```
Function abcde()
```

```
Return 3  
  
Endf  
  
Variable a = abcde()
```

则是不允许的。软件不能够在命令提示符中一并执行上述 4 条语句。

使用者除了可以通过拷贝和粘贴的办法将一行或者多行拷贝到命令提示符中，还可以通过键入 Shift-Enter 在一个命令中开始新的一行（由于一些安卓输入法不支持 Shift-Enter 键盘事件，键入 Shift-Enter 在一个命令中开始新的一行是基于安卓的命令提示符工具所不具备的功能）。和基于安卓的命令提示符工具不同，无论输入提示符在命令中间还是命令末尾，键入 Enter（回车）键（没有同时按下 Shift）都会触发该命令被执行。JAVA 版的图形界面的可编程科学计算器人机交互界面参见下图：

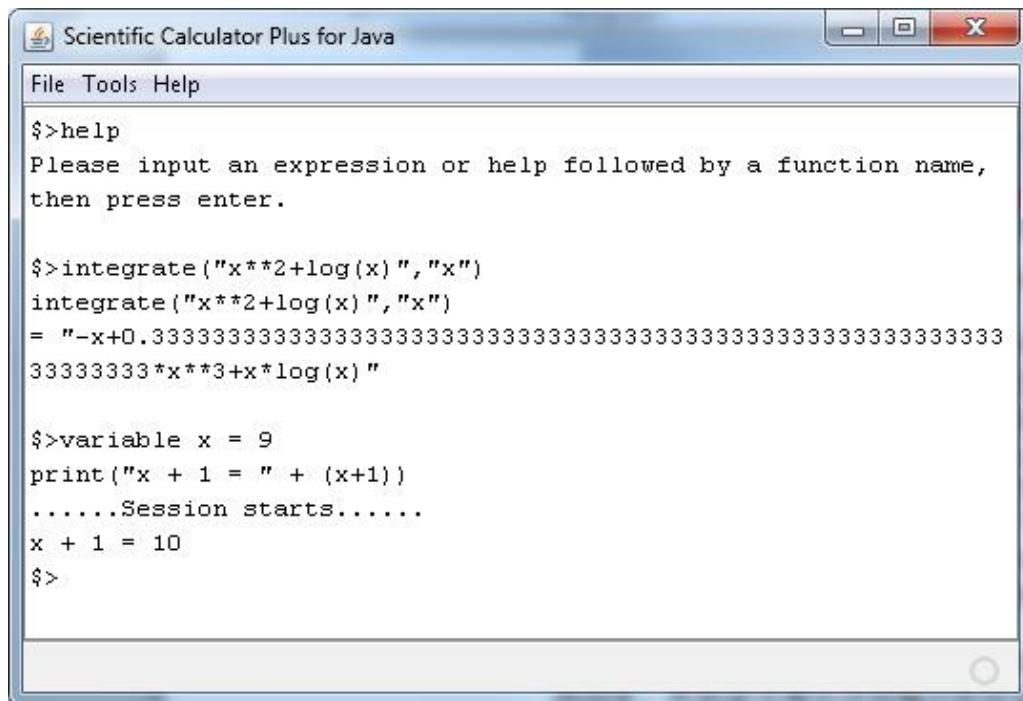
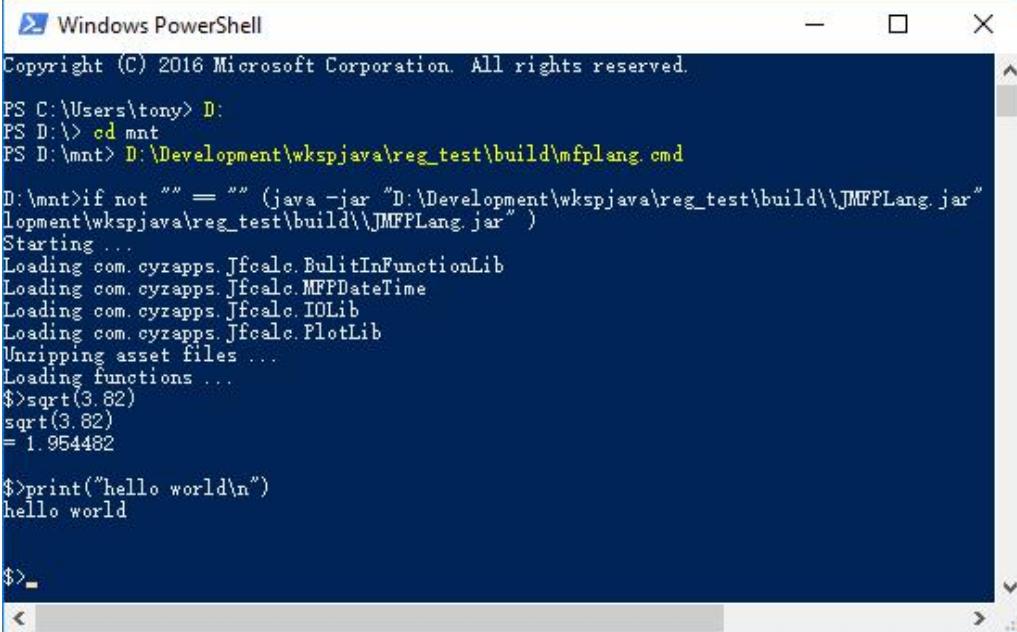


图 1.32: JAVA 版的基于图形界面的可编程科学计算器人机交互界面。

基于 JAVA 的系统控制台界面的可编程科学计算器和 Python 脚本语言的解释器工作机理非常类似。在 Windows 系统中，用户需要打开一个命令提示符或者 PowerShell 窗口，然后输入

Path\to\mfplang.cmd

并回车，MFP 语言解释器人机交互界面便启动了，参见下图。注意这里的 path\to\mfplang.cmd 是 mfplang.cmd 文件所在的路径。



```
Windows PowerShell
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.BuiltinFunctionLib
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

$>
```

图 1.33： JAVA 版的基于系统控制台界面的 MFP 语言解释器人机交互界面。

相应地，在 Unix/MacOSX/Linux/Cygwin 系统中，用户打开一个终端，然后输入

path\to\mfplang.sh

并回车，也可以启动 MFP 语言解释器人机交互界面。但和在 Windows 下不同的是，用户需要执行

Chmod 777 path\to\mfplang.sh

命令将 mfplang.sh 文件设置为可执行。

用户需要注意一点，和图形界面的 JAVA 版可编程科学计算器所不同的是，MFP 语言解释器人机交互界面不支持 Shift-ENTER 键。换句话说，它不支持多行命令输入。

如果用户想要退出 MFP 语言解释器人机交互界面，输入 quit 并回车即可。Ctrl-C 键也可以终止 MFP 语言解释器人机交互界面。

如果用户想直接运行某个 mfps 脚本而不是启动交互式 MFP 语言解释器，在 Windows 中用户可以输入

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

， 在 Unix/MacOSX/Linux/Cygwin 中，则需要执行

```
path1/to/mfplang.sh path2/to/script.mfps param1 param2 ...
```

。 这里 path1/to/mfplang.cmd（或者 path1/to/mfplang.sh）是文件 mfplang.cmd（或者 mfplang.sh）的路径，path2/to/script.mfps 是需要执行的脚本的路径。注意脚本文件名不一定必须是 script.mfps。Param1, param2, …，是脚本的运行参数（如果有的话）。

为了让 MFP 解释器能够知道脚本文件的入口函数，用户必须在脚本文件的头部，在任何 MFP 代码之前但是 shebang 声明行和文件帮助信息内容之后声明@execution_entry。比如一个 mfps 脚本包含以下内容：

```
#!/usr/bin/mfplang

# 本行是文件级的帮助信息。和 MFP 代码无关。本行必须位于 shebang 行

# 之后和@execution_entry 声明之前。

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

Citingspace ::test_cs

Function test_f(a, b)

Return a + b

Endf

EndCs
```

假设用户正在使用 Windows 系统，上述文件的文件名是 myscript.mfps。该文件位于当前工作目录中并且 AnMath 目录位于用户的路径搜索列表中。那么，用户只需要输入

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

便可得到结果为 7。如果用户使用的是 Unix/Linux/MacOSX/Cygwin，那么用户需要在/usr/bin 目录下建立一个软链接 mfplang，直接链到 mfplang.sh 文件。然后在终端中输入

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

同样也可以获得结果为 7。

@execution_entry 的使用方法将在后面的章节中详细说明。

无论是图形界面的 JAVA 版可编程科学计算器，还是基于系统命令行的 MFP 语言解释器，它们的设置文件都是位于 mfplang.cmd 和 mfplang.sh 所在目录（通常也就是 AnMath 目录）的 settings.json 文件。如果这个文件不存在，那么软件将会使用缺省设置。一个示例 settings.json 文件的内容如下：

```
{
    "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"
}
```

。这里要注意的是 SCRIPT_FOLDER_PATH 和 ADDITIONAL_USER_LIBS 两项设置，在 1.7.1 版以前，JAVA 版可编程科学计算器的所有脚本文件都是保存在根目录（也就是 AnMath 目录）的 scripts 目录中，和安卓版的可编程科学计算器完全一致。但是从 1.7.1 版开始，JAVA 版可编程科学计算器的脚本库目录是可以改变的，并且可以有多于一个脚本库，除了第一个脚本库（也就是旧版本的 scripts 目录），其他的脚本库都可以是一个 mfps 文件，也可以是一个目录。这样大大增加了 MFP 语言的灵活性。还要注意，如果脚本库的路径不是绝对路径，而是相对路径，比如上面 “.. \\externlibs\\ 第一 folder”，那么是相对于 mfplang.cmd 和 mfplang.sh 所在目录（通常也就是 AnMath 目录）的路径，而不是相对于用户当前目录的路径。

用户不必手动创建或者修改这个 settings.json 文件。用户可以打开图形界面的 JAVA 版可编程科学计算器，选择“工具”菜单，“设置”子菜单，软件会弹出设置对话框，参见下图。其中，红色方框是第一个脚本库，缺省设置为 scripts 目录。注意该脚本库必须是一个目录。蓝色方框是其他脚本程序库，每一行代表一个库，每个库既可以是一个目录，也可以是一个文件。

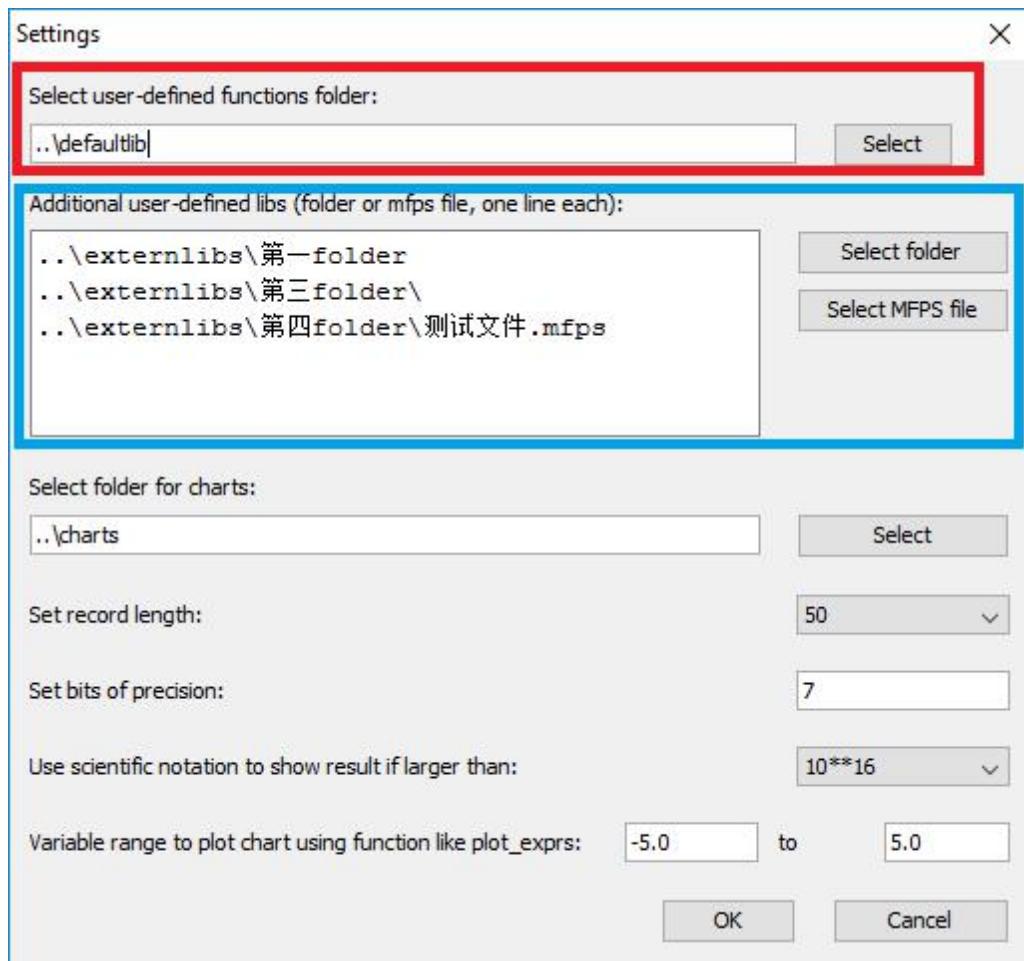


图 1.34：设置 JAVA 版可编程科学计算器的脚本程序库所在路径。

当然，在安卓上，用户很难知道具体的文件路径，所以，在安卓版的可编程科学计算器中，所有的 mfps 脚本文件还是保存在 scripts 目录中，如果一个 mfps 脚本文件不是保存在 scripts 目录中，那么，它在安卓版的可编程科学计算器启动时不会被加载。但如果该文件正确定义了 @execution_entry 标注，用户可以在安卓版的可编程科学计算器的文件管理器中长按该文件的图标运行它，运行完之后，该文件中的内容将会被从内存中卸载，用户还是无法在智慧计算器或者命令提示符中运行该文件中定义的函数。

第 10 节 创建 MFP 应用工具

可编程科学计算器的每一次计算都实际上是基于其数学引擎，MFP 编程语言，的一个函数。为了方便用户，可编程科学计算器提供了将一个 MFP 函数

数（不论是软件自带的函数还是用户自己编写的函数）编译打包为独立的安卓 APK 安装包的功能。

用户需要进行 3 步设置以生成一个安卓 APK 安装包。第一步，用户需要输入应用名称，应用包 ID 和应用版本信息。需要注意应用包 ID 必须是 20 个字节长的独一无二的 ID，否则用户创建的应用无法在诸如谷歌商店的网站上发布。用户还可以在这一步为应用选择图标和输入应用的帮助信息。如果用户选择使用缺省帮助信息，MFP 函数的帮助信息将会在应用帮助页面中显示。

第二步用户需输入函数名和 APK 的最终使用者需要输入的参数。注意 APK 的最终使用者需要输入的参数和函数的参数不见得完全一样。APK 的创建者可以为函数的一些参数设置缺省值，这样可以 APK 的最终使用者就不必进行太多的输入。

最后一步是设置 APK 文件名以及设置如何为 APK 文件签名。如果用户使用测试用签名，APK 文件可以被安装，但不能被发布。如果想发布，就必须使用已有的签名或者创建一个新的签名。用户所有的签名都保存在 AnMath\signkeys 目录中。

以上 3 步完成之后，将会出现一个提示对话框询问用户下一步是安装还是共享创建的 APK 包，抑或直接退出。如果用户选择退出，他（她）仍然可以在以后访问位于 AnMath\apks 目录的 APK 包。

创建 MFP 应用最好是建立于用户有一定的编程基础上，所以，在本手册的对 MFP 编程章节之后，在第 8 章中，对如何创建 MFP 应用有着进一步的说明，用户可以参见该章获取更详细的信息。

第 11 节 帮助工具

可编程科学计算器从第一版开始就提供了完整的，html 格式的帮助手册，启动可编程科学计算器之后，点击救生圈图标，即启动了帮助。用户可以点击帮助文档中的链接进入不同的帮助页面。

从 1.6.7 版开始，可编程科学计算器还提供了基于 PDF 格式的用户手册和 MFP 编程语言程序开发指南（也就是本文档）。用户点击救生圈图标，会让用户选择是阅读 html 格式的帮助文档，共享或者阅读（需要装有 PDF 阅读

器软件比如 Acrobat Reader) PDF 格式的手册还是拷贝示例代码（以方便阅读和执行），参见下图：



图 1.35： 可编程科学计算器的帮助选项。

除了独立的帮助工具，用户在启动其他工具后，比如智慧计算器或者计算积分，点击菜单按钮，也可以选择帮助菜单进入对应的帮助页面。

在智慧计算器和命令提示符的输入键盘上，还有快捷帮助按钮。用户点击快捷帮助按钮，然后再敲入函数名或者操作符名或者 MFP 编程语言的关键字，然后点击运行按钮，可以让智慧计算器或者命令提示符显示对应的函数，操作符或者关键字的帮助信息。

小结

可编程科学计算器是为所有的，哪怕没有编程基础的用户所开发的软件。本章详细介绍了在无需编程的情况下使用可编程科学计算器进行计算和绘图的办法。

本章内容的重点在于，第一，如何使用智慧计算器进行计算，数学公式拍照识别以及对数学表达式绘图；第二，如何在命令提示符中运行函数以及如何在个人电脑上运行图形界面和基于系统控制台的 MFP 命令提示符工具；第三，如何根据变量的表达式和变化范围在不同的坐标系中绘制图形；第四，如何计算导数，定积分和不定积分；第五，如何设置输入键盘。其他的内容，比如程序文件和创建 MFP 应用虽然不是关于编程，但是和编程密切相关，如果用户对编程不感兴趣，完全可以跳过。

本章最吸引人的部分在于根据变量的表达式和变化范围绘制三维图形。本章详细展示了绘制球体，柱体，椎体以及非常复杂的图形比如上海“东方明珠”电视塔的办法。如果用户有兴趣，也可以尝试用可编程科学计算器提供的绘制 3 维图形工具绘制一些其他的有趣图形，比如 3 维金字塔图（金字塔图形有四个侧面都是三角形，而底座则是正方形），这对于帮助用户熟练掌握本软件的使用是一个很好的锻炼。

第 2 章 MFP 编程语言基础

正如前面所指出，即便用户对编程完全不了解也可以使用可编程科学计算器的绝大部分功能。但是，如果用户能够自己写一些简单的代码，把这些功能组合起来，就能够实现一些用普通的计算器无法完成的计算。

第 1 节 MFP 编程语言概述

可编程科学计算器内置的 MFP 编程语言就是实现功能组合的有力工具。MFP 编程语言仅包含十多条语句，简单易学，但又功能强大。如果用户要把一组可编程科学计算器的功能组合在一起，一并运行，可以使用 MFP 语言，编写一个函数，在函数中调用这些功能和其他用户自己编写的函数。换句话说，使用 MFP 语言编程，函数，是用户自己创建功能组的入口。

函数包含函数名和参数（如果有参数的话），在函数编写完成之后，用户可以打开命令提示符或者智慧计算器或者运行电脑上基于 JAVA 的可编程科学计算器，输入

函数名(参数 1 的值，参数 2 的值，参数 3 的值…，最后一个参数的值)

或者如果函数没有参数，输入

函数名()

然后点击运行按钮（在安卓上运行命令提示符或者智慧计算器），或者按回车键（在电脑上运行基于 JAVA 的可编程科学计算器），函数即开始执行。需要注意的是，在智慧计算器中，用户的打印语句输出不会显示，所以推荐在命令提示符上或者在基于 JAVA 的可编程科学计算器上运行函数。

MFP 语言支持常用的操作符，比如+，-，*（乘），/（除），**（次方），&（位与），|（位或），^（异或），=（赋值），==（等于）等，和数学表达式。除此之外，MFP 语言还支持解代数方程（组），二进制数（格式为 0b 打头，比如 0b0011100），八进制数（格式为 0 打头，比如 0371.242），16 进制数（格式为 0x 打头，比如 0xAF46BC.0DD3E），复数，数组和矩阵，字符串，函数，变量，条件语句，循环语句，注释和帮助等等。MFP 语言对大小写不敏感。但是需要注意的是，在 MFP 代码中，除了字符串中的内容和注释，不能出现中文和其他任何双字节基于 Unicode 的字符。比如，用户定义一个函数的名字叫我的函数 1，是非法的，只能将名

字改为 my_func1。还要注意，有时候，代码中的字符看起来像是普通的单字节 ASCII 字符，但是实际上是双字节基于 Unicode 的字符，比如 1 是双字节基于 Unicode 的字符而不是普通的单字节 ASCII 字符 1，这种情况下，运行 MFP 代码会报错。所以，用户在拷贝粘贴代码的时候必须特别小心，因为从网页，email 或者 Word 文档中拷贝出来的文本往往带有双字节基于 Unicode 的字符。

MFP 编程语言包含以下语句：

```
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  
cittingspace, using cittingspace  
@compulsory_link  
@build_asset  
@execution_entry
```

在 MFP 语言中每一条语句可以占据一行或者多行。换句话说，如果一条语句太长，它可以被分割为几行，除了最后一行在每一行的末尾都跟随有字符串“_”。例如，假设一个函数拥有十个参数，如果将函数声明语句放在一行中将会非常的长：

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

。为了让程序代码更容易阅读，我们可以把函数声明语句分成 3 行，如下所示：

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

。并且我们依然可以在每一行的末尾添加注释，在以上例子中，我们可以为每一行添加注释如下（每一行//符号和其后面的文字就是注释，注释对代码的逻辑没有影响，仅仅为了方便阅读）：

```
function abcde(para1, para2, para3, _// 第一行有 3 个参数。  
para4, para5, para6, _ // 第二行也有三个参  
数。  
para7, para8, para9, para10) // 第三行有 4 个参数。
```

需要注意的是在 MFP 语言中每一行最多只能容纳一条语句。不像 C/C++，MFP 语言不支持语句分隔号，比如“;”。

MFP 编程语言还包括一些标注，比如@language 和@compulsory_link。标注不影响程序的运行，但是会在生成帮助信息和编译 APK 安装包时起作用。

最后要注意，MFP 编程语言的代码源文件的后缀一定是.mfps（不区分大小写），比如 test.mfps，myFunc.mfps 等等。如果不是这个后缀，软件将不会读入源文件。

第 2 节 MFP 编程语言基本数据类型

MFP 编程语言支持如下数据类型：

1. 布尔值（TRUE 或者 FALSE）。

- 普通的实数值。注意实数值可以为整数，也可以为分数。用户不必像使用 C 语言那样关心一个实数值到底是整数还是分数，MFP 会自动判断并作转换。MFP 编程语言可以处理任意大的数，比如，10000 的 10000 次方；对于很小的数，MFP 的精度则可以达到 $5*10E-48$ 。当数值小于这个值时，MFP 会视该值为 0。

注意，MFP 不但支持 10 进制数据格式，还支持二进制数据格式，8 进制数据格式和 16 进制数据格式，对于每个进制都既支持整数，也支持分数。二进制数格式为 0b 打头，比如 0b0011100，八进制数格式为 0 打头，比如 0371.242，16 进制数格式为 0x 打头，比如 0xAF46BC.0DD3E。用户可以按照这些格式输入这些数据，在计算时，MFP 自动将它们转化为十进制数进行计算，显示的结果也是十进制数。如果用户想把一个十进制数转换为其他进制，需要调用函数 conv_dec_to_bin（十进制转二进制），conv_dec_to_hex（十进制转 16 进制）和 conv_dec_to_oct（十进制转 8 进制）。

还要注意，布尔值可以被看作是一种特殊的实数值，布尔值 TRUE 的对应实数值为 1，FALSE 的对应实数值为 0。实数值 0 的对应布尔值为 FALSE，其他实数值对应的布尔值为 TRUE。实数值和布尔值之间的转换是自动进行的，不需要用户干预。

最后，实数值还包括一个特殊的值，NAN，这个值表示没有定义的数值，比如用 1/0 会得到 NAN。

- 复数值。复数值实际上是两个实数值的组合，一个实数值对应实部，一个对应虚部。复数在 MFP 语言中被写为 $a + b * i$ 或者 $a + bi$ ，注意 a 和 b 都是可正可负的实数。如果 a 等于 0，该复数可以被写成 $b*i$ 或者 bi 。注意在 MFP 的语法中， bi 是对的，但是 $b i$ （b 和 i 之间有空格）是错的。MFP 会认为隔有空格的 b 和 i 是两个不同的变量。如果 b 等于 0，MFP 会在适当的时候将 $a + bi$ 转化为实数 a。反过来，如果需要进行复数运算，MFP 也会自动地把一个实数转化为虚部为 0 的复数。以上这些转化，都不需要用户干预。但需要注意，除非虚部为 0，否则复数值是不可以转化为布尔值的。最后，复数值还包括一个特殊的值，NANi，这个值表示没有定义的虚数数值，就如同 NAN 表示没有定义的实数数值一样。

4. 数组和矩阵。MFP 支持数组矩阵操作。数组是由一列元素组成，元素之间由“,”分割开，整列元素由“[”和“]”所包围。考虑以下例子：[1, 2, 3+i], [[1, 2, 3+i]] 和 [[4, 5], [sqrt(8.9), -i], [1.71, stdev(2, 3, 4)]]。在这些例子中，[1, 2, 3+i] 是包含 3 个元素的一维向量（也是数组），[[1, 2, 3+i]] 是 1*3 的数组而 [[4, 5], [sqrt(8.9), -i], [1.71, stdev(2, 3, 4)]] 是 3*2 的数组。

为存取数组中的元素，编程人员需要使用索引。索引是由一列由“[”和“]”所包围的正整数所组成。例如，考虑一个数组变量 a 等于 [[4, 5], [sqrt(8.9), -i], [1.71, stdev(2, 3, 4)]], 那么 a[2, 0] 就是 1.71, a[1] 就是 [sqrt(8.9), -i]，注意索引值都是从 0 开始。考虑另外一个例子，假设变量 b 是 [1, 2, 3+i]，那么 b[2] 等于 3+i，但 b[0, 2] 则不合法。

MFP 语言能够支持矩阵的加，减，乘，除和转秩。比如，[[1, 2], [3, 4], [5, 6]] + [[2, 3], [4, 5], [6, 7]] = [[3, 5], [7, 9], [11, 13]]（矩阵加法），[[2, 3], [4, 5]] * 2 = [[4, 6], [8, 10]]（矩阵乘法），[1, 2] * [[3, 4]] = [11]（矩阵乘法），[[5, 6], [7, 8]] / [[1, 2], [3, 4]] = [[-1, 2], [-2, 3]]（矩阵除法）和 [[2, 3], [4, 5]]' = [[2, 4], [3, 5]]（矩阵转秩）。注意在矩阵加减法中矩阵的维度必须相同，在矩阵乘法中，第一个操作数的最后一维的长度必须第二个操作数的第一维的长度。矩阵的除法则要求除数和被除数为同尺寸方阵。

不同于 Matlab，MFP 不支持在对矩阵元素赋值时自动改变矩阵的尺寸和维度。比如，假设一个数组变量 a 是 [1, 2]，那么语句 a[1]=3 是合法的但语句 a[2]=3 或者 a[0, 1]=3 是非法的（因为超过了数组的尺寸和维度）。幸运的是，MFP 有很多内建的和预定义的函数，使用这些函数能够访问和修改甚至超过矩阵尺寸和维度的矩阵元素。这样一来，使用者可以调用这些函数来增加矩阵的尺寸和维度。

5. 字符串。MFP 支持字符串操作。字符串的引用符为双引号（注意双引号一定要是英语输入法的单字节 Ascii 双引号，也就是”，而不是中文输入法中的双字节 Unicode 双引号，比如“或者”），比如，

"abc"表示字符串 abc。和 C/C++不同，MFP 的字符串并非字符所构成的数组。MFP 提供了一系列内置的和预定义的字符处理函数。

6. NULL。NULL 表示一个空的，不存在的数值，通常用于表示函数返回值不是需要的数值。NULL 无法和其他的数据类型相互转化。

第 3 节 MFP 编程语言所支持的操作符

MFP 编程语言支持如下操作符：

运算操作符	定义	例子
=	赋值操作符。本操作符将一个数值赋给一个变量。	$x = 3 + 4$ 将数值 7 赋给变量 x。
==	等于号。本符号意味着本符号左边的表达式和符号右边的表达式有同样的数值。	$x + 1 == 7 + 3$ 意味着 $x + 1$ 和 $7 + 3$ 有同样的数值，也就是 10。通过这个表达式，我们可以解出 x 的值为 9。
(左括号。位于一对括号中的表达式有更高的运算优先级。	$x * (y + 3)$
)	右括号。位于一对括号中的表达式有更高的运算优先级。	$x * (y + 3)$
[左方括号。左方括号为一个数组的定义的开始。	$[[1,2,3],[4,5,6]]$ 定义了一个 2*3 矩阵，第一行为包括 3 个元素（也就是 1, 2 和 3）的数组，第二行也为包括 3 个元素（也就是 4, 5 和 6）的数组。该矩阵本身也为一个数组包括 2 个数组元素。

	右方括号。右方括号为一个数组的定义的结束。	$[[1,2,3],[4,5,6]]$ 定义了一个 2×3 矩阵，第一行为包括 3 个元素（也就是 1, 2 和 3）的数组，第二行也为包括 3 个元素（也就是 4, 5 和 6）的数组。该矩阵本身也为一个数组包括 2 个数组元素。
,	逗号将数组中的每一个元素分隔开。	数组 $[2,3,4,5]$ 有 4 个元素被 3 个逗号分隔开。
+	加号（支持复数，矩阵和字符串）	$2.3 + 4.6 + 2i = 6.9 + 2i$ $[1, 2] + [3, 4] = [3, 6]$ $[1, 2] + " hello" = "[1,2] hello"$
-	减号（支持复数和矩阵）	$4.6 - 2.3 = 2.3$ $[1, 2] - [3, 4] = [-1, -1]$
*	乘号，支持复数和矩阵，比加号和减号优先级高。	$1 + 2 * 3i = 1 + 6i$ $[[1, 2]] * [[3], [4]] = [[11]]$
/	除号，支持复数和矩阵，比加号和减号优先级高。	$6 - 3 / 2i = 6 + 1.5i$ $[[10, 0.5]] / [[1, 2], [3, 4]] = [[-19.25, 9.75]]$
\	左除号（主要用于矩阵相除，比如计算 $Ax=b$ 中的 x 时， $x=A\backslash b$ ，注意第一个操作数是除数，第二个是被除数，如果除数不是矩阵，和除号的功能完全一样）。	$6 - 3 / 2i = 6 + 1.5i$ $[[1, 2], [3, 4]] \backslash [[11], [32]] = [[10], [0.5]]$
&	位与，比加减乘除的优先级高。注意位与只接受非负运算数，并且将自动把运算数取整。	$4 + 5.2 \& 3.7 = 5$
	位或，比加减乘除的优先级高。注意位或只接受非负运	$2.8 / 5.2 3.7 = 7$

	算数，并且将自动把运算数取整。	
\wedge	位异或，比加减乘除的优先级高。注意位异或只接受非负运算数，并且将自动把运算数取整。	$14.2 / 5.2 \wedge 3.7 = 2.3667$
$**$	次方，比加减乘除和所有的位操作符优先级高。注意如果两个运算数都可以为复数。	$2 * 4 ** 3 = 128$ $(-4) ** 0.5 = 2i$ $3** (2+i) = 4.0935 + 8.0152*i$
$+$	正号，比所有的二元操作符优先级高。	$4 - +3 = 1$ $+[4,3] = [4, 3]$
$-$	负号，比所有的二元操作符优先级高。	$4 ** -1 = 0.25$ $-1 ** 4 = 1$ $-[3.71i, [4, 5.33-6i]] = [-3.71*i, [-4, -5.33+6*i]]$
$\%$	百分号，比所有的二元操作符和正负号优先级高。	$-401.78\% = -4.0178$ $5.77\% = 0.0577$
!	否，比所有的二元操作符和正负号优先级高。如果操作数为 0，返回 True，如果操作数非 0，返回 False。注意其操作数必须为实数。	$!-0.2 = \text{False}$
\sim	位非，比所有的二元操作符和正负号以及否的优先级高。注意它仅接受非负运算数。如果运算数不是整数，将自动转换为整数。	$\sim 0 = -1$ $\sim 0.1 = -1$ $\sim 9 = -10$

!	阶乘号，比所有二元操作符以及一元左置操作符优先级高。注意它仅接受非负运算数。如果运算数不是整数，将自动转换为整数。	$7! = 5040$
'	一维二维矩阵或者单一元素的转秩，比所有的二维操作符和左置一维操作符的优先级高。注意它能将一维矩阵转换为二维矩阵但如果操作数是单一元素，则返回操作数的原始值。	$(3 + 4i)' = (3 + 4i)$ $[1, 2+3i]' = [[1], [2+3i]]$ $[[1, 2], [3, 4]] = [[1, 3], [2, 4]]$

第4节 function, return 和 endf 语句

在 MFP 语言中 Function 语句是用户定义一个函数的开始。Function 语句的语法为（注意函数名参数名都必须是英文）：

Function 函数名(参数 1, 参数 2, 参数 3, ...)

。注意这里”...”意味着在 parameter3 参数之后还可能有任意个数个参数。如果一个函数没有可选参数，函数的声明应该类似于

Function abcd(para1, para2, para3, para4)

。换句话说不能够在参数列中有”...”。

如果一个函数有可选参数，可选参数的个数保存在一个系统变量中，该系统变量名字叫做 opt_argc。所有的可选参数作为一个数组保存于系统变量 opt_argv 中。换句话说，第一个可选参数的值为 opt_argv[0]，第二个为 opt_argv[1]，以此类推。

一个函数可以返回一个任意类型的数值或者什么都不返回。但使用者不用在 Function 语句中声明返回值，MFP 会自动处理返回值。

Return 语句用于退出函数并返回数值。比如：

```
Return "Hello word" // 返回字符串"Hello word"，注意代码中的双引号必须是英文单字节双引号"而不能是中文双字节双引号“或者”。
```

或

```
Return // 什么都不返回
```

。

Endf 语句用于标记函数的结尾，该语句不接受任何参数。

以下例子用于在命令提示符上打印世界你好语句并返回整数 1。注意由于包含用户的打印输出，这个例子只能在命令提示符或者基于 JAVA 的可编程科学计算器上运行，在智慧计算器上用户看不到世界你好的输出。本实例可以在本手册自带的示例代码所在目录（通常是 SD 卡中 AnMath 目录下的 scripts/manual 目录）中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

```
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 Helloworld(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 “.

    //虽然代码中不能有中文，但是字符串中和注释里可以有中文。

    //另外注意代码中的双引号必须是英文单字节双引号"而不能是

    //中文双字节双引号“或者”。

    print("Hello world! (Chinese:世界你好!) my name is " + my_name)

    return 1

endf

```

运行上述程序，在命令提示符或者基于 JAVA 的可编程科学计算器中输入命令::mfpxexample::Helloworld("Bob")并执行，得到输出结果为：

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

第 5 节 variable 语句

Variable 语句用于定义一个或任意多个变量，其语法为：

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

，这里 var1, var2, var3, ..., varN 为变量名，N 为一个正整数，
 {=expr*} 含义为变量初始化的部分可以省略（变量缺省初始值为 NULL，表示一个空值）。例如：语句

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

定义了 4 个变量名字分别为 a, b, c 和 d。它们的初始值分别是"hello, world"（字符串），NULL，"hello, world7"（这是将数字 7 添加到字符串 "hello, world" 的尾部得到字符串"hello, world7"）和[2, 3, [5, 8]]（数组）。

以下例子定义并赋值了两个变量 a 和 b 并打印输出它们的值。同上，这个例子只能在命令提示符或者基于 JAVA 的可编程科学计算器上运行，在智慧计算器上看不到打印输出。本实例可以在本手册自带的示例代码所在目录

(通常是 SD 卡中 AnMath 目录下的 scripts/manual 目录) 中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

```
function DeclareVariable()  
    variable a = [1, 2, 3], b  
    b = 4.75-0.67i  
    print("a = " + a + "; b = " + b + "\n") //\n 表示输出时换行  
endf
```

运行上述程序，在命令提示符或者基于 JAVA 的可编程科学计算器中输入命令::mfpxexample::DeclareVariable () 并执行，得到输出结果为：

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

第 6 节 if, elseif, else 和 endif 语句

If, elseif, else 和 endif 组成了 MFP 语言中的条件程序块。它们的语法如下：

```
If conditionA
```

//如果 conditionA 满足，执行后续语句直到遇到 elseif 或者 else，然后跳转到 endif 之后的那条语句执行。

```
.....
```

```
elseif conditionB
```

//如果 conditionB 满足，执行后续语句直到遇到 elseif 或者 else，然后跳转到 endif 之后的那条语句执行。

```
.....
```

```
elseif condition
```

//如果 conditionB 满足，执行后续语句直到遇到 elseif 或者 else，然后跳转到 endif 之后的那条语句执行。

```
.....  
.....  
.....  
.....  
.....  
else  
    //如果以上条件都不满足，执行后续语句直到遇到 endif，然后执行 endif  
    //之后的那条语句。  
.....  
endif
```

。在上述例子中，conditionA，conditionB 和 conditionC 均为表达式，它们的运算结果应该为布尔值。如果它们运算值不是布尔值，MFP 将会将运算结果转换为布尔值。如果做不到这一点（比如，强制将一个字符串或者数组转化为布尔值），MFP 将报错。

以下例子使用了条件语句对用户输入的数值范围进行分析。本实例可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

```
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在用户调用本函数的时候输入，必须为实数

    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

```

运行上述程序，在命令提示符或者基于 JAVA 的可编程科学计算器中输入命令::mfpxexample::IfStatement (-1) 并执行，得到输出结果为：

a < 0!

，输入命令::mfpxexample::IfStatement (0.5) 并执行，得到输出结果为：

0 <= a < 1!

，输入命令::mfpxexample::IfStatement (1) 并执行，得到输出结果为：

1 <= a < 10!

，输入命令::mfpxexample::IfStatement (20) 并执行，得到输出结果为：

a >= 10!

第 7 节 while, loop; do, until 和 for, next 语句

While 和 loop, do 和 until，以及 for 和 next 构成 MFP 语言中的三对循环语句。它们语法如下：

While condition // condition 的值为布尔量，为 TRUE 则进入循环

.....

Loop

Do

.....

Until condition // condition 的值为布尔量，为 TRUE 则终止 do 循环

For variable var = from_value to to_value step step_value

.....

Next

。在 for 语句中，condition 为一个表达式，其计算值为布尔值（或者可以被转换为布尔值）。Var 是 for 语句索引变量的名字。From_value 为 var 变量的起始值，step_value 为 var 变量每次增加多少，注意 step_value 可正可负。To_value 为 var 的目标值。如果 Var 的值超出了 to_value，var 将会停止改变。注意如果 var 曾经定义过，variable 关键字可以被省略掉。下面是 for 语句的一个例子，注意当 for 语句索引变量的值和 to_value 值相等时，for 循环仍然被执行，只有当索引变量的值超出（这个例子是小于）to_value 值，for 循环才被终止：

```
variable idx  
  
for idx = 1 to -1 step -2  
  
print_line("idx == " + idx)
```

next

上述代码的运行结果是

```
idx == 1  
  
idx == -1
```

注意这些循环均支持 break 和 continue 语句。如果 break 语句被执行，MFP 跳出最内一层循环。如果 continue 语句被执行，MFP 忽略最内层循环中的 continue 语句之后的语句并跳转到循环开始处开始执行。

以下例子使用了上述循环语句和 break 以及 continue 语句。本实例可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

```
function Testloops(a)

    if a == 0

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

        // 测试 for 语句, 以及 break 和 continue 语句

        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

            // 测试 while 语句和 do 语句

            variable idx = 4

            while idx < 8

                variable idx1 = idx
```

```
do
    idx1 = idx1 + 2
    until idx1 > 11
    idx = idx + idx1 /5
loop
//打印出最后 idx 的值
print("after while and do, idx value is " + idx + "\n")
endif
endf
```

运行本示例，如果用户输入命令为::mfpxexample::testloops(0)，最后的输出结果是

idx = -4

idx = -5

idx = -6

如果用户输入命令是::mfpxexample::testloops(1)，最后的输入结果为

after while and do, idx value is 8.88

第 8 节 break 和 continue 语句

Break 语句用于循环（也就是 While ... loop, do ... until, for ... next）或者条件程序块 select ... case ... default ... ends。Break 语句使 MFP 跳出最内一层循环或者 select ... case ... default ... ends 程序块。

Continue 语句用于循环（也就是 While ... loop, do ... until, for ... next）。Continue 语句使 MFP 忽略最内一层循环中该 continue 语句之后的语句并跳转到最内一层循环的开始处开始执行。

Break 和 continue 语句的代码示例参见上一节的例子。

第9节 select, case, default 和 ends 语句

Select, case, default 和 ends 构成了 MFP 中除 if 之外的另外一种条件程序块。他们的语法如下：

select expr // expr 是一个数值或者是可以算出一个最终值的表达式

case value1 // 如 expr 的值为 value1，执行下述语句直到遇到 break

.....

break // 遇到 break 就跳至 ends 语句的最后一句执行

case value2 // 如 expr 的值为 value1，执行下述语句直到遇到 break

.....

break // 遇到 break 就跳至 ends 语句的最后一句执行

.....

.....

.....

Default //如果 expr 的指不等于其上任何一个 value 值，执行下述语句

.....

ends // 程序块结束

。注意如果一个 case 块的结尾没有 break 语句，MFP 将会执行下一个 case 块或者 default 块（如果下个条件分支为 default）的语句。

以下例子使用了 select, case 和 default 语句对用户的输入参数值进行判断。本实例可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

```
@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.
```

```
            // 由于 case 2 没有 break 语句，MFP 将会继续执行 case 3 里面的代码，
```

```
            // 直到遇到一个 break 或者 ends 为止。
```

```
            print("a is 2\n")
```

```
        case 3
```

```
            print("a is 3\n")
```

```
            break
```

```
        default
```

```
            print("a is other value\n")
```

```
    ends
```

```
endf
```

用户运行这个程序，输入::mfpexample::TestSelect(1)并执行，会看到输出为

```
a is 1
```

，输入::mfpexample::TestSelect(2)，由于 case 2 没有 break 语句，所以会看到输出为

```
a is 2
```

```
a is 3
```

，输入::mfpexample::TestSelect(3)，输出为

```
a is 3
```

，输入::mfpexample::TestSelect(4)，会执行 default 语句后面的内容，所以输出为

```
a is other value
```

```
。
```

第 10 节 try, throw, catch 和 endtry 语句

Try 和与 try 相关的语句是用来侦测和处理程序中异常的工具，对于刚刚开始学习编程的人士来讲，可能比较复杂。由于依靠 try 程序块实现的大部分功能可以用条件语句通过打印输出的办法来代替，初学者可以跳过此节，阅读后面的内容。

在 MFP 语言中，异常指的是程序中的错误，比如遇到没有定义的变量，或者强制将一个数值转化为其不能转化的数值类型。当一个异常发生时，如果没有 try 语句的保护，MFP 会直接把异常丢给上一层调用函数。反之，如果有 try 语句块的保护，并且异常的类型和对应 catch 语句的捕获条件相符，异常将会被拦截住并处理。

详细说来，Try 语句用于开始一个 Try 程序块。Try 语句没有任何参数。在 Try 程序块中，任何由于 MFP 语言触发的异常将会被抛出并传递到跟随 try 程序块的某一个 catch 语句中处理。当然，如果没有一个 catch 语句能够处理这个被抛出来的异常，该异常将会被抛向外层的程序块或函数。

Throw 语句有一个字符串参数。如果 throw 语句被执行，MFP 打印出字符串参数然后退出。

Catch 语句可以接受一个表达式作为参数，也可以不接受参数。如果不接受参数，该 catch 语句捕获任何由该语句对应的 try 程序块抛出的异常。如果它有一个表达式参数，该表达式参数作为一个异常过滤器用于决定一个异常是否由该 catch 语句处理。如果异常过滤器表达式的值是布尔值 true，该异常被捕获，否则，该异常被传递到下一个 catch 语句，或者被抛向外层的程序块或函数。Catch 语句提供三个字符串类型的内部参数，也就是 level，type 和 info。参数 level 存储异常的层级，它的值是“LANGUAGE”（也就是编程语言级别的异常，比如没有 endif 语句跟随 if 语句或者一个用户定义的抛出字符串的异常）或者“EXPRESSION”（也就是表达式级别的异常，比如被零除或者缺少右括号），参数 type 是 MFP 编程语言内部定义异常的类型，参数 info 是异常的内容。如果开发人员用一个 throw 语句抛出一个字符串，info 的值就是这个字符串。这三个参数仅能用于 catch 语句中的异常过滤器。但是异常过滤器可以使用变量名和函数名空间中的任意变量和函数。如果一个变量和这三个 catch 语句的内部参数重名，它将被重名内部参数重载。

Endtry 用于结束一个 try/catch 程序块，它不接受任何参数。

以下例子使用了上述语句侦测运行过程中出现的错误并进行处理。本实例可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

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 (捕获所有异常)
```

```
print ("dbc is undefined\n") //变量 dbc 没有声明
```

```
EndTry
```

```
EndS
```

```
Throw "my exception" //throw an exception by user
```

```
// 用户主动抛出一个异常
```

```
Catch (1+2)==4
```

```
// here (1+2)==4 is an exception filter. If an exception
```

```
// satisfies (1+2)==4 it is caught here.
```

```
// 这里(1+2)==4 是一个异常过滤器，如果一个异常满足(1+2)==4，
```

```
// 它就会被这里的 catch 语句捕获
```

```
//will never be here because 1+2 never equals 4.
```

```
//由于 1+2 永远不等于 4，所以没有异常会被捕获。
```

```
print ("Exception satisfying (1+2) == 4 is caught")
```

```
catch false
```

```
// here false is another exception filter. Clearly,  
  
// no exception will be caught.  
  
// 这里 false 也是一个异常过滤器，显然，不会有异常被捕获  
  
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.  
  
// 这里的异常过滤器有两个条件，第一，异常的级别为"LANGUAGE"  
  
// 级，第二，异常的信息为"my exception"。我们可以通过比较异  
  
// 常级别(level)和异常信息(info)，并使用逻辑函数 and 和 or，  
  
// 来定义各种各样的异常过滤器。  
  
//  
  
// 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.  
  
// level 和 info 是 catch 语句内部的变量，不是 catch 语句不能够用。  
  
// 但是我们可以把 level 和 info 的值赋予其它变量以便以后读取。  
  
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.
```

```
// 和其他语言不同，由于 MFP 支持无穷大和无定义数，除以 0 不会
```

```
// 造成中断。
```

```
c = 3/0
```

```
Catch
```

```
print ("Divided by zero!\n") // will not be here
```

```
// 这一句不会被执行
```

```
EndTry
```

```
Endtry
```

```
Endf
```

用户运行这个程序，输入`::mfpxexample::testTryCatch()`并执行，会看到输出的结果为

```
a == 3
```

```
dbc is undefined
```

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

第 11 节 solve 和 slvreto 语句

Solve 语句开始了一个内联求解代数方程程序块。Solve 语句可以拥有任意个数的变量作为参数。这些变量必须预先声明。这些作为参数的变量将成为内联求解代数方程程序块的待解变量。在以下例子中，用户首先声明了三个变量 `x`, `y` 和 `z`，然后开始执行一个内联求解代数方程程序块以解出这三个变量的值。

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

```
solve x, y, z
```

```
...
```

待解变量 x, y 和 z 的初始值是什么无关紧要。如果 solve 程序块能够解出 x, y 或 z 的值, x, y 或 z 将被赋予这个新值, 否则, x, y 或 z 的值将保持为进入 solve 程序块之前的值不变。.

Slvreto 语句结束了一个内联求解代数方程程序块。它有一个可选参数。该参数必须为内联求解代数方程程序块之前声明的一个变量。该变量用于储存每一个待解变量的所有的根。用户需要调用系统提供的三个函数 get_num_of_results_sets 、 get_solved_results_set 、 get_variable_results 去获得所有变量的一组解或者某一个变量的所有的根。比如, 以下语句都是合法的:

```
slvreto
```

```
slvreto all_results
```

以下代码是一个内联求解代数方程程序块的完整示例。本实例可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到。

```
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.

// x, y and z 为待解变量

// 变量 a, b 和 c 也用于本内联求解代数方程程序块

// 但是, a, b 和 c 不是待解变量所以它们的值不会改变。

solve x, y, z

// Note that in the following equations

// == must be used instead of =.

// 注意必须使用 ‘==’ 而不是赋值符号 ‘=’ 。

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.

// 用于存放每一个变量的所有的根, 可以省略

print("\nx == " + x + "\ny == " + y + "\nz == " + z)

print("\nnumber of result sets is ")

// print number of result sets (打印解集个数)

print(get_num_of_results_sets(a))

// It is possible that solve block cannot solve the

// equations. If so, slverto 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 in a future edition.

    // 存在 MFP 无法解出方程的情况，如果是这样，内联求解代数方程

    // 程序块返回一个空值。注意，由于内联求解代数方程程序块返回值

    // 的格式可能会改变，所以一定要用系统提供的函数，也就是

    // get_num_results_sets。

    if (get_num_of_results_sets(a) > 0)

        // 0 means first result set, 1 is 2nd result set, ...

        // 0 意味着第一组解，1 是第二组解，依次类推。

        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.

        // 注意 get_variable_results 有两个参数，第一个是 solve 程序块

        // 返回的包含所有解的变量，第二个是待解变量在 solve

```

```

    // 语句中的位子。0 表示第一个待解变量，1 表示第二个，依次类推。

    // 也有可能一些变量能够解出但另一些变量无法解出，在这种情况下，

    // 在 solve 程序块返回值中，没有解出的变量的值为 NULL。

    // Now print all roots of y (打印 y 的所有根)

    print("\nAll roots of y are ")

    print(get_variable_results(a, 1))

else

    // cannot solve (解不出来)

    print("\nSorry, cannot solve x, y and z")

endif

return

Endf

```

在以上例子中，我们可以得到

$x = 0.14781939$

$y = 6.84549421$

$z = 29.66719489$

解集个数为 2

第二组解是 $[-3.38250623, -3.22386342, -10.25720306]$

y 的所有根为 $[6.84549421, -3.22386342]$ 。

如果想编写内联求解代数方程程序块，用户需要注意以下事项：

1. 在内联求解代数方程程序块有两种变量，普通变量和待解变量。待解变量为 `solve` 语句的参数。在上述例子里， x , y 和 z 为待解变量而 a , b 和 c 为常规变量。常规变量的值在内联求解代数方程程序块中是已知的而待解

变量的值是未知的。无论是哪一种变量，都必须在内联求解代数方程程序块之前预先声明。

2. 在内联求解代数方程程序块中的方程表达式必须使用 ‘==’ 而不是赋值号 ‘=’。但是，在内联求解代数方程程序块中程序员可以赋值给普通变量和待解变量（虽然不推荐这样做）。比如：

```
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
```

在上述例子中，如果我们删掉 $c = 7$ 这一行，我们所解的方程为 $3 * x + y / 5 == 9$ 和 $y * 4 - x * 5 == 6$ 。但如果我们保留 $c = 7$ 这一行，我们所解的方程则为 $3 * x + y / 5 == 9$ 和 $y * 4 - x * 7 == 6$ 。

如果用户给一个待解变量赋值，该待解变量将不再待解而自动转变为普通变量。这样做必须非常小心，因为赋值待解变量可能会影响到赋值语句之前的方程式。比如：

```
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
```

在上述例子中，如果我们删掉 $x = 7$ 这一行，我们所解的方程为 $3 * x + y / 5 == 9$ 和 $y * 4 - x * 5 == 6$ 。但如果我们保留 $x = 7$ 这一行，我们实际上解得方程为 $y * 4 - 7 * 5 == 6$ 。这是由于一开始，MFP 语言无法通过第一个方程式解出 x 或者 y ，所以，它继续分析第二个表达式。在这个表达式中， x 被赋值为 7，这样一来， x 和 a, b 以及 c 一样不再为待解变量，所以，第三个方程式变成了 $y * 4 - 7 * 5 == 6$ 。我们可以得到 y 的值为 10.25。然后 MFP 语言回过头来分析第一个方程式，这时， x, y 和 a, b 以及 c 一样都不再为待解变量，该方程式变成 $3 * 7 + 10.25 / 5 == 9$ 。这是一个比较表达式，返回值为 FALSE。所以，我们最后得到的解是 x 为 7 而 y 为 10.25。

3. 从上述例子中可以看到，每一个待解变量的所有根都存放在内联求解代数方程程序块的最后的 slvreto 语句的返回变量中。在当前的 MFP 语言中，slvreto 语句的返回变量实际上是一个 2 维矩阵，矩阵的每一列为所有待解变量的一个解集。但是，请注意在后续版本中 slvreto 语句返回变量的数据结构可能会发生改变。所以，用户必须使用系统所提供的函数，也就是 `get_num_of_results_sets`，`get_solved_results_set` 和 `get_variable_results` 等函数从 slvreto 语句的返回变量提取待解变量的根。所有待解变量在第一个解集中的根为待解变量在内联求解代数方程程序块运行完毕之后的值，除非在第一个解集中该待解变量无解。

很显然，MFP 语言内联求解代数方程的功能并非是万能的。过于复杂的代数方程（组）无法被 MFP 语言解出。在这种情况下，slvreto 语句返回空值，调用函数 `get_num_of_results_sets` 返回值为 0。而待解变量则都保持它们进入 solve 程序块之前的原始值不变。还有一种可能是一些待解变量能够被 MFP 语言解出，但另外的待解变量的值无法被解出。这时，无法被解出的待解变量保持它们的原始值不变。而如果我们使用函数 `get_solved_results_set` 或者 `get_variable_results` 从 slvreto 语句返回值中取回无解变量的值，我们取回的是 NULL。

4. MFP 语言内联求解代数方程所获得的解集个数为每个待解变量的解的个数的乘积。比如，

```
variable x, y, z
```

```
solve x, y
```

```
log(x) == 3
```

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

```
slvreto z
```

返 回 3 组 解 集 , 每 一 组 解 集 都 是 [20.08553692318766792368478490971028804779052734375, -1]。这是由于 $y**3 + 3 * y**2 + 3 * y+ 1 == 0$ 有三个相同的根。这样一来, 虽然 $\log(x) == 3$ 只有一个根, 但每一个 y 的根必须有一个 x 来对应, 所以我们最后得到 3 组解集。

第 12 节 help 和 endh 语句以及@language 标注

在 MFP 语言中, Help 语句是一个帮助信息块的开始而 endh 语句是一个帮助信息块的终止。

注意虽然在一个帮助信息块中的语句不会影响函数的运行, 当用户输入 "help 函数名" 或者 "help 函数名(函数参数个数)" 时帮助信息块却能提供必要的帮助信息。条件是该帮助信息块正好位于函数声明的上方。在以下例子 (可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到) 中一个帮助信息块位于函数 abcd 的上方:

```
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
```

。当用户输入 "help ::mfexample::testHelp" 或者 "help ::mfexample::testHelp(2)" 时，如果系统语言是英语并且英语也是默认语言，该用户将看到如下帮助信息：

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.

，如果系统语言是日语但是日语不是默认语言，该用户将看到如下帮助信息：

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.

，如果系统语言是简体中文但简体中文不是默认语言，，该用户将看到如下帮助信息：

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.

，如果系统语言是简体中文而且简体中文是默认语言，，该用户将看到如下帮助信息：

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.

。

如果用户仅仅只是想在一行的末尾加一些注释，可以使用类似于 C++ 的 “//”，在一行中，“//”之后的内容均为注释。“//”可以位于一行最开始。

第 13 节 citingspace 以及 using citingspace 语句

用户也许已经注意到，从 1.7 版开始，MFP 编程语言和以前有些不同。比如，在以前版本的用户手册中，示例程序的函数名都是以 mfpExample_开头，这样避免了 mfp 手册的示例程序和软件自带的程序或者用户自己定义的程序相互冲突。

从 1.7 版开始，用户手册的示例程序不再以 mfpExample_开头。比如，在上一节中的例子函数的名字，就是简单的 testHelp。那么，如果用户自己定义了一个 testHelp 函数，并且参数个数和例子函数 testHelp 相同，MFP 语言如何区分它们呢？

一个比较好的办法是对名字相同的函数使用不同的引用空间。当且仅当函数的引用空间相同，函数名相同以及参数个数相等，MFP 才认为它们是同一函数。比如，从本版开始，示例程序都在 mfpexample 的引用空间中。在示例源文件的头部均有 citingspace ::mfpexample 的语句，在示例源文件尾部均有 endcs 的语句。这就表明，所有的例子程序，都是定义在 ::mfpexample 引用空间内，在其他的引用空间内，即使定义了同名函数，也不会发生冲突。

引用空间本身是一个层级的概念，最上层的引用空间，名字是一个空字符串，如果一个 `mfps` 文件中没有定义任何引用空间，MFP 语言就会认为该文件中所有的函数均被定义在最上层的引用空间中。

在最上层的引用空间的内部（也就是下一层），用户可以定义另外的引用空间，比如一个名叫 `mfpexample` 的引用空间（引用空间名，和函数名一样，不分大小写，所以 `mfpExample` 和 `MFPexample` 以及 `mfpexample` 是等价的）。注意 `mfpexample` 虽然是该引用空间的名字，但是，如果用户想找到这个引用空间，必须使用该引用空间的绝对引用路径，绝对引用路径的表示方式是：

最上层引用空间名::下一层引用空间名::再下层引用空间名::...::目标引用空间名

。由于引用空间 `mfpexample` 位于最上层引用空间的下一层，所以，它的绝对路径是`::mfpexample`（注意最上层引用空间的名字是一个空字符串）

用户在执行 MFP 语言的例子函数时，则需要指出是哪个引用空间，比如以下函数定义在引用空间`::mfpexample` 中：

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

那么，上述函数的调用就应该是`::mfpexample::testCS1(a)`。换句话说，就是函数的绝对引用路径加上`::`再加上函数名和函数参数。

用户可能会问，在可编程科学计算器 1.6.7 及其以前版本中，没有引用空间的概念，函数调用就是函数名加上参数（比如 `testCS1(a)`），那么，可编程科学计算器 1.7 版如何保证和以前的版本兼容呢？

为了实现兼容的目的，MFP 语言中加入了基本引用路径的概念，也就是说，如果一个函数的引用空间位于一条基本引用路径上，那么调用该函数时，不用写明绝对引用路径。

1.7 版的 MFP 语言的基本引用路径为顶层引用空间和 mfp 引用空间和 mfp 所有的下级（或者更下级）引用空间。任何函数，如果定义在这些引用空间中，调用的时候不用写明绝对引用路径。

由于在旧版本的可编程科学计算器中，所有的 mfps 文件都没有定义引用空间，那么，这些 mfps 文件在新的 1.7 版中，将自动地被看成使用最顶级引用空间。那么在这些 mfps 文件中定义的函数，将被认为是最顶级引用空间中的函数，而这些函数的调用是不需要写明绝对引用路径的（当然如果写明，也是可以的）。

比如，用户在旧版的可编程科学计算器中，定义了函数 Myf_abc()，并且在其他的函数中调用了 Myf_abc。显然，调用的方式就是 myf_abc()。在新版本中，由于函数 Myf_abc() 是定义在最上层引用空间的，所以调用是不需要写明绝对引用路径，所以 myf_abc() 还是正确的。当然，如果用户在调用时写明绝对引用路径也就是 ::myf_abc()，也是可以的。

还要注意的是，对于软件自带的函数，在旧版中没有引用空间的概念，在新版软件中，自带函数的引用空间都位于引用空间 ::mfp 的一个子空间（下层或者更下层引用空间）中。比如，函数 tan(x)，在新版软件中位于引用路径为 ::mfp::math::trigon 的引用空间中，而函数 log(x) 在新版软件中位于引用路径为 ::mfp::math::log_exp 的引用空间中。无论系统自带的函数位于哪一个引用空间中，它们都是引用空间 ::mfp 的某一个子空间，所以，都是被包括在基本引用路径中，在调用的时候不需要指定引用路径。这样确保了和旧版软件的兼容性。

从 2.0 版开始，MFP 语言加入了对面向对象编程的支持。class 语句被引入。class 语句和 citingspace 保持兼容，每一个 class 就是一个 citingspace。每一个 class 内定义的静态函数可以用上面所给出的使用 citingspace 引用路径的办法访问。但是，如果不是静态函数，由于需要初始化对象，所以无法通过 citingspace 引用路径的办法访问。注意，当前 class 还不支持静态变量，所以变量无法通过 citingspace 引用路径的办法访问。比如下面的例子：

```
class ABC  
Function f0()  
    print("\nThis is ::ABC::f0()\n")
```

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

通过::ABC::f0() 调用静态函数 f0 是合法的，但无法通过::ABC::f1() 调用非静态函数 f1，原因是 f1 需要通过类 ABC 的对象来访问。

除了基本引用路径，用户也可以在基本引用路径之外自定义额外的引用路径。用户定义引用路径需要使用 using citingspace 语句，该语句指明哪条额外的引用路径还要被使用。比如语句 using citingspace ::abcd 就是告诉 MFP 语言除了最高层引用空间和::mfp 以及其子引用空间之外还要使用引用路径::abcd。

需要注意的是，在 citingspace 和 using citingspace 语句也可以使用相对引用空间。比如在引用空间::mfpexample 中，用户又定义了

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

那么，引用空间 Abcd 就是使用的相对引用路径，由于引用空间 Abcd 定义在::mfpexample 中（也就是说::mfpexample 是其直接上层引用空间），它的绝对引用路径就是::mfpexample::abcd。

又比如，在引用空间::mfpexample 中，用户声明了语句 using citingspace abcd，那么由于 abcd 不是绝对引用路径，MFP 语言将使用这条 using citingspace 语句所在的最内层的引用空间来构析 abcd 的绝对引用空间，所以，它的绝对引用空间就是::mfpexample::abcd。

还比如，用户在引用空间::mfpexample 中的某一个函数中，调用了 abcd::testCS(a) 函数，假如没有任何 using citingspace 语句在调用之前被声明，那么这个使用相对引用路径的 abcd::testCS(a) 的绝对引用路径就应该是::mfpexample::abcd::testCS(a)。

如果用户自己定义了好几层引用空间，并且在引用空间中和一些函数体内部声明了一些 using citingspace 语句，那么，如果在这些函数中调用其他的函数，MFP 语言对引用空间的查访顺序为：

1. 函数调用语句所在的直接引用空间（也就是最内层的引用空间）；
2. 函数调用语句所在的最内层的程序块中的 using citingspace 所指定的引用空间。注意这些 using citingspace 语句必须位于函数调用语句的上方，并且越靠近函数调用语句，在进行引用空间搜索时，就具有越高的优先级；
3. 函数调用语句所在的外层的程序块中的 using citingspace 所指定的引用空间，越往外层，using citingspace 语句所指定的引用空间的优先级就越低。和第 2 点类似，这些 using citingspace 语句必须位于函数调用语句的上方，并且越靠近函数调用语句，在进行引用空间搜索时，就具有越高的优先级；
4. 函数调用语句所在的直接引用空间内但是在函数体外的 using citingspace 所指定的引用空间。请注意些 using citingspace 语句必须位于函数调用语句的上方，并且越靠近函数调用语句，在进行引用空间搜索时，就具有越高的优先级；
5. 函数调用语句所在的非直接引用空间（也就是包含最内层引用空间的上层或者更上层引用空间，直到最顶层引用空间）和在非直接引用空间内的 using citingspace 语句所指定的引用空间。注意，在这里，这些引用空间的优先级是非直接引用空间 > 该非直接引用空间中的 using citingspace 语句所指定的引用空间 > 更上层的非直接引用空间 > 更上层的非直接引用空间中的 using citingspace 语句所指定的引用空间...。比如，假如用户在某个 mfps 文件中定义了如下引用结构：

```
Citingspace level1 //绝对引用路径为::level1
```

```
Using citingspace aaaa //绝对引用路径为::level1::aaaa  
Using citingspace bbbb //绝对引用路径为::level1::bbbb  
Citingspace level2 //绝对引用路径为::level1::level2  
Using citingspace cccc //绝对引用路径为::level1::level2::cccc  
Using citingspace dddd //绝对引用路径为::level1::level2::level3  
Citingspace level3 //绝对引用路径为::level1::level2::level3  
Function asmallfunc()  
endf  
Endcs  
Endcs  
Endcs
```

， 并且假设上述代码就是该 mfps 文件的全部， 那么，在函数 asmallfunc 看来，引用空间的优先级顺序为： ::level1::level2::level3 > ::level1::level2 > ::level1::level2::ddd > ::level1::level2::cccc > ::level1 > ::level1::bbbb > ::level1::aaaa > 最顶层引用空间。

6. 系统基本引用路径所指向的引用空间。

在这里需要注意几点。第一点，如果用户在非常靠近调用函数调用的位置用 using citingspace 语句指定了一个已经具有较低优先级的引用空间，那么该引用空间的优先级将会被提升。比如，在引用空间::aaaa 内，用户定义了某个函数，在该函数内，用户要调用另外一个函数，在调用另外那个函数之前，用户又用 using citingspace 声明了::bbbb 引用路径，代码结构如下：

```
Citingspace ::aaaa
```

```
.....
```

```
Function callsomething()  
.....  
Using citingspace ::bbbb  
Anotherfunc() // 这里调用函数 Anotherfunc()
```

.....

Endf

.....

Endcs

。显然，在此例子中，当调用 Anotherfunc() 函数时，MFP 语言搜索引用空间的顺序为（由高到低排列）：

1. ::aaaa
2. ::bbbb
3. 顶级引用空间
4. ::mfp 以及其下级引用空间

。如果这时用户在 Using citingspace ::bbbb 下面增加一条 using citingspace 语句（增加后的代码结构见下）：

```
Citingspace ::aaaa
```

.....

```
Function callsomething()
```

.....

```
Using citingspace ::bbbb
```

```
Using citingspace // 这条语句的意思是使用顶级引用空间
```

```
Anotherfunc() // 这里调用函数 Anotherfunc()
```

```
.....
```

```
Endf
```

```
.....
```

```
Endcs
```

，由于 Using citingspace 比 Using citingspace ::bbbb 更靠近函数调用语句 Anotherfunc()，所以顶级引用空间的搜索优先级这时比::bbbb 还要高。换句话说，顶级引用空间的搜索优先级被提升了。新的搜索引用空间的顺序为（由高到低排列）：

1. ::aaaa

2. 顶级引用空间

3. ::bbbb

4. ::mfp 以及其下级引用空间

。注意虽然顶级引用空间的搜索优先级被提升了，但是它仍然比::aaaa 的搜索优先级低。原因是::aaaa 是调用语句所在的直接（最内层）引用空间，直接引用空间总是具有最高的搜索优先次序。

第二点是程序块的概念。程序块是指一个有开始有结束完整的程序部分。比如 function 和 endf 之间的部分就构成一个程序块，if 和 endif （也包括 if 和 elseif，if 和 else，elseif 和 elseif，elseif 和 else，elseif 和 endif，以及 else 和 endif），for 和 next，while 和 loop，do 和 until，try 和 catch，catch 和 endtry，select 和 ends，这些语句对之间的部分都构成程序块。显然，程序块可以相互嵌套，比如：

```
If a == b
```

```
For idx = 0 to 10 step 1
```

```
Next
```

Endif

，就是一个 for 程序块嵌在一个 if 程序块内。注意在 MFP 语言中，外层程序块声明的 using citingspace 语句，只要在内层程序块的上方，内层程序块的函数调用都可以看到。当然，外层程序块中声明的 using citingspace 语句在引用地址中的搜索优先级总是比在内层程序块中声明的 using citingspace 语句的搜索优先级低。但是在内层程序块中声明的 using citingspace 语句，外层程序块中是看不到的。在上述例子中，如果在 if 程序块中声明 using citingspace ::aaaa，在 for 程序块中声明 using citingspace ::bbbb，那么 for 程序块中的引用地址搜索顺序是先 ::bbbb，再是 ::aaaa，但在 if 程序快中引用地址搜索顺序只包括 ::aaaa。

If a == b

Using citingspace ::aaaa // for 程序块看得见，但优先级低。

For idx = 0 to 10 step 1

Using citingspace ::bbbb // if 程序块看不见。

Next

Endif

第三点，引用空间在一个 mfps 文件中的优先级不影响另外一个 mfps 文件中的优先级。比如有两个 mfps 文件，一个文件的内容为：

Citingspace ::aaaa

Using citingspace ::bbbb

Citingspace ::cccc

Function funcA()

Endf

Endcs

```
Endcs
```

另外一个文件的内容为：

```
Citingspace ::aaaa
```

```
Citingspace ::cccc
```

```
Using citingspace ::bbbb
```

```
Function funcB()
```

```
Endf
```

```
Endcs
```

```
Endcs
```

那么在函数 funcA 看来，引用空间的优先顺序为::cccc > ::aaaa > ::bbbb，而在函数 funcB 看来，引用空间的优先顺序为::cccc > ::bbbb > ::aaaa。

第四点，虽然不同文件中的引用空间的优先级排序不会相互影响，但要注意定义在同一个引用空间中的各个函数，哪怕定义在不同文件中，只要该空间被引用，则它们都是可见的。比如，在上一个例子中，函数 funcB 定义在引用空间::bbbb 中，虽然函数 funcA 位于另外一个文件中，但由于在 funcA 之前已经声明使用引用空间::bbbb，所以 funcB 对于 funcA 是可见的，funcA 可以直接调用 funcB（只要没有同名并且同参数数目的函数）。

第五点，citingspace 除了在 mfps 代码文件中声明，不能在其他任何地方使用。而 using citingspace 语句可以在命令提示符和基于 JAVA 的可编程科学计算器中使用，但不能在智慧计算器中使用，在这些交互式的工具中，用户可以使用

```
Shellman list_cs
```

命令查看所有使用的引用空间，也可以使用

```
Shellman add_cs citingspace_path
```

把 citingspace_path 指向的引用空间添加到引用空间搜索列表中。这条指令和 using citingspace 具有一样的效果。用户还可以使用

```
Shellman delete_cs citingspace_path
```

命令把 citingspace_path 指向的引用空间从引用空间搜索列表中删除。

Using citingspace 语句（或者 shellman add_cs 命令）在命令提示符和基于 JAVA 的可编程科学计算器中使用时，将会添加新的引用空间搜索路径。但是这些新的引用空间搜索路径的优先级总是低于顶级引用空间（原因是顶级引用空间就是人机交互模式下命令提示符或者基于 JAVA 的可编程科学计算器的直接引用空间）。正是因为顶级引用空间在人机交互模式下在命令提示符或者基于 JAVA 的可编程科学计算器中具有最高的搜索优先级，所以，用户在命令提示符或者基于 JAVA 的可编程科学计算器中输入函数命令时，可以用省略掉起始::的相对引用空间路径。比如一个函数包括完整引用空间路径的名字为::abcdef::ghijk::lmn(a, b)，在命令提示符或者基于 JAVA 的可编程科学计算器中，如果用户想调用这个函数，可以输入并运行，比如 abcdef::ghijk::lmn(1, 2)，这和输入::abcdef::ghijk::lmn(1, 2)的效果完全一样。

最后需要指出的是 citingspace 和 using citingspace 语句不能在 solve 程序块中出现。

以下代码示例（可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到）可以帮助用户理解引用空间运作机制和搜索优先次序：

```
Citingspace ::mfpexample  
// 别的函数，和本章节不相干，不在此列出  
// .....,  
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.

    //注意这里调用 testCS2(a, b)时会出错，原因是 testCS2(a, b)定义在引用空
    // 间::mfpxexample::abcd 中。虽然 using citingspace abcd 已经被声明，但
    //由于 abcd 是相对引用路径而不是绝对引用路径，而本函数 (testCSRef)
    // 定义在引用空间::Abcd 中，这样一来，using citingspace abcd 语句
    // 所指向的引用空间的绝对引用路径就是::Abcd::abcd。但这个引用空间并
    // 不存在，而里面就更不会有 testCS2(a, b)函数了，所以会出错。

    testCS2(2, 3)

endf

using citingspace ::mfpxexample::abcd

Function testCSRef1()

    // here testCS2(a, b) is called after citingspace ::mfpxexample::abcd is
    // declared to use. So MFP will be able to find the right function
    // which is ::mfpxexample::abcd::testCS2

    //这里， testCS2(a, b)在 using citingspace ::mfpxexample::abcd 语句之后调
    // 用， 函数 testCS2(a, b)是定义在引用空间::mfpxexample::abcd 中，所以
    // 可以正确执行。

    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.

//这里，第一个 testCS2(a,b)函数（在 if 程序块内的 testCS2）在

// using citingspace ::mfpexample::abcd 语句之后调用，所以能够正确执行。

// 第二个 testCS2(a,b)函数在 for 程序块内，它能够看到上层程序块，也就是

// if 程序块的 using citingspace ::mfpexample::abcd 的语句，所以也能够正

// 常执行。但是最后一个 testCS2(a,b)函数在 if 程序块之外，它无法看到在

// if 程序块内的 using citingspace ::mfpexample::abcd 语句，所以无法正确

// 执行，会出错。

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(":mfpxexample::abcd is not declared to use out of if block\n")

    testCS2(2, 3)

endf
```

Function testCSRef2()

```
variable a = 3

// call ::mfpxexample::testCS1 because testCSRef2() is inside (both

// ::_efgh_ and) ::mfpxexample. Citingspace ::_efgh_ has a higher

// priority than ::mfpxexample. 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 ::mfpxexample. And there is. So ::mfpxexample::testCS1

// is called.

// 调用::mfpxexample::testCS1 函数。这是因为 testCSRef2() 函数既位于

// 引用空间::_efgh_ 中，也位于引用空间::mfpxexample 中。但是

// ，由于::_efgh_ 位于引用空间::mfpxexample 的内部，而 testCSRef2()

// 又位于::_efgh_ 内部，所以，引用空间::_efgh_ 具有更高的优先级，

// 如果在引用空间::_efgh_ 内有定义一个名叫函数 testCS1 并且只有

// 一个参数的函数，则这个函数将会被调用，但是由于::_efgh_ 中没有
```

```
// 这个函数，所以 MFP 会在更外层的引用空间中寻找，最后 MFP 在  
// 引用空间::mfpexample 中找到了这个函数，所以该函数被调用。  
  
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 still lower priority than the innermost  
    // citingspace declaration, in this case it is ::_efgh_). Thus  
    // ::abcd::testCS1 is called.  
  
    // 由于 using citingspace ::abcd 在最内层的程序块中被声明，它  
    // 具有比::mfpexample 引用空间更高的优先级（但是它的优先级仍然  
    // 比::_efgh_要低，原因是::_efgh_ 是 testCSRef2() 函数最直接  
    // 的引用空间）。所以，最后的结果是函数::abcd::testCS1 被调用。  
  
    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,
```

```
// ::mfpxexample::abcd::testCS1 is called.  
  
// using citingspace ::mfpxexample::abcd 语句位于下面 testCS1 函数调用  
// 的正上方，所以引用空间::mfpxexample::abcd 具有比除了::__efgh__ 之外  
// 所有其他引用空间都要高的优先级。由于引用空间::__efgh__ 中不包括  
// 一个叫 testCS1 并且只有一个参数的函数，::mfpxexample::abcd::testCS1  
// 函数将被调用。  
  
testCS1(a)  
  
endf  
  
endcs  
  
using citingspace ::mfpxexample::abcd
```

```
citingspace ::__efgh__  
  
Function testCSRef3()  
  
// function testCS2(a, b) is defined in the citingspace  
  
// ::mfpxexample::abcd and statement using citingspace ::mfpxexample::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.  
  
// 函数 testCS2(a, b) 被定义在引用空间::mfpxexample::abcd 中，并且，在  
// 外层引用空间::abcd 中（也就是并非最内层的直接引用空间::__efgh__，  
// 但引用空间::abcd 包含了::__efgh__）声明了
```

```
// using citingspace ::mfpexample::abcd, 而且这个声明是在本函数调  
// 用之前。所以，这里调用函数 testCS2(2,3)能够成功。  
testCS2(2,3)  
endf  
Endcs  
// 别的函数，和本章节不相干，不在此列出  
// .....  
endcs
```

用户在命令提示符或者基于 JAVA 的可编程科学计算器中输入并运行“`::mfpexample::testCS1(3)`”，将会得到：

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

，然后用户输入“`using citingspace :: mfpexample`”并运行，将会看到以下提示：

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

上述提示的意思是说，`::mfpexample` 已经被加入引用空间搜索路径了，它现在具有比除了顶级应用空间之外的其他所有引用空间都要高的搜索优先级。

用户再输入“`testCS1(3)`”并运行，将会得到：

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

，然后用户输入“`abcd:: testcs1(3)`”并运行，将会得到：

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

，然后再输入“`::mfpexample :: abcd :: testCS1(3)`”，将会看到：

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

，然后用户运行“abcd::testcsref()”，将会看到如下错误信息：

```
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!
```

，如果用户运行“abcd::testcsref1()”，则不会出错，运行结果是：

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

，用户运行“_efgh_::testCSRef1()”，将会看到结果为：

```
::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!
```

，运行“_efgh_::testCSRef2()”，结果为：

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

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

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

, 运行 “_efgh_::testCSRef3()” , 得到的结果为:

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

第 14 节 class 和 endclass 语句

1. 类的声明

class 和 endclass 语句定义了 MFP 语言的类。class 语句是类定义的开始。如果 class 语句中没有父类声明, class 就是直接派生于 MFP 语言的最基本的 object 类型。这样的 class 语句的例子如下:

```
class Class_Name
```

反之, 如果类是从一个或者多个父类中直接派生而来, class 语句则如下面的例子所示:

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

这里, 直接派生是指该类是父类的子类, 而不是孙类, 曾孙类甚至更后辈。

还需要注意, 在 class 语句中, 父类的名字是可以包含完整或者部分的引用空间路径的, 比如:

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

上述语句是完全合法的。MFP 将根据当前引用的所有引用空间以及它们的优先次序来找到每一个父类。

2. 嵌套的类和类的成员

类可以定义在另外一个类的里面。这种类被称为嵌套的类。在这种情况下，被嵌入的类仅仅只相当于嵌套类的引用空间。比如，类 A 定义在引用空间::AAA::bbb 中，而一个嵌套的类 B 定义在 A 中，则类 A 的完整引用空间路径为::AAA::bbb::A 而类 B 的完整引用空间路径为::AAA::bbb::A::B。除了它们引用空间路径上的相似，嵌套类和被嵌入的类是相互独立的。此外，嵌套类总是对外部可见的。

类的成员则包括函数和变量。它们分为两大类，私有（使用 private 关键字，只有类成员函数可访问，外部不可见）和公有（使用 public 关键字，外部可见），比如：

```
public variable self memberA = 7, memberB = "Hello", memberC  
private function memberFunc(a, b, c)  
...  
endf
```

需要注意的是，如果声明类成员时既没有使用 private 也没有使用 public 关键字，那么该成员被视为公有，也就是 public。

类成员变量的声明和普通变量的声明略有不同。首先，如上面指出的，类成员变量的声明语句前面可以加上 private 或者 public 关键字；其次，在 variable 关键字之后，必须加入一个 self 关键字。这个 self 关键字意思是该语句声明的变量不是静态变量。需要注意的是，在现阶段，MFP 不支持静态变量，所以，如果类成员变量的声明语句没有 self 关键字将会被忽略；最后，和函数内的变量一样，类成员变量可以在声明语句中被初始化。但是，MFP 只允许用存粹的值而不是函数来初始化类成员变量。比如：

```
variable self varA = [[1, 2]]
```

是对的而

```
variable self varA = func(3, 4)
```

则会出错。更多的类成员变量声明语句示例如下：

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

```
private variable self varD
```

和成员变量一样，类的成员函数声明语句前面可以加入 public 或者 private 关键字。此外，如果函数的第一个参数是 self，这个函数就不是静态的，否则，这个函数就是静态的。在函数的内部，使用 self 关键字再加一个点可以访问类的成员，比如：

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

    self.MemberA = a

    self.MemberB = b

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

endf
```

如果类的成员函数是静态的，它显然就不能访问类的非静态成员。一个静态成员函数的例子如下：

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

    return a+b+c

endf
```

虽然 self 关键字可以用于访问类的成员和类的父类（包括父类的父类和更上辈的类）的公共（public）成员，但如果类和它的某些父类拥有同样名字的成员变量或者相同声明的成员函数，self 关键字则只能够访问本类中的成员而无法访问父类中的成员。比如：

```
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)
        // 调用 ChildClass 的 memberFunc 函数，而不是 SuperClassA 或者
        // SuperClassB 的对应函数。
        return self.memberFunc(3)
    endf

endclass

```

为了访问父类成员，则必须使用 `super` 成员变量。这个成员变量是一个数组，第一个元素是第一个父类的对象，第二个元素是第二个父类的对象， \dots ，以此类推。注意这里的对象都是切片的对象，也就是说通过 `super` 返回的父类的对象是本类对象的一部分。这样一来，如果在上面的示例中开发者想调用父类的 `memberFunc` 函数，代码应该这样写：

```

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)
        // 调用 SuperClassA 的 memberFunc 函数
        variable x = self.super[0].memberFunc(3)
        // 调用 SuperClassB 的 memberFunc 函数
        variable y = self.super[1].memberFunc(4)
        return x + y
    endf

endclass

```

注意，如果一个类的声明语句没有包括任何父类，那么该类有一个唯一的父类，就是 MFP 语言的 object 类型。在这种情况下，self.super[0] 返回被切片的 object 对象。

在 MFP 语言中，成员函数和成员变量均可以被覆写（override）。如果 MFP 通过 self 关键字访问一个成员（函数或者变量），而该成员已经被继承树上的多个类覆写，那么它总是访问的最“下层”的成员。比如，如果类 A 是从类 B 派生而来，A 和 B 均有一个成员变量叫 C，并且我们已经定义了如下函数：

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

那么，如果一个 A 的对象作为参数被传入这个函数，那么 A 的成员变量 C 的值将会被打印出来。而如果一个 B 的对象作为参数被传入这个函数，那么 B 的成员变量 C 的值将会被打印出来。

但是，上面的规定有时候会给开发者带来困惑。比如，在上面的例子中类 B 有一个公共成员函数，它读取成员变量 C 的值。类 B 的开发者并不知道别的程序员会从类 B 派生出类 A，所以他假定成员变量 C 一定是类 B 的成员变量 C。但如果第三个开发者创建了一个类 A 的对象，并且调用了该对象从类 B 那里继承的读取成员变量 C 的值的成员函数，那么这时该成员函数实际上读取的是类 A 而不是类 B 的成员变量 C 的值。

```
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
```

如果类 B 的开发者想要确保在类 B 的成员函数中读取的是类 B 的成员变量 C，那么就必须通过类的 this 成员变量访问成员变量 C。所有的类都有一个 this 成员变量，该变量返回当前函数所在的类的一个（被切片的）的对象。一个例子如下：

```
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
```

和公共的 super 成员变量不同的是，this 是私有的。

3. 构造函数和魔术函数

如果开发者想要以一个 MFP 类作为模板创造一个对象，必须调用构造函数。和其他的编程语言不同，MFP 类的构造函数是内置的，不能够开发者自定义，不能被重载，也不能够被覆写。构造函数没有参数。它所做的一切工作就是根据成员变量的声明初始化成员变量。如果一个成员变量在声明中没有给出初始值，那么它就会被初始化为 NULL。构造函数返回值是该类的一个对象。构造函数的一个例子如下。在该例子中，类 Abcd 定义在引用空间::AAA::bbb 中，这个类的构造函数就是 Abcd()，包括引用空间路径的完整名称为::AAA::bbb::Abcd()。

```
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
```

由于开发者无法自定义构造函数，所以自定义的初始化步骤应该放在公共的成员函数中。这种成员函数的名称，返回类型和参数都可以由开发者来决定，但是 MFP 语言推荐使用`_init_`作为函数名，并且函数返回值为该对象自身。注意`_init_`只是一个普通的成员函数，而并非一个魔术函数，它可以被重载（overload），也可以在任何时候被多次调用。一个例子如下：

```

cittingspace ::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

// __init__函数和其他用户自定义的普通成员函数一样，它可以被重载

public function __init__(self, a, b, c)

    self.a = a

    self.b = b

    self.c = c

    return self

endf

endclass

```

```
function printABC()
```

```

using citingspace ::AAA::bbb

variable obj = abcd().__init__()

obj.__init__(3, 2, 1)

obj.__init__([5,4], [2,3], "WWW").printMembers()      // self.a = [5, 4]
self.b = [2, 3] self.c = WWW

endf

```

MFP 的类还提供了一些内置的魔术函数。这些魔术函数可以被用户自定义相同声明的函数所覆写。函数`__to_string__`是最常用的魔术函数。这个函数吧一个对象转换为字符串。当把一个对象和字符串相加，或者以这个对象为参数调用 MFP 内置的`to_string`函数时，这个函数将会被调用。

函数`__deep_copy__`返回该对象的深度拷贝。当以这个对象为参数调用 MFP 内置的`clone`函数时该魔术函数将会被调用。

函数`__equals__`判断该对象是否和另外一个变量的值相等。当使用`==`操作符并且操作符的左操作数为该对象时该魔术函数将会被调用。

函数`__hash__`返回该对象的哈希值。当以这个对象为参数调用 MFP 内置的`hash_code`函数时该魔术函数将会被调用。

函数`__copy__`返回该对象的浅拷贝。它的默认行为是创建一个新的对象，但是该对象成员变量均引用旧的对象的对应值。

函数`__is_same__`判断该对象是否和另外一个变量的值相同（注意不是相等）。这个函数仅仅判断两个对象的引用是否一样。所以，实际上它也和未重载的操作符`==`的功能是一致的。这个函数在开发者覆写`__equals__`函数，需要直接比较引用时很有用。MFP 不推荐覆写这个函数。

以下例子给出了上述函数的使用方法：

```

class SampleClass

variable self a = 1, b = 2

public function __equals__(self, o)

```

```
        print("User defined __equals__\n")

        if self.__is_same__(o) // 判断 self 和 o 是否指向同一个对象

            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

endif

public function __hash__(self)

    print("User defined __hash__\n")

    return a + b * 19

endif

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)      // 将会输出 Class SampleClass, a = 1 b = 2

    variable obj2 = clone(obj1)      // 将会输出 User defined __deep_copy__

    print("obj1 == obj2 is " + (obj1 == obj2))      // 将会先输出 User defined
    __equals__, 然后再输出 obj1 == obj2 is true

    print(hash_code(obj2)) // 将会先输出 User defined __hash__, 然后再输出 39

endf

```

第 15 节 call 和 endcall 语句

call 和 endcall 语句定义了 MFP 语言 call 程序块的边界。call 程序块是一段不在本线程中而是在别的线程中执行的指令。call 语句是 call 程序块的开始。在 call 语句中，call 关键字后面紧跟着连接对象或者 local 关键字，然后是 on 关键字，最后是一串 call 程序块的参数变量。call 程序块的参数变量都是在 call 语句之前就已经声明的普通的变量。每一个参数变量都只能被一个 call 语句所使用。endcall 语句标志着 call 程序块的中止。endcall 语句有一个可选参数。该参数是 call 程序块在本地进程的返

回变量。返回变量也是在 call 程序块之前声明的普通变量。返回变量只能被一个 call 程序块所使用而且不能同时作为 call 程序块的参数变量。

我们可以把一个 call 语句看作是一个函数的开始。与普通函数不同的是，call 程序块并非是在本地线程中运行，而是在另外一个线程中运行。运行 call 程序块的线程可以是本进程、本机的另外一个进程、或者是另外一台设备。

如果 call 程序块是在本进程中运行另外一个线程，call 关键字后面紧跟着的并非一个连接对象，而是关键字 local。在这种情况下，由于 call 语句仍然在本进程空间中运行，本进程空间中的任何变量对于 call 程序块来说都是可读可写的，所以 on 关键字和其后的一串 call 程序块的参数变量不起任何作用，可以省略。call 程序块生成的线程在读写本进程空间中的变量时，操作是原子性的，也就是说，只有一个线程对变量的值修改完成之后，另外一个线程才能够读，反之亦然。

如果 call 程序块是在另外一个进程中运行，运行 call 程序块的进程通过 call 语句中的连接对象和本地进程连接。运行 call 程序块的进程可以看到 call 程序块参数变量的值的变化，也可以修改 call 程序块参数变量的值。call 程序块对自己的参数变量的值的修改会反应到本地进程。但是需要注意的是，MFP 语言并不保证本地进程和 call 程序块进程对程序块参数变量的值的修改会被实时同步到对方，也不保证按修改的顺序进行同步传递。MFP 语言唯一保证的是在一个进程内对一个 call 程序块参数变量的修改是原子性的，也就是只有上一次修改完成了之后，对值的新的修改，不管是来自客户端还是服务器端，才能开始。需要注意的是这个原子性只是应用于一个进程。由于 call 程序块进程和本地进程都有一份程序块参数变量的拷贝，对于一个程序块参数变量的两份拷贝在不同的进程中同时进行修改不违背修改的原子性。其他任何除了返回变量和 call 序块参数变量的任何变量，call 程序块和本地进程都有自己独立的拷贝，在一方修改变量值不会影响到另一方的同名变量的值。

由于对 call 程序块能够实现（跨进程）变量操作的原子性，所以不论 call 程序块是在本地进程空间还是在另外的进程，都可以通过变量锁（比如调用 suspend_until_cond 函数）进行（跨进程）线程同步。这一特性是 MFP 语言的一个重大优势。

当 call 程序块遇到 endcall 语句或者 return 语句时停止运行并返回。如果 return 语句返回一个值，位于本地进程的 endcall 语句将收到返回值并将返回值赋给 call 程序块在本地进程的返回变量（如果 endcall 语句声明了返回变量的话）。

需要注意的是，不同于 call 程序块的参数变量，call 程序块的返回变量采用的是阻塞模式。换句话说，当 call 程序块被发送到远端执行后，任何在本地进程读取 call 程序块的返回变量的值的语句都将被阻塞，直到 call 程序块返回（不管有没有返回值）为止。

以下例子展示了 call 程序块在本地进程中如何开启一个新线程：

```
variable a = 3, b = 4

//由于仍然在本地进程空间中，所有的变量都可以直接读写，所以无需使用 on 关键字以及
//其后的参数变量

call local

    a = "HELLO"

    suspend_until_cond(a) //阻塞 call 程序块所在线程直到 a 的值发生变化

    //暂停 call 程序块所在线程，以便让启动线程阻塞在 suspend_until_cond 函数

    sleep(1000)

    b = 24 //将 b 的值设置为 24，启动线程才能摆脱阻塞状态继续运行

endcall

//暂停启动线程 1 秒，以便让 call 程序块所在线程启动并阻塞在 suspend_until_cond 函数

sleep(1000)

a = 9 //修改变量 a 的值，call 程序块所在线程得以运行

suspend_until_cond(b, false, "==", 24) //启动线程阻塞在变量 b，直到变量 b 的值等于 24

print_line("a = " + a + " b = " + b) //现在 a 和 b 的值都已经在 call 程序块中更新了
```

以下则是 call 程序块应用于不同进程一个实例：

```
variable local_interface, remote_interface, ret

//客户端（本地进程）地址

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")

//服务器端（运行 call 程序块的进程）地址

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

//从客户端连接到服务器端

ret = ::mfp::paracomp::connect::connect(local_interface, remote_interface)

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

//connect 函数的返回值是一个基于数组的字典，“CONNECT”关键字所对应的就是连接对象
//的定义。如果 connect 函数失败，“CONNECT”关键字对应的值为 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 // 变量 d 用于同步锁

// 只有变量 a, b 和 d 在 call 程序块中的赋值修改对启动线程可见，其他变量在 call 程
// 序块中也可以被赋值修改，但启动线程看不见

call conn on a, b, d

    print("Before suspend_until_cond(d, false, \"==\", 888), d = " + d + "\n")

    suspend_until_cond(d, false, "==", 888) // 等 d 的值变成 888，程序继续运行，
// 否则程序阻塞在这里
```

```
    print("After suspend_until_cond(d, false, \":\", 888), d = " + d + "\n")  
  
    sleep(5000) // 暂停 call 程序块所在线程，以便让启动线程阻塞在  
    suspend_until_cond 函数  
  
    d = 213 // 改变 d 的值，启动线程应该能够收到 d 的新值。启动线程收到新值之后  
    才能摆脱阻塞状态继续运行  
  
    // 再暂停 call 程序块所在线程。此时启动线程应该已经到达 print_line("c = " + c)  
    语句，  
  
    // 但由于 call 程序块所还未返回，启动线程无法读取返回值 c 所以被再次阻塞  
  
    sleep(5000)  
  
    a = 88  
  
    b = "KIL"  
  
    return 54  
  
endcall c  
  
  
sleep(10000) // 暂停启动线程 10 秒钟，以便让 call 程序块所在线程启动并阻塞在  
suspend_until_cond 函数  
  
d = 888 // 设置 d 的值为 888，call 程序块所在线程将会收到该新值  
  
suspend_until_cond(d) // 线程阻塞在此，只有 d 的值发生变化才会继续运行  
  
print_line("New value of d is " + d) // 我们必须先取回 c 的值。c 的值能够取回方才意  
味着 call 程序块已经返回  
  
// 线程阻塞在读取变量 c 的值时刻。只有当 c 的值从 call 程序块返回线程才能继续运行  
  
// 当 c 的值取回之后，我们可以打印出 a 和 b 的值。可以看到这时 a 和 b 的值已经发生了  
更改。如果我们在 print("c = " + c) 语句之前打印 a 和 b 的值，  
  
// 我们可能无法观察到 a 和 b 的值发生了变化  
  
print_line("c = " + c)  
  
print("a = " + a + " b = " + b)
```

```
close_connection(conn) //关闭连接
```

```
close_local(local_interface) //关闭本地通信协议界面
```

以上代码是由客户端进程所执行，在服务器端，我们需要运行以下代码接收连接请求并运行 call 程序块：

```
variable local_interface, ret
```

```
local_interface = ::mfp::paracomp::connect::generate_interface("TCPIP",
"192.168.1.107") //服务器端（运行 call 程序块的进程）地址
```

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

```
print("initialize_local ret = " + ret + "\n")//监听连接请求。监听线程将在后台工作。
```

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

```
print("listen ret = " + ret + "\n")
```

```
//下面这条 input 语句将阻塞程序的运行。如果服务端代码是一个简单的 MFPS 脚本并且是在 bash 或者 Windows 命令提示符中运行，input 语句可以阻止服务器程序的退出所以是必
```

```
//不可少的。但是如果是在安卓或者 MFP 语言的 JAVA 界面程序中运行，只要安卓应用或 JAVA 界面程序不退出 input 语句就是不必要的。因为这种情况下服务器端的进程并没有中止。
```

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

先运行以上服务器端代码，然后在不同的设备中运行客户端代码。在运行之前需要确保客户端和服务器端的地址是正确的。开发者可以看到在服务器端两条消息被打印出来，一条是 Before suspend_until_cond(d, false, "==" , 888), d = 27，另一条是 After suspend_until_cond(d, false, "==" , 888), d = 888。在客户端会打印出变量 a, b, c 和 d 的新值，其中，call 程序块的返回变量 c 的新值是一个基于数组的字典，call 程序块的返回值 54 位于该字典中。

第 16 节 @compulsory_link 标注

当用户将一个函数编译创建一个 APK 包的时候，可编程科学计算器不是拷贝所有用户自定义的 mfps 代码文件而仅仅抽取相关的代码。在某些时候，比如调用 integrate 或者 plot_exprs 函数时，函数参数是一个字符串或者基于字符串的变量。这样一来，可编程科学计算器在编译的时候无法判断哪些函数在运行时将会被调用。用户在这种情况下需要在代码中，最好在 integrate 或者 plot_exprs 函数调用语句的前一行，增加一个注释指令 @complusory_link 告知可编程科学计算器哪些用户自定义的函数需要链接入 APK 包。比如

...

```
@compulsory_link           get_functions(":mfpexample::expr1",
":mfpexample::expr2(2)")

integrated_result = integrate(expression_str, variable_str)
```

...

在上面例子中，::mfpexample::expr1 和 ::mfpexample::expr2 是用户自定义的函数。它们在运行时将会被 integrate 函数调用用于计算积分。所有的名字叫做::mfpexample::expr1 的函数都会被链接入 APK 包。但是对于名字叫::mfpexample::expr2 的函数，当且仅当它正好有两个参数或者有可选参数时，该函数才会被链接入 APK 包。

如果用户想把所有自己定义的函数链接入 APK 包，请用如下语句

```
@compulsory_link get_all_functions()
```

。这样一来，可编程科学计算器将链接所有函数。但是用户创建的应用装载速度会比较慢。并且，用户还必须保证所有的函数都必须已经定义，否则打包时会出现编译错误。

注意 get_functions 和 get_all_functions 实际上都是 MFP 语言内置的函数，但是和普通的 MFP 语言自带函数不同，这两个函数位于 mfp_compiler 引用空间中。该引用空间只会在打包 APK 的时候被加入，在其它情况下用户看不到该引用空间内的函数，除非给出完整的函数引用路径。

还要注意@compulsory_link 指令必须位于一个函数的内部，如果它在 function 语句前面或者 endf 语句之后，它不会有任何作用。

第 17 节 @build_asset 标注

当使用 MFP 语言开发游戏或者一些需要使用声音或者图像的应用时，MFP 程序需要读取和使用声音或者图像文件。这些文件被称为资源文件。如果 MFP 脚本只是在本地存储，也就是硬盘或者 ROM 中运行，开发者只需要将资源文件放置在脚本所在目录或者任何其他目录，然后在代码中指出资源文件的完整路径便可。但是，MFP 脚本可以被打包成安卓应用，还可能被发送到远端的沙盒中运行（这里的沙盒是 MFP 并行计算的概念，指的是在远端运行的 MFP 会话）。当 MFP 脚本被打包或者发送到远端时，其相关资源文件也必须被打包或者发送到远端。所以，开发者必须使用@build_asset 标注告诉 MFP 资源文件在打包后或者发送至远端沙盒后的新的位置，然后在代码中针对不同的情况告诉 MFP 在运行代码时如何找到资源文件。

以下代码段演示了如何正确地将资源文件（即 food.png）复制到目标位置，以及如何在运行时加载资源文件。请注意，作为标注，@build_asset 语句在编译时执行，即当我们从 MFP 脚本构建 APK 包或 MFP 脚本正在发送源代码和资源文件到远程设备时执行。另请注意，@build_asset 语句很长，因此使用 MFP 的换行符（空格后跟下划线字符）将其分为三行。

@build_asset 语句调用函数 iff 来处理三种情况不同的情况。首先是编译发生在远程会话（MFP 术语中的沙盒）中。当远程会话启动另一个远程会话，从而需要将资源文件传输到新的远程会话时，可能会发生这种情况。在这种情况下，函数 is_sandbox_session() 返回 true，并且资源文件必须位于临时目录的资源子文件夹中，该资源子文件夹的路径由函数 get_sandbox_session_resource_path() 返回。第二种情况是在 MFP 应用程序中进行编译。当 MFP 应用启动远程会话并准备传输资源文件时，就是这种情况。在这种情况下，函数 is_mfp_app() 返回 true。同样，如上所述，这种情况下的源路径不是字符串，而是一个三元素数组。数组的第一个元素为 1，表示源资源文件位于 Android 应用的 APK 中。第二个元素是一个函数调用，即 get_asset_file_path("resource")，它返回 Android 应用程序 assets 资料夹中 resource.zip 文件的路径。最后一个元素是源资源文件在 android 应用程序 assets 资料夹中 resource.zip 压缩包中的压缩路径。第三种情况是，当 MFP 脚本在 JVM 或 Android 作为独立脚本（即，不是作为

Android 应用程序) 运行时进行编译。因为在此示例中，资源文件在这种情况下与源脚本位于同一文件夹中，所以调用函数 `get_src_file_path()` 返回源脚本的完整路径，然后调用函数 `get_upper_level_path` 获取包含源脚本和资源文件的文件夹的路径。要想获取 `iff` 函数的详细信息，可以在命令行中键入 `help iff` 并回车。

```
@build_asset
copy_to_resource(if(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
```

`@build_asset` 语句之后的代码在运行时执行。相似地，这些代码也考虑了三种情况。第一种情况是在远程会话也就是 MFP 术语中的沙盒中运行。在这种情况下，`food.png` 文件位于临时目录的资源文件夹中的名为 `images` 的文件夹中。资源文件夹的路径由函数 `get_sandbox_session_resource_path()` 返回。第二种情况是作为 MFP 应用程序运行。在这种情况下，资源文件（即 `food.png`）位于应用程序 `assets` 资料夹的 `resource.zip` 文件中。函数调用 `get_asset_file_path("resource")` 返回 Android 应用程序 `assets` 资料夹中 `resource.zip` 文件的路径。`"images/food.png"` 是源资源文件到 Android 应用程序 `assets` 资料夹中 `resource.zip` 压缩包的压缩路径。第三种情况是该游戏作为独立脚本在 JVM 或 Android 上运行。在此示例中，在这种情况下，资源文件与源脚本位于同一文件夹中，所以调用函数

`get_src_file_path()` 返回源脚本的完整路径，然后调用函数 `get_upper_level_path` 获取包含源脚本和资源文件的文件夹的路径。请注意，仅在第 2 种情况下，即游戏作为 Android 应用程序运行，资源文件才作为 zip 条目保存在 zip 压缩包中。在其他两种情况下，资源文件是硬盘或 ROM 中的普通文件。因此，在第 2 种情况下，调用函数 `load_image_from_zip`，而在其他两种情况下，调用函数 `load_image` 来加载图像。要获取这两个函数的详细使用信息，只需在 MFP 命令行中输入 `help load_image_from_zip` 和 `help load_image` 即可。

还要注意`@build_asset` 指令必须位于一个函数的内部，如果它在 `function` 语句前面或者 `endf` 语句之后，它不会有任何作用。

第 18 节 @execution_entry 标注

正如前面的章节所述，MFP 脚本能够像其他任何脚本一样被执行。但是用户需要在脚本的头部增加一个`@execution_entry` 标注，告诉 MFP 语言解释器如何运行该脚本。

`@execution_entry` 标注的语法为

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

这里，`function_name` 是（包含或部分包含或者不包含 `citingSpace` 路径的）函数名，由于`@execution_entry` 标注位于任何 `citingSpace` 和 `using citingSpace` 语句之前，寻找函数时 MFP 解释器仅仅只搜索默认 `citingSpace` 的搜索路径（比如`::` 和 `::mfp`）。所以，如果完整的 `citingSpace` 路径没有给出，用户需要保证 MFP 解释器仍然能够找到该函数。此外，调用的入口函数未见得必须是该脚本文件中所定义的函数。它可以是其它脚本文件所定义的函数，甚至还可以是 MFP 的内建函数。

`param_string1, param_string2, ...` 是入口函数的参数。注意这些参数的写法和 MFP 调用函数时参数的写法基本一样，唯一的不同是`@execution_entry` 所需要的函数参数包含两种占位替换符，`#` 和 `@`。比如，在以下语句中，入口函数是 `create_file`。该函数包含两个参数，第一个参数是基于字符的文件名，第二个参数是一个布尔值。在`@execution_entry` 语句中，“`Date_`” + `@` 的意思是当 MFP 解释器从命令提示符中运行脚本时，脚本文件名之后的第一个参数被当成一个字符串，并且被添加到字符串

"Date_"的尾部，成为 create_file 函数所生成的文件的文件名。注意这里用户不能直接把第一个参数写成"Date_@"因为占位替换符在双引号中就不再具有占位替换的功能而变成了一个普通的字符。而#的意思是，当 MFP 解释器从命令提示符中运行脚本时，脚本文件名之后的第二个参数被当作一个数值，@execution_entry 语句将求取该参数的值并转换为一个布尔量。

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

这样，当用户运行脚本文件时（假设脚本文件的文件名是 myscript.mfps），如果使用下述命令

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

MFP 解释器将会调用以下 MFP 函数语句

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

。采用这种占位替代字符的办法，而不是其他脚本语言常用的\$args 变量，所获得的好处是显而易见的。首先，它避免了大量用于解析脚本文件参数的代码，节约了大量开发时间；其次它可以有效地区分字符串和数值；第三它和代码中的控制字符（比如，逗号，方括号，圆括号等）完全兼容；最后它不会破坏 MFP 语言的 citingspace——function 结构。比如，假设用户想要使用一个数组作为参数，该数组中的元素既包含数值，也包含字符串，那么用户只需要在脚本文件头部声明，比如如下的语句

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

然后在命令提示符中，用户输入

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

，MFP 解释器就能够自动执行

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

。需要注意的是，在系统的命令行窗口中运行命令，命令的参数直接是用空格分隔开的。所以在上述例子中，77+38.44i 中间不能存在任何空格，否

则空格之前的部分将会被用于替换第一个占位替换符#，空格之后的部分将会被用于替换第二个占位替换符@，整个函数调用就出错了。

@execution_entry 也支持可选参数。在参数列表中，位于尾部的@... 或... 表示存在可选参数并且它们都是字符串，而位于尾部的#... 则表示存在可选参数并且它们都是数值。需要强调的是，@execution_entry 语句中只能声明一次可选参数并且它必须位于语句的最后，仅仅在参数列表的终止符，也就是反圆括号，之前。比如

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

或者

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

的意思是用户调用该脚本文件时，需要至少文件两个参数，第一个参数被当作一个数值，第二个参数被当作一个字符串。如果多于两个参数，后续参数都被当作字符串处理。而

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

的意思是户调用该脚本文件时，需要至少文件两个参数，第一个参数被当作一个数值，第二个参数被当作一个字符串。如果多于两个参数，后续参数都被当作数值处理。

@execution_entry 语句也可以不申明任何参数，比如

```
@execution_entry func1
```

的意思是，用户在运行该脚本时，可以提供任何数量的文件参数，并且这些参数都会被当作字符串。当然，前提条件是 func1 函数确实需要那么多字符串参数。否则，如果 func1 的函数定义和调用命令不匹配 MFP 脚本解释器将会报错。

如果一个 mfps 脚本包含有正确定义的@execution_entry 语句，那么在安卓中，用户打开可编程科学计算器的文件管理工具，长按该 mfps 脚本图标，可以运行该脚本。如果该脚本需要参数，可编程科学计算器会自动弹出对话框让用户输入参数，参数间用空格隔开。如果是在 Windows 中，用户可

以通过适当的设置将该脚本变为一个可执行文件。操作步骤为鼠标右键点击该脚本，选择打开方式，然后将 `mfps` 文件关联到 `AnMath` 目录下的 `mfplang.cmd` 程序即可。但要注意这种方式不支持运行时输入文件参数。如果是在 Unix/MacOSX/Linux/Cygwin 中，首先用户需要调用 `chmod 777 mfps` 文件名将该脚本设置为可执行，然后在 `/usr/bin` 目录下建立一个软链接链至 `AnMath` 目录下的 `mfplang.sh` 程序，最后还需要在脚本的第一行加入 shebang 声明：

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

。进行完上述操作后，该脚本文件就可以如同 `bash` 脚本一样可执行了。并且用户还可以传递文件参数给该脚本。

当然，标准的 shebang 声明应该是

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

但为了支持这种声明方式，用户需要对操作系统的环境变量进行设置。这是一个操作系统的使用方法的问题，在这里不进行详述。如果用户有兴趣，可以参见一些 Unix/Linux 使用指南之类的 IT 教材。

第 19 节 如何部署用户创建的 MFP 函数程序

用户需要遵循以下步骤以实现和使用自己所编写的函数：

1. 启动可编程科学计算器；
2. 打开程序编辑器编写代码并保存。

很显然，在手机上敲代码不是一件容易的事情，一个比较快捷的办法是：

1. 将您的安卓设备通过 USB 电缆连接到个人电脑上；
2. 将您的安卓设备的存储卡设置为可读写；
3. 在个人电脑上找到您的安卓设备存储卡（包括 SD 卡和设备自带的存储器）所对应的文件夹或盘符，找到其中的 `AnMath` 目录，进入 `AnMath/Scripts` 子目录。在这个目录下创建一个 `.mfps` 文件，比如 `my_prog.mfps`；

4. 用个人电脑上的文本编辑器编辑 my_prog.mfps，比如，在该文件中写入以下内容（本函数可以在本手册自带的示例代码所在目录中的 MFP fundamental 子目录中的 examples.mfps 文件中找到）：

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

    Variable avg_value

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

    Return avg_value

Endf
```

然后保存，再将移动设备和个人电脑断开；

5. 打开可编程科学计算器；

6. 打开命令提示符，敲入：

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

然后回车，会看到函数返回 0。用户也可以先输入 using citingspace ::mfpexample 语句并回车，然后敲入 myfunc(1, 2, 3, 4)，可以看到同样的结果。

您需要注意的是：

1. 如果是在个人电脑上输入函数，用户可能需要在断开移动设备和个人电脑连接之后重新启动可编程科学计算器，否则，新编的函数有可能不被载入；
2. 函数必须申明，也就是必须有如下定义：

```
Function XXXX(...)
```

```
Endf
```

否则函数无法被找到；

3. 在 1.6.7 及其以前版本中，不同的函数必须具有不同的函数名。当用户自定义函数时，必须保证自定义的函数名和已有的软件内建的和自定义的函数不重名。我们建议，在旧版本中，用户自定义函数时，函数名总是以 MyF 开始，也就是 MyF*****。从 1.7 版开始，MFP 语言引入了引用空间的概念，用户可以在不同的引用空间内定义同名同参数个数的函数，所以，这个要求不再是必要的了；
4. 一些安卓设备，比如三星 Galaxy Express，在连接到个人电脑上后，不允许用户从个人电脑上直接在手机存储卡文件夹中创建新的文件。这个限制能够防止病毒在个人电脑和手机之间传播，但是，它也给用户创建自定义函数程序的时候造成了一些小的麻烦。一个解决办法是，把位于手机存储卡中的 AnMath 目录整个拷贝到个人电脑的一个可读写的位置，在那里创建新的.mfps 函数程序文件，并且在那里启动基于 JAVA 的可编程科学计算器以调式编写的代码。调试完成后，再将 AnMath 文件夹整个拷贝到手机存储卡中的原来位子以覆盖原来的 AnMath 目录。
5. 用户生成的.mfps 文件必须位于 AnMath\scripts 目录或者它的子目录下或者多层次子目录下。比如用户生成一个叫做 abc.mfps 的文件存放在一个名字叫做

AnMath\scripts\mylib\文件库 1

的目录中，在可编程科学计算器启动的时候，文件 AnMath\scripts\mylib\文件库 1\abc.mfps 会被找到，里面定义的函数会被自动地加载。

小结

本章介绍了 MFP 编程语言的基本编程语句和使用方法。MFP 编程语言非常简单，语法和关键字与 Basic 语言类似，有编程基础的用户可以很快掌握，而没有编程基础的用户完全可以把本章当作自己的编程启蒙教材。

用 MFP 编程语言编程，需要为每一项功能提供一个函数（function 语句）；在函数体内，如果用户需要创建变量，需要使用 variable 语句进行声明；MFP 语言的条件语句是 if……elseif……else……endif 和 select……case……default……ends；对于循环则使用 while，for 或者 do，如果要跳出循环，则使用 break 或者 continue。

MFP 虽然和 Basic 一样简单易懂，但在语言设计上比 Basic 语言要更强大。

首先，MFP 支持异常处理（try……throw……catch……endtry 语句），能在程序出错时进行数据数值的恢复，MFP 也支持求解数学表达式（solve 语句），这项功能有助于学生核对自己的计算结果。

更重要的是，MFP 支持引用空间（cittingspace），用户可以开发出完全同名，同参数个数的函数，把它们放在不同的引用空间中，而不会有任何冲突。引用空间的使用非常灵活，通过将函数放入不同的 cittingspace 中以及在不同的位置调用 using cittingspace 语句，用户可以自由的决定在哪一个 mfps 文件内，哪一个函数能够调用哪一个引用空间中的函数，哪一个引用空间先被看到，哪一个后被看到。

引用空间是 MFP 语言从玩具语言迈向真正的工具语言的重要一步。引用空间的引入保证了 MFP 语言的无限的可扩展性，并且为下一步对面向对象的支持提供了条件。当然，对于初级用户来讲，引用空间的概念过于复杂，所以可以完全不管它。MFP 语言的设计保证了对旧的代码 100%的兼容。用户完全可以保留过去的编程习惯。

从 1.7.1 版开始，MFP 语言通过@execution_entry 标注实现了在操作系统命令行中运行脚本的功能。@execution_entry 标注的语法简洁易懂，使用起来灵活方便，是 MFP 语言相对于其他脚本语言的一个优势。

第3章 MFP 编程语言对数、字符串和数组的操作

截止当前版本，MFP 还是一个面向过程，基于函数调用的编程语言，MFP 对于各种各样复杂的数，字符串和数组一方面有内建的支持，另一方面也提供了一系列函数用于操作这些数据类型。

第1节 MFP 对数的操作函数

1. MFP 对整数的操作

MFP 对整数的操作函数包括，如何对一个小数（也就是浮点数）取整，以及如何求整数的模。

对一个浮点数取整，MFP 提供了三个函数：round，ceil 和 floor。round 函数是对一个浮点数四舍五入，比如 1.6 取整为 2，-1.6 取整为 -2，1.4 取整为 1，-1.4 取整为 -1；ceil 函数是返回不小于该浮点数的最小的整数，比如 ceil(1.6) 返回 2，ceil(-1.6) 返回 -1，ceil(-5.0) 返回 -5；floor 函数是返回不大于该浮点数的最大的整数，比如 floor(1.6) 返回 1，floor(-1.4) 返回 -2，floor(-5.0) 返回 -5；

round，ceil 和 floor 还能支持在任意小数位截断取值，这时候，这 3 个函数都需要两个参数，第一个参数是要被截断取值的浮点数，第二个参数是在哪一个小数位截断，比如 round(-1.82347, 4) 就是在小数点后面第四位截断并四舍五入取值，得到 -1.8235；同理，ceil(-1.82347, 4) 就是在小数点后面第四位截断，并返回小数点后面最多跟随四位数字，并且不小于 -1.82347 的最小的数，也就是 -1.8234，而 floor(-1.82347, 1) 就是在小数点后面第 1 位截断，并返回小数点后面最多跟随 1 位数字，并且不大于 -1.82347 的最大数，也就是 -1.9。

MFP 求整数的模的函数名字叫 mod。Mod(x, y) 返回 x 除以正整数 y 的余数（余数必须为正数），如果 x 或者 y 不是整数，将被首先转换为整数，转换的办法是如果被转换数大于 0，则取比不比被转换数大的最大整数，如果被转换数小于 0，则取比不比被转换数小的最小整数。比如 Mod(-17.8214, 4.665) 相当于求 Mod(-17, 4) 其结果为 3。而 Mod(17.8214, 4.665) 相当于求 Mod(17, 4) 其结果为 1。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 numbers, strings and arrays 子目录中的 examples.mfps 文件中找到) :

```
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
```

上述程序的运行结果如下：

```
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 进制转换函数

MFP 支持用二进制，8 进制，10 进制和 16 进制来表示一个整数或者小数。二进制数格式为 0b 打头，比如 0b0011100，八进制数格式为 0 打头，比如 0371.242 或者 00.362，16 进制数格式为 0x 打头，比如 0xAF46BC.0DD3E。用户输入这些数字，MFP 在计算的时候将任何非 10 进制数自动转换为 10 进制数然后进行计算。比如用户可以把一个变量赋值为 0b1101 然后进行计算：

```
Variable a = 0b1101 //相当于 Variable a = 13
```

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

，最后计算出来的结果为

```
a+1 = 14
```

之所以得到 10 进制结果，是因为 MFP 的计算结果数值永远是用十进制表示的，用户需要调用以下函数将一种进制的数或者该进制数表达字符串转换为另一种进制的表达字符串（如果是要转化为 10 进制，则转换结果不是一个字符串而是 10 进制的数值）：

`conv_bin_to_dec(x)` 将一个二进制的非负数或代表该数的字符串 x (x 可以为浮点数，也可以为整数) 转换为一个十进制的数值。

`conv_bin_to_hex(x)` 将一个二进制的非负数或代表该数的字符串 x (x 可以为浮点数，也可以为整数) 转换为一个代表 16 进制数的字符串。

`conv_bin_to_oct(x)` 将一个二进制的非负数或代表该数的字符串 x (x 可以为浮点数，也可以为整数) 转换为一个代表八进制数的字符串。

`conv_dec_to_bin(x)` 将一个十进制的非负数或代表该数的字符串 x (x 可以为浮点数，也可以为整数) 转换为一个代表二进制数的字符串。

`conv_dec_to_hex(x)` 将一个十进制的非负数或代表该数的字符串 x (x 可以为浮点数，也可以为整数) 转换为一个代表 16 进制数的字符串。

`conv_dec_to_oct(x)` 将一个十进制的非负数或代表该数的字符串 x (x 可以为浮点数，也可以为整数) 转换为一个代表八进制数的字符串。

`conv_hex_to_bin(x)` 将一个 16 进制的非负数或代表该数的字符串 `x` (`x` 可以为浮点数，也可以为整数) 转换为一个代表二进制数的字符串。

`conv_hex_to_dec(x)` 将一个 16 进制的非负数或代表该数的字符串 `x` (`x` 可以为浮点数，也可以为整数) 转换为一个十进制的数值。

`conv_hex_to_oct(x)` 将一个 16 进制的非负数或代表该数的字符串 `x` (`x` 可以为浮点数，也可以为整数) 转换为一个代表八进制数的字符串。

`conv_oct_to_bin(x)` 将一个八进制的非负数或代表该数的字符串 `x` (`x` 可以为浮点数，也可以为整数) 转换为一个代表二进制数的字符串。

`conv_oct_to_dec(x)` 将一个八进制的非负数或代表该数的字符串 `x` (`x` 可以为浮点数，也可以为整数) 转换为一个十进制的数值。

`conv_oct_to_hex(x)` 将一个八进制的非负数或代表该数的字符串 `x` (`x` 可以为浮点数，也可以为整数) 转换为一个代表 16 进制数的字符串。

还要注意，这里的代表某个进制数的字符串不包括进制的起头，比如二进制数 `0b1101`（相当于十进制数 13）的表达字符串是“`1101`”，而不是“`0b1101`”。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `numbers, strings and arrays` 子目录中的 `examples.mfps` 文件中找到：

```
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

```

函数打印输出的结果为:

```

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") =
0.0111110011101010111100100101000111000001100100111011001110100110
10001011000110011010010000010101111010001011110000010110100111000
01000111011011110010101001

conv_dec_to_bin(.487960) =
0.0111110011101010111100100101000111000001100100111011001110100110
10001011000110011010010000010101111010001011110000010110100111000
01000111011011110010101001

```

```
conv_dec_to_bin(0.48700) =
0.011111001010110000001000001100010010011011101001011100011010100
11111101111100111011011001000101101000011100101011000001000001100
010010011011101001011110001

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") =
10101011000000000011100110111010.1111111110000000011011101111000
11001

conv_hex_to_dec("0AB0039BA.FFE01BBC64") =
2868918714.99951337193851941265165805816650390625

conv_hex_to_oct("0AB0039BA") = 25300034672

conv_oct_to_bin("027400330.017764") =
101111000000000011011000.0000011111111101

conv_oct_to_dec("027400330.017764") = 6160600.0312042236328125

conv_oct_to_hex("027400330") = 5e00d8
```

3. MFP 逻辑操作函数

逻辑操作函数是对布尔值（TRUE 或者 FALSE）进行操作，常常用于条件判断语句 if 或者 elseif，以及条件判断函数 iff（这里也将一并说明）。

MFP 提供了三个逻辑判断函数：and、or 和 xor。

and 函数接受不少于 1 个的任意个数的参数，返回这些参数的逻辑与值。如果某一个参数不是布尔类型，将会被自动转换为布尔类型（如果能够自动转换的话，如果不能自动转换，将会抛出异常）。

比如，`and(True, 3>2, 1-1)` 我们得到 False，原因在于，True，3>2 的布尔值都是 True，但是 1-1 得 0，它的布尔值是 False，三个参数中，有一个是 False，返回就是 False。而如果把第三个参数改为 1-2 得 -1，它的布尔值是 True，返回就是 True。

or 函数接受不少于 1 个的任意个数的参数，返回这些参数的逻辑或值。如果某一个参数不是布尔类型，将会被自动转换为布尔类型（如果能够自动转换的话，如果不能自动转换，将会抛出异常）。

比如，`or(True, 3>2, 1-1)` 我们得到 True，原因在于，True，3>2 和 1-1 的布尔值中有 True，or 函数的参数，只要有一个是 True，返回就是 True。而如果把第一个参数改为 False，第二个参数改为 3<2，它们的布尔值都变为了 False，返回就是 False。

xor 函数是逻辑异或函数，它有两个参数，如果这两个参数的值不等，则返回 True，否则返回 False。

用户需要搞清楚上述三个函数和位与&，位或|以及位异或^这三个操作符的区别。首先，位操作符的操作数都是正整数，如果遇到不是正整数的操作数，位操作符将尝试强制转换为正整数。而逻辑函数，除了 xor，的参数都是布尔值，如果不是布尔值，则强制转换为布尔值。

其次，位操作符只接受一前一后两个操作数，而逻辑函数 and 和 or 可以接受任意不少于 1 个参数。

最后，位操作符是对操作数的每一个比特位进行逻辑比较，比如 `7&8`，7 的所有比特位为 111，8 的所有比特位为 1000，`7&8` 得到 0000 也就是 0，而逻辑操作函数则是把 7 和 8 转换为布尔值后进行逻辑比较，7 和 8 转换为布尔值后都是 True，所以 `and(7, 8)` 得到 True。

最后介绍条件判断函数 iff。函数 iff 是 if 语句的函数表达形式，它需要最少三个参数，其语法为：`iff(条件 1, 如果条件 1 为 True 的结果, 条件 2, ..., 如果条件 2 为 True 的结果)`。

2, 如果条件 2 为 True 的结果, ..., 如果所有条件都不是 True 的结果)。参数条件 1, 条件 2, ... 为代表条件的布尔值, iff 函数的返回值由条件值决定。比如, iff(true, 3, 2) 返回 3, iff(3 < 2, 3, 2) 返回 2 (这是因为 $3 < 2$ 是 False), iff(3 < 2, 3, 5 > 4, 5, 6 == 9, 6, 9) 返回 5, 以及 iff(3 < 2, 3, 5 < 4, 5, 6 == 9, 6, 9) 返回 9。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 numbers, strings and arrays 子目录中的 examples.mfps 文件中找到) :

```
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

```

函数打印输出的结果为：

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. MFP 对复数的操作函数

众所周知，复数包括实部和虚部，也可以表示为幅值和幅角。MFP 提供了 4 个函数：real, image, abs 和 angle 用于分别返回一个复数的实部、虚部、幅值和幅角。注意 abs 也可以返回一个实数的绝对值。

比如 real(-3+2i) 返回 -3, image(-3+2*i) 返回 2 (注意返回的是虚部的实数值，如果要返回虚数值，需要增加一个参数 true，比如 image(-3+2*I, true) 返回 2i)，abs(-3+2*i) 返回 3.60555128，angle(-3+2*i) 返回 2.55359005 (基于弧度)。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 numbers, strings and arrays 子目录中的 examples.mfps 文件中找到) :

```
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+2i))  
    print("\nabs(-3+2*i) = " + abs(-3+2 * i))  
    print("\nangle(-3+2*i) = " + angle(-3+2i))  
endf
```

函数打印输出的结果为:

```
real(-3+2*i) = -3  
image(-3+2*i) = 2  
abs(-3+2*i) =  
3.6055512754639893469033040673821233212947845458984375  
angle(-3+2*i) =  
2.5535900500422257345460612287910790449857054711524495709749445923
```

第 2 节 MFP 对字符串的操作函数

对字符串的操作包括定位字符串中的一个或多个字符，将几个字符串连接起来，或者将一个字符串分解，等等。

首先，用户可能会想要知道一个字符串中间包含多少个字符。这时可以调用 `strlen` 函数，比如 `strlen("abcdefg!")` 会返回 8，表示里面有 8 个字符。

然后，用户可能会想知道其中第 3 个和第四个字符是什么，这时，就要用到 `strsub` 函数。`strsub(str, start, end)` 返回字符串 str 的子字符串。该子字符串从字符 start 开始到字符 end-1。注意字符串的第一个字符是 0 号字符。这时，调用 `strsub("abcdefg!", 2, 4)` 得到一个新的字符串 "cd"，它包含两个字符，第一个是原字符串的第 3 个字符（索引号为 2），第二个是原字符串的第 4 个字符（索引号为 3）。

注意，MFP 没有字符数据类型，只有字符串类型，所以，哪怕用户只返回一个字符，返回的数据仍然是一个字符串，比如 `strsub("abcdefg!", 2, 3)` 返回字符串 "c"。

如果用户想返回从原字符串的某个字符开始，到原字符串的尾部的所有字符，则 `strsub` 的第三个参数可以省略，比如 `strsub("abcdefg!", 2)` 得到 "cdefg!"。

如果 `strsub` 的第二个参数或者第三个参数超出了字符串的范围，则会报错，比如运行 `strsub("abcdefg!", 2, 10)` 会得到出错提示： Invalid parameter range!。

如果用户想把若干个（不少于 2 个）字符串顺次连接起来，则可以调用 `strcat` 函数，比如 `strcat("abc", "hello", "1,3,4")` 将三个参数字符串连接起来返回字符串 "abchello 1,3,4"。也可以直接使用加号，也就是 "abc" + "hello" + "1,3,4"，也会得到同样的结果。

如果用户想把字符串切割为若干个子字符串，则应该调用 `split` 函数。`split(string_input, string_regex)` 将字符串 `string_input` 按照正则表达式 `string_regex` 分割为若干个子字符串并返回包含所有子字符串的数组。正则表达式可以非常复杂，用户需要阅读 JAVA 语言的 Pattern 类和 `String.split` 函数的帮助文档获得更多关于正则表达式使用方法的信息。

在这里给出几个例子分别包括将一个字符串用空格来分割，用冒号来分割，用字母来分割和用逗号来分割。

比如，`split(" ab kkk\t6\nnd", "\s+")`会返回一个字符串数组["", "ab", "kkk", "6", "d"]。这是因为，“\s+”是基于空格分割的正则表达式，而原字符串中，包括普通的空格，\t也就是缩进符，\n也就是换行符，它们都被看作空格的一种，所以，都作为切割标志。另外还要注意，在这个例子中 ab 和 kkk 之间的空格不止一个，但是，由于分割的正则表达式里面有个+号，相连的空格会被当作单一的切割标志。

再比如，`split("boo:and:foo", ":")`返回["boo", "and", "foo"]以及`split("boo:and:foo", "o")`返回["b", "", ":and:f"]，在这里，由于分割的正则表达式里面没有+号，所以，相连的 o 不会被当作单一的切割标志，所以，boo:会被切割为“b”、“”、“:...”。

还比如，`split(",Hello,world,", ",")`则使用逗号作为切割符，返回["", "Hello", "world"]，第一个逗号前面没有内容，切割返回空字符串，与此对比最后一个逗号后面没有内容，却被忽略了，这是用户需要注意的。

除了上述基本的几个字符串函数，还有一些其他的函数用于实现对字符串复杂的操作，比如，`trim_left`, `trim_right` 和 `trim` 函数分别去除参数字符串左边，右边和两边的空白字符（包括换行符和缩进符）并返回新字符串，例如`Trim("\t \tabc def \n ")`返回字符串"abc def"。

用户还可以对字符串进行一些转换操作，比如`to_lowercase_string`将参数字符串中所有的大写字母转化为小写字母，`to_uppercase_string`将参数字符串中所有的小写字母转化为大写字母，而`to_string`返回参数（可以是任何数据类型）的打印值（也是一个字符串），比如`to_lowercase_string("abEfg")`返回"abefg"，而`to_string(123)`返回字符串"123"。

用户也可以对两个字符串的部分或全部内容进行比较。比如，`strcmp(src, dest, src_start, src_end, dest_start, dest_end)`比较源字符串src（从src_start到src_end）和目标字符串dest（从dest_start到dest_end）。如果src和dest相等返回0，如果src大于dest返回大于0的值，如果src小于dest返回小于0的值。注意字符串索引从0开始，

`src_end` 和 `dest_end` 的索引位置为最后一个被选中字符的索引位置加一。另外，最后四个参数可以省略，如果被省略，`src_start` 和 `dest_start` 的缺省值为 0，`src_end` 和 `dest_end` 的缺省值为对应字符串的长度。

而 `stricmp(src, dest, src_start, src_end, dest_start, dest_end)` 在忽略字母大小写的前提下比较源字符串 `src`（从 `src_start` 到 `src_end`）和目标字符串 `dest`（从 `dest_start` 到 `dest_end`）。如果 `src` 和 `dest` 相等返回 0，如果 `src` 大于 `dest` 返回大于 0 的值，如果 `src` 小于 `dest` 返回小于 0 的值。注意字符串索引从 0 开始，`src_end` 和 `dest_end` 的索引位置为最后一个被选中字符的索引位置加一。另外，最后四个参数可以省略，如果被省略，`src_start` 和 `dest_start` 的缺省值为 0，`src_end` 和 `dest_end` 的缺省值为对应字符串的长度。

字符串比较函数的例子包括

`stricmp("abc", "ABc")` 得到 0，原因是在忽略大小写的前提下，“abc”和“ABc”相同。

`strcmp("abcdefgk", "defik", 5, 8, 2, 5)` 则比较“abcdefgk”的第 6, 7, 8 个字符和“defik”的第 3, 4, 5 个字符，也就是比较“fgk”和“fik”，所以返回 -2，也就是第一个字符串小于第二个字符串的意思。

最后在这里介绍两个函数，`conv_ints_to_str` 和 `conv_str_to_ints`，第一个函数用于将一组整数转换成一个 Unicode 字符串，每一个整数分别对应字符串中的一个字符。用户也可以仅仅输入一个单一的整数，则该函数将这个单一的整数转化为包含一个 Unicode 字符的字符串。第二个函数则用于将一个 Unicode 字符串转换成一个整数数组。通常，一个 Unicode 字符对应一个整数（但也有可能对应两个整数，如果该 Unicode 字符超出了 UTF-16 字符集的范围。但这种情况很少出现）。由于汉字和一些常用的特殊符号比如¥Σ⑨都是 Unicode 符号，调用这两个函数，可以帮助用户在程序中输入输出这些符号，比如：

`conv_str_to_ints("中文汉字¥Σ⑨")` 会返回 [20013, 25991, 27721, 23383, 165, 8721, 9320]

而利用 `conv_str_to_ints` 函数的返回值作为 `conv_ints_to_str` 的参数，我们得到

conv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320]) 返回“中文汉字Σ⑨”。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 numbers, strings and arrays 子目录中的 examples.mfps 文件中找到) :

```
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("\nssplit(\" ab kkk\t6\nnd\", \"\\s+\") = " _

        + split(" ab kkk\t6\nnd", "\s+"))

    print("\nssplit(\"boo:and:foo\", \":\") = " _

        + split("boo:and:foo", ":"))
```

```

print("\nssplit(\"boo:and:foo\", \"o\") = " _
+ split("boo:and:foo", "o"))

print("\nssplit(\"Hello,world,\", \",\") = " _
+ split(",Hello,world,", ","))

print("\nTrim(\"\\t \\tabc    def  \\n  \") = " _
+ Trim("\t \tabc    def  \n  "))

print("\nnto_lowercase_string(\"abEfg\") = " _
+ to_lowercase_string("abEfg"))

print("\nnto_string(123) = " + to_string(123))

print("\nnstrcmp(\"abc\", \"ABc\") = " + strcmp("abc", "ABc"))

print("\nnstrcmp(\"abcdefgk\", \"defik\", 5, 8, 2, 5) = " _
+ strcmp("abcdefgk", "defik", 5, 8, 2, 5))

print("\nconv_str_to_ints(\"中文汉字\S\S\") = " _
+ conv_str_to_ints("中文汉字\S\S"))

print("\nconv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320]) = " _
+ conv_ints_to_str([20013, 25991, 27721, 23383, 165, 8721, 9320])))

endf

```

测试程序输出结果为：

```

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\nnd", "\\\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

strcmp("abc", "ABc") = 0

strcmp("abcdefgk", "defik", 5, 8, 2, 5) = -2

conv_str_to_ints("中文汉字\S\S\S\S\S") = [20013, 25991, 27721,
165, 8721, 9320]

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

第3节 MFP对数组和矩阵的操作函数

数组和矩阵是 MFP 数据类型的一个重要组成部分。数组中包括若干个元素，每个元素可以是数，字符串或者数组。数组支持基本的 MFP 操作符加减乘除转置以及整数次方，此外，软件还提供了一系列的函数用于操作数组和矩阵。

1. MFP 创建数组的函数

首先需要了解的是如何创建一个数组。MFP 提供了一些函数。第一个是 alloc_array 函数。Alloc_array 函数接受一个或多个参数，如果参数多于一个，每个参数都必须为正整数，表示生成数组的在每一个对应维度上的尺寸。如果参数只有一个，并且是一个数组，那么数组中的每一个元素都必须为正整数，表示生成数组的在每一个对应维度上的尺寸。生成的数组，每一个元素都被初始化为 0。比如 alloc_array(3) 返回 [0, 0, 0]，而 alloc_array(2, 3, 4) 和 alloc_array([2, 3, 4]) 都返回 [[[0, 0, 0, 0], [0,

`0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]]。`

如果用户希望 `alloc_array` 返回的数组，每个元素不是被初始化为 0，而是其他的某个值，则需要使用 `alloc_array` 的另外一个用法：`alloc_array(x, y)`，其中 `x` 是一个数组，`x` 中的每一个元素都必须为正整数，表示生成数组的在每一个对应维度上的尺寸，`y` 表示所有元素的初始值。比如调用 `alloc_array([2, 1], "hello")` 得到 `[["hello"], ["hello"]]`。

第二是 `eye`, `ones` 和 `zeros` 函数。`eye(x)` 返回正整数 `x` 乘 `x` 的 2 维方阵 I。注意表达式 `eye(0)` 返回常数 1。

`ones` 函数返回一个所有元素都是 1 的矩阵，和 `alloc_array` 相似，本函数的参数用于决定矩阵的尺寸，要么为一批正整数，要么为一个正整数数列。

`zeros` 函数返回一个所有元素都是 0 的矩阵，和 `alloc_array` 相似，本函数的参数用于决定矩阵的尺寸，要么为一批正整数，要么为一个正整数数列。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `numbers, strings and arrays` 子目录中的 `examples.mfps` 文件中找到）：

```
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("\nnones(2, 3) = " + ones(2, 3))

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


endf

```

上述函数的运行结果如下：

```

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. 获取数组的尺寸和判断数组的特征

对数组进行操作的第二步是获取一个已有数组的尺寸，这需要调用 size 函数。size 函数有两种使用方法，第一种，只接受一个参数，也就是所操作的数组矩阵：size(x) 返回矩阵 x 的尺寸向量。注意如果 x 不是一个矩阵，则总是返回[]。注意，这里的尺寸向量是指每个维度的最大的尺寸向量，

比如`[1, 2+3i, [5, "hello", [9, 10], 6], 11, 12]`包括 5 个元素，分别为 1, 2+3i, [5, "hello", [9, 10], 6], 11 和 12，所以，第一维的尺寸为 5，第二维，元素 1, 2+3i, 11 和 12 的尺寸都是`[]`（这里，任何复数都被视作单一元素），唯有`[5, "hello", [9, 10], 6]`是一个数组包含 4 个元素，所以第二维的尺寸为 4，第三维，元素 5, "hello" 和 6 都是单个的元素（这里，任何字符串都被视为单一元素），尺寸都是`[]`，唯有`[9, 10]`是一个包含两个元素的数组，所以第三维尺寸为 2。所以，最终我们得到`size([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12]) == [5, 4, 2]`。

`size` 函数的第二种是用办法接受两个参数：`size(x, y)` 返回矩阵 `x` 前 `y` 维的尺寸向量，如果 `x` 少于 `y` 维，返回完整的尺寸向量，注意 `y` 必须为正整数。此外如果 `x` 不是一个矩阵，则总是返回`[]`。类似上面的例子，如果调用

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

获取数组`([1, 2+3i, [5, "hello", [9, 10], 6], 11, 12])`尺寸的前两维会得到返回值为`[5, 4]`。

对数组或矩阵进行操作的第三步是判断数组矩阵的特性。用户可使用`is_eye(x)` 函数判断参数 `x` 是否是一个 I 矩阵（单位矩阵）；

用户可以使用`is_zeros(x)` 函数判断参数 `x` 是否是一个元素全为 0 的矩阵；

用户还可以使用`includes_inf(x)`, `includes_nan(x)`,
`includes_null(x)`, `includes_nan_or_inf(x)` 和
`includes_nan_or_inf_or_null(x)` 来分别判断参数 `x` 是否为一个包含值为 inf 或 -inf 的元素的数组，是否为一个包含值为 nan 的元素的数组，是否为一个包含值为 null 的元素的数组，是否为一个包含值为 nan 或者 inf 或 -inf 的元素的数组，以及是否为一个包含值为 nan 或者 inf 或 -inf 或者 null 的元素的数组。

以下是上面几个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 numbers, strings and arrays 子目录中的 examples.mfps 文件中找到)：

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

上述函数的运行结果如下：

```

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. 对数组赋值

对数组进行操作的第四步是对数组赋值，显然，用户可以用对变量复制的常规办法把一个数组赋值给一个变量，比如：

```
Variable a = [1, 2, 3]
```

```
a[1] = [7, 9]
```

但是，很多时候，用户需要在程序中间动态地给数组赋值，并且被赋予的值位置超过了数组的范围，比如上面，在将 $a[1]$ 赋值为 $[7, 9]$ 之后，用户还想将 $a[4]$ 赋值为 $3+6i$ ，但是数组 a 不存在 $a[4]$ 这个元素，这个时候，就需要调用 `set_array_elem` 函数。`set_array_elem(x, y, z)` 将 $x[y]$ 赋值为 z ，并且返回新的 x 。注意 x 不是必须为矩阵， y 必须为正整数向量。 y 的值可以超出 x 的尺寸和 维度。比如，如果 $x=3$, $y=[1, 2]$, $z=2+3i$ ，那么 `set_array_elem(x, y, z)` 等于 $[3, [0, 0, 2+3i]]$ 。还要注意，调用了 `set_array_elem` 函数之后， x 的值可能会自动变为新值，也可能不会。所以，必须将 `set_array_elem` 的返回值赋予 x ，以保证 x 的值得到更新。所以，`set_array_elem` 的正确的调用方法是

```
x = set_array_elem(x, y, z)
```

回到上面的例子，如果想将 $a[4]$ 赋值为 $3+6i$ ，办法是

```
a = set_array_elem(a, 4, 3+6i)
```

，运行完此语句之后，得到 a 的值为 $[1, [7, 9], 3, 0, 3 + 6 * i]$ 。 $a[3]$ 原本也不存在，但由于需要生成 $a[4]$ ，所以 $a[3]$ 的值也被自动生成并赋值为 0

用户在建立自己的函数的时候，有时需要用数组作为参数，在函数体内，又调用了 `set_array_elem` 或者赋值语句给数组的一部分赋值，这个时候需

要注意，数组在子程序中发生了变化，主程序中也会看到。比如，有个子程序的定义为

```
function subfunc1(array_value)
...
Variable my_array = Array_value
my_array[2] = 7
...
Endf
```

现在用户在主程序中调用`::mfpexample::subfunc1`，代码如下

```
...
variable array_val = [1, 2, 3]
::mfpexample::subfunc1(array_val)
print(array_val)
...
```

在运行完`print(array_val)`后，用户会发现，`array_val`变成了`[1, 2, 7]`了。

之所以会这样，是因为，MFP 在传递数组参数时，不是把数组整个拷贝到子程序的栈中，而是数组的引用拷贝到子程序的栈中。这样一来，子程序和主程序实际上操作的是同一个数组。在上述代码中，子程序又将参数数组赋值给一个新的变量，但是，数组的赋值也仅仅只是引用的拷贝，最终，子程序中数组的改变也造成了主程序中相应的改变。

那么，如果用户需要在子程序中对数组的值加以改变，但又不希望主程序受影响，该怎么办呢？这时可以调用`clone`函数。这个函数接受一个参数，该参数可以为任何数据类型，包括数，数组和字符串，该函数将参数

的值拷贝并返回，返回值和参数值虽然在数值上相同，但在内存中保存在不同的地方，不会相互影响，比如上面的例子，我们将子程序改为

```
function subfunc2(array_value)
...
Variable my_array = clone(Array_value)
my_array[2] = 7
...
Endf
```

现在用户在主程序中调用::mfpxexample::subfunc2，代码如下

```
...
variable array_val = [1, 2, 3]
::mfpxexample::subfunc1(array_val)
print(array_val)
...
```

在运行完 print(array_val) 后，用户会发现，array_val 还是[1, 2, 3]。

以下是对数组赋值和通过参数传递数组的例子程序。本例子可以在本手册自带的示例代码所在目录中的 numbers, strings and arrays 子目录中的 examples.mfps 文件中找到）：

```
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)

    ::mfpxexample::subfunc2(array_val)

    // clone function called in ::mfpxexample::subfunc2, any change inside

    // will not affect main function.

    // 由于 clone 函数被调用，子函数改变数组参数的值不会对主函数有影响。

    print("\nWith clone, after calling sub function array_val is " + _

        array_val)

    ::mfpxexample::subfunc1(array_val)

    // clone function not called in ::mfpxexample::subfunc2, value changes

    // of array_val inside will affect main function.

    // 由于 clone 函数没有被调用，子函数改变数组参数的值会对主函数有影响。

    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

    // 在调用 set_array_elem 函数后，array_val 有 5 个元素了。

```

```
    print("\nAfter set_array_elem array_val is " + array_val)  
endf
```

用户在命令提示符下运行`::mfpexample::assignValue2Array()`，或者先输入`using citingspace ::mfpexample`，然后再执行`assignValue2Array()`，结果如下：

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]

小结

本章详细说明了MFP编程语言对实数，复数，数组以及字符串的操作办法，MFP编程语言对复数，数组和字符串有内建的支持，但是，为了实现各种复杂的功能，仍然需要调用函数。MFP编程语言提供了一整套的对浮点数四舍五入的函数，一整套进制转换函数，一组存取复数实部和虚部的函数，一整套获取字符串信息（如长度，子字符串等）的函数以及一组操作数组的函数（包括创建，获取尺寸和赋值）。

需要注意的是，对MFP编程语言来说，数组和矩阵是两个相关但并不完全一样的概念。MFP的矩阵多半是指二维方阵，而数组的维度是任意的，每一组的长度也不一定要一样。矩阵是一种特殊的数组。本手册在后面详细介绍使用矩阵的数学计算函数。

第4章 MFP 编程语言对于各种数学和科学计算的支持

前面已经介绍了 MFP 语言基本运算操作符（包括加减乘除等），以及用 solve 语句解方程，这一章将侧重于介绍 MFP 语言所提供的进行各种数学计算的函数。

第1节 内置的数学常用变量

前面的章节已经介绍了 MFP 一些内置的变量包括虚数单位 i ，空值 $null$ 以及无定义数 nan 和 $nani$ 。为了方便进行数学计算，MFP 还内置了

1. 代表无穷大的变量 inf ，这个变量表示实数正无穷大。显然， $-inf$ 表示实数负无穷大。
2. 代表虚数无穷大的变量 $infi$ ，这个变量表示在虚轴正方向上的无穷大，显然，在虚轴负方向上的无穷大为 $-infi$ 。

之所以定义这个变量是因为虚轴上的无穷大是无法通过 inf 和 i 相乘得到（它们的乘积等于 $nan+infi$ ），所以必须用一个特别的变量来定义。

3. 代表圆周率的变量 pi ，它的值等于
3.1415926535897932384626433832795028841971693993751058209749445923。
4. 代表自然对数的变量 e ，它的值等于
2.7182818284590452353602874713526624977572470936999595749669676277。

以上的这些变量常用于下面所介绍的函数的参数。

第2节 单位转换函数和返回物理化学常量值的函数

MFP 语言提供了单位转换函数 `convert_unit`。`convert_unit(value, from_unit, to_unit)` 将基于某一个单位的数值转换为基于另外一个单位的数值。第一个参数是将要转换的数值，第二个参数是将被转换的单位（单位是一个对大小写敏感的字符串），第三个参数是转换后的单位（单位是一个对大小写敏感的字符串）。比如，`convert_unit(23.71, "m", "km")` 返回 0.2371。

本函数支持以下单位：

1. 长度单位：“um”（微米），“mm”（毫米），“cm”（厘米），“m”（米），“km”（公里），“in”（英寸），“ft”（英尺），“yd”（码），“mi”（英里），“nmi”（海浬），“AU”（天文单位），“ly”（光年），“pc”（秒差距）；
2. 面积单位：“mm2”（平方毫米），“cm2”（平方厘米），“m2”（平方米），“ha”（公顷），“km2”（平方公里），“sq in”（平方英寸），“sq ft”（平方英尺），“sq yd”（平方码），“ac”（英亩），“sq mi”（平方英里）；
3. 体积单位：“mL”（毫升），“L”（升），“m3”（立方米），“cu in”（立方英寸），“cu ft”（立方英尺），“cu yd”（立方码），“km3”（立方公里），“fl oz(Imp)”（液盎司（英制）），“pt(Imp)”（品脱（英制）），“gal(Imp)”（加仑（英制）），“fl oz(US)”（液盎司（美制）），“pt(US)”（品脱（美制）），“gal(US)”（加仑（美制））；
4. 质量单位：“ug”（微克），“mg”（毫克），“g”（克），“kg”（千克），“t”（吨），“oz”（盎司），“lb”（磅），“jin”（市斤），“jin(HK)”（斤（香港）），“jin(TW)”（台斤）；
5. 速度单位：“m/s”（米每秒），“km/h”（千米每小时），“ft/s”（英尺每秒），“mph”（英里每小时），“knot”（节）；
6. 时间单位：“ns”（纳秒），“us”（微秒），“ms”（毫秒），“s”（秒），“min”（分钟），“h”（小时），“d”（天），“wk”（礼拜），“yr”（年）；
7. 力单位：“N”（牛顿），“kgf”（千克力），“lbF”（磅力）；
8. 压强单位：“Pa”（帕斯卡），“hPa”（百帕），“kPa”（千帕），“MPa”（兆帕），“atm”（大气压），“psi”（每平方英寸上受到的磅力压力），“Torr”（毫米汞柱）；
9. 能量单位：“J”（焦耳），“kJ”（千焦），“MJ”（兆焦），“kWh”（千瓦时），“cal”（卡路里），“kcal”（千卡），“BTU”（英热单位）；

10. 功率单位: "W" (瓦特) , "kW" (千瓦) , "MW" (兆瓦) , "cal/s" (卡路里每秒) , "BTU/h" (英热单位每小时) , "hp" (马力) ;

11. 温度单位: "0C" (摄氏度) , "0F" (华氏度) , "K" (开氏温标) 。

用户也能够在 MFP 中获取一些物理或者化学常量值。函数 get_constant(const_name, n) 返回一个由区分大小写的字符串 const_name 所对应的常数值，返回的数值将会四舍五入后保留小数点后面 n 位有效数值，这里 n 为非负整数并且可以省略。如果 n 被省略，返回值将不会被四舍五入处理。本函数支持以下常数：

1. 圆周率 (const_name == "pi") ;

2. 自然对数 (const_name == "e") ;

3. 真空中的光速[m/s] (const_name == "light_speed_in_vacuum") ;

4 . 万 有 引 力 常 数 [$m^3/kg/(s^2)$] (const_name == "gravitational_constant") ;

5. 普朗克常数[J*s] (const_name == "planck_constant") ;

6 . 磁 常 数 (真 空 磁 导 率) [N/(A 2)] (const_name == "magnetic_constant") ;

7 . 电 常 数 (真 空 电 容 率) [F/m] (const_name == "electric_constant") ;

8. 基本电荷[c] (const_name == "elementary_charge_constant") ;

9. 阿伏伽德罗常数[1/mol] (const_name == "avogadro_constant") ;

10. 法拉第常数[C/mol] (const_name == "faraday_constant") ;

11. 气体常数[J/mol/K] (const_name == "molar_gas_constant") ;

12. 玻尔兹曼常量[J/K] (const_name == "boltzmann_constant") ;

```
13. 标准重力[m/(s**2)] (const_name == "standard_gravity") ;
```

举个例子，如果用户输入 `get_constant("pi", 4)`，结果将会是 3.1416；如果用户输入 `get_constant("pi", 8)`，结果将会是 3.14159265；如果用户输入 `get_constant("pi", 0)`，将会得到 3，如果用户输入 `get_constant("pi")` 返回值将是 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679（小数点后有 100 位数字），这个数值就是本软件内部所使用的圆周率数值。

以下是上面这两个函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `math libs` 子目录中的 `examples.mfps` 文件中找到）：

```
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\", \"f1 oz(US)\") = " _
```

```
        + convert_unit(23.71, "m3", "f1 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
```

上述例子的运行结果如下：

```
convert_unit(23.71, "m3", "f1 oz (US)") =
801730.47826069746286815067409321511215541775256437992850079847482
18874

get_constant("pi", 4) = 3.1416

get_constant("pi", 8) = 3.14159265

get_constant("pi", 0) = 3

get_constant("pi") =
3.1415926535897932384626433832795028841971693993751058209749445923
```

第3节 三角函数双曲三角函数

MFP 语言的双曲函数和双曲三角函数和数学上对这些函数的定义同名，用法也一样。唯一需要注意的是，对于三角函数（包括反三角函数），如果结尾没有字母 d，则是基于弧度的计算，结尾有字母 d，则是基于角度的计算，比如 $\cos(\pi/3)$ 返回 0.5，而 $\text{asind}(0.5)$ 则等于 30，表示 30 度。还要注意这些函数都支持复数计算，比如 $\text{asind}(8)$ 返回 $90 - 158.63249757 * i$ 。

MFP 所有三角函数和反三角函数的列表如下：

函数名	函数帮助信息
acos	acos(1)： acos(x)返回 x 的反余弦值，注意 x 可以为复数。
acosd	acosd(1)： 函数 acosd(x)为余弦函数的反函数,注意返回值为角度数。
asin	asin(1)： asin(x)返回 x 的反正弦值，注意 x 可以为复数。

asind	asind(1) : 函数 asind(x)为正弦函数的反函数,注意返回值为角度数。
atan	atan(1) : atan(x)返回 x 的反正切值, 注意 x 可以为复数。
atand	atand(1) : 函数 atand(x)为正切函数的反函数,注意返回值为角度数。
cos	cos(1) : cos(x)返回 x 的余弦值, 注意 x 可以为复数。
cosd	cosd(1) : 函数 cosd(x)返回基于角度的 x 的余弦值。
sin	sin(1) : sin(x)返回 x 的正弦值, 注意 x 可以为复数。
sind	sind(1) : 函数 sind(x)返回基于角度 x 的正弦值。
tan	tan(1) : tan(x)返回 x 的正切值, 注意 x 可以为复数。
tand	tand(1) : 函数 tand(x)返回基于角度 x 的正切值。

MFP 所有双曲三角函数的列表如下：

函数名	函数帮助信息
acosh	acosh(1) : 函数 acosh(x)为双曲余弦函数的反函数。
asinh	asinh(1) :

	函数 $\text{asinh}(x)$ 为双曲正弦函数的反函数。
atanh	$\text{atanh}(1) :$ 函数 $\text{atanh}(x)$ 为双曲正切函数的反函数。
cosh	$\text{cosh}(1) :$ 函数 $\text{cosh}(x)$ 为双曲余弦函数。
sinh	$\text{sinh}(1) :$ 函数 $\text{sinh}(x)$ 返回 x 的双曲正弦值。
tanh	$\text{tanh}(1) :$ 函数 $\text{tanh}(x)$ 用于计算 x 的双曲正切值。

以下是上述函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `math libs` 子目录中的 `examples.mfps` 文件中找到) :

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

```

运行上述例子函数返回的结果是：

```

cos(pi/3) = 0.5

tand(45) = 1

sin(1 + 2*i) =
3.1657785132161682200525265356615738370974142362979081716694416027
+
1.959601041421606115476765045922954107994611047413801140191859959i

asind(0.5) =
30.00000000000003071288025528876242434085090019423660218589920376

acos(8) = -2.768659383313573751905778408399783074855804443359375i

acosh(4.71 + 6.44i) =
2.771116084398325796200879267416894435882568359375 +
0.94305685974139741301058847966487519443035125732421875i

sinh(e) = 7.544137102816974971286612117182812653481960296630859375

atanh(e) =
0.385968416452652396397837719632661901414394378662109375 +
1.5707963267948966192313216916397514420985846996875529104874722962
i

```

。需要注意的是 asind(0.5) 的返回值应该是 30 (sin30 度等于 0.5)，但是实际上的結果是
30.00000000000003071288025528876242434085090019423660218589920376

，这是由于 asind 是将参数先转换为复数在进行计算的，转换的过程会造成微量的计算误差。

第 4 节 指数，对数和次方函数

MFP 支持一系列的指数，对数和次方函数，参见下表：

函数名	函数帮助信息
exp	$\exp(1) :$ $\exp(x)$ 返回自然对数 e 的 x 次方，x 可以为实数或者虚数。
lg	$\lg(1) :$ 函数 $\lg(x)$ 返回 x 的自然对数。
ln	$\ln(1) :$ 函数 $\ln(x)$ 返回 x 的自然对数。
log	$\log(1) :$ $\log(x)$ 返回 x 的自然对数，注意 x 可以为复数。
log10	$\log10(1) :$ 函数 $\log10(x)$ 返回 x 以十为底的对数。
log2	$\log2(1) :$ 函数 $\log2(x)$ 返回 x 以 2 为底的对数。
loge	$\loge(1) :$ 函数 $\loge(x)$ 返回 x 的自然对数。
pow	$\text{pow}(2) :$ $\text{pow}(x,y)$ 返回 x 的 y 次方，注意 x 和 y 可以为实数，可以为虚数。 如果结果有多个值，返回第一个值。

	<code>pow(3) :</code> pow(x,y,z)返回包含 x 的 y 次方的前 z 个值组成的一个向量，如果 x 的 y 次方只有少于 z 个值，返回所有值。注意 y 必须为实数，x 可以为实数，可以为虚数，z 必须为正整数。
<code>sqrt</code>	<code>sqrt(1) :</code> 函数 sqrt(x)返回实数 x 的平方根值。

这里需要注意几点：

第一， $\lg(x)$ 、 $\log(x)$ 、 $\ln(x)$ 和 $\log_e(x)$ 返回的都是 x 的自然对数， $\log_2(x)$ 返回 x 的以 2 为底的对数， $\log_{10}(x)$ 返回 x 以 10 为底的对数。这些求对数的函数的参数都可以为复数。至于以其他任何数为底的对数，可以由这些对数函数相除得到，比如求 x 以 3 为底的对数的计算公式为 $\log(x)/\log(3)$ 。

第二， pow 函数有两种用法，第一种是 $\text{pow}(x, y)$ ，它返回 x 的 y 次方，注意 x 和 y 可以为实数，可以为虚数。如果结果有多个值，返回复平面上幅角从 0 度开始逆时针旋转遇到的第一个值。这种用法等价于使用次方操作符，也就是 $x**y$ 。比如 $\text{pow}(32, 0.2)$ 也就是 32 的 $1/5$ 次方得到 2，但是 $\text{pow}(-32, 0.2)$ 却不会返回 -2（虽然 -2 是它的一个根），而是返回 $1.61803399 + 1.1755705 * i$

第二种是 $\text{pow}(x, y, z)$ ，它返回包含 x 的 y 次方的前 z 个值（在复平面上从幅角 0 度开始逆时针旋转遇到的前 z 个值）组成的一个向量，如果 x 的 y 次方只有少于 z 个值，返回所有值。注意 y 必须为实数，x 可以为实数，可以为虚数，z 必须为正整数。比如，如果想要返回 -32 的 $1/5$ 次方的所有根，可以调用 $\text{pow}(-32, 0.2, 5)$ 得到一个包含 5 个元素的数组也就是 $[1.61803399 + 1.1755705 * i, -0.61803399 + 1.90211303 * i, -2, -0.61803399 - 1.90211303 * i, 1.61803399 - 1.1755705 * i]$ 。

第三， $\text{sqrt}(x)$ 函数实际上和表达式 $x**0.5$ 或者 $\text{pow}(x, 0.5)$ 的效果完全一样，都是返回在复平面上从幅角 0 度开始逆时针旋转遇到的 x 的第一个平方根。比如 $\text{sqrt}(4) == 2$ ， $\text{sqrt}(-2) ==$

$1.41421356 * i$ 和 $\sqrt{-2+3i} == 0.89597748 + 1.67414923 * i$ 。

以下是上述函数例子程序。本例子可以在本手册自带的示例代码所在目录中的 `math libs` 子目录中的 `examples.mfps` 文件中找到）：

```
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
```

上述函数的运行结果如下：

$\lg(e) == 1$

$\log(9, 3) == \log(9)/\log(3) == 2$

$\log2(3 + 4i) ==$

2. 3219280948873622916712631553180615794157506196217129274315603707

+

1. 337804212450976175615004492526409565791145361743891813677556325i

$\text{pow}(32, 0.2) == 2$

$\text{pow}(-32, 0.2) ==$

1. 6180339887498949025257388711906969547271728515625 +

1. 175570504584946274206913585658185184001922607421875i

$\text{pow}(-32, 0.2, 5) ==$

[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. 6741492280355400682758136732173307274213575287387175311747860088

i

第5节 矩阵相关函数

上一章中介绍了 MFP 对数组的相关操作函数，这里，必须指出数组的概念和数学上的矩阵的相关和不同的地方。

首先，数学上的矩阵就是 MFP 中的数组，所以，对数组的基本操作，包括创建，存取，修改函数，以及加减乘除转置等操作符，对数学上的矩阵都是适用的。

其次，数学上的矩阵式一种特殊的数组，它必须是二维的，并且，在大多数情况下是一个方阵。这里介绍的函数，适用于数学上的矩阵（除了一个例外，也就是 dprod 函数，它用于两个一维向量的点乘），对于其他的 MFP 的数组，并不适用。

以下是当前版本的 MFP 所支持的矩阵函数的列表：

函数名	函数帮助信息
adj	adj(1)： 函数 cofactor(x)返回 2 维方阵 x 的伴随矩阵。
cofactor	cofactor(1)： 函数 cofactor(x)返回 2 维方阵 x 的余因子矩阵。
det	det(1)： det(x)计算 2 维方阵 x 的行列式值。
deter	deter(1)： deter(x)计算 2 维方阵 x 的行列式值。
dprod	dprod(2)： 函数 dprod 用于计算两个同等长度的一维向量 [x1, x2, ... xn] 和 [y1, y2, ... yn] 的点乘值。
eig	eig(1)： eig(A)计算 2 维方阵 A 的特征向量和特征值。这个函数

	<p>返回一个包含两个成员的列表。第一个成员是特征向量矩阵，每一列是一个特征向量。第二个成员是一个对角矩阵，每一个对角线元素是一个特征值。注意运算这个函数非常耗费内存和 CPU 资源，如果在手机上运行，矩阵 A 的尺寸最好不要超过 6*6，如果在电脑上运行，最好不要超过 8*8，否则可能造成内存不足而程序崩溃或者运行很长时间而没有结果。</p> <p><code>eig(2) :</code></p> <p><code>eig(A, B)</code>计算 2 维方阵 A 相对于同样尺寸的方阵 B 的特征向量和特征值，也就是 $Av = \lambda v$，在这里，λ 是一个特征值，v 是一个特征向量。第二个参数，B，可以省略，其缺省值为 I 矩阵。这个函数返回一个包含两个成员的列表。第一个成员是特征向量矩阵，每一列是一个特征向量。第二个成员是一个对角矩阵，每一个对角线元素是一个特征值。注意运算这个函数非常耗费内存和 CPU 资源，如果在手机上运行，矩阵 A 的尺寸最好不要超过 6*6，如果在电脑上运行，最好不要超过 8*8，否则可能造成内存不足而程序崩溃或者运行很长时间而没有结果。</p>
<code>get_eigen_values</code>	<p><code>get_eigen_values(1) :</code></p> <p><code>get_eigen_values(A)</code>计算 2 维方阵 A 的特征值。这个函数返回所有特征值，包括重复的特征值的列表。注意运算这个函数非常耗费内存和 CPU 资源，如果在手机上运行，矩阵 A 的尺寸最好不要超过 6*6，如果在电脑上运行，最好不要超过 8*8，否则可能造成内存不足而程序崩溃或者运行很长时间而没有结果。</p> <p><code>get_eigen_values(2) :</code></p> <p><code>get_eigen_values(A, B)</code>计算 2 维方阵 A 相对于同样尺寸的方阵 B 的特征值，也就是 $Av = \lambda v$，在这里，λ 是一个特征值，v 是一个特征向量。第二个参数，B，可以省略，其缺省值为 I 矩阵。这个函数返</p>

	回所有特征值，包括重复的特征值 的列表。注意运算这个函数非常耗费内存和 CPU 资源，如果在手机上运行，矩阵 A 的尺寸最好不要超过 6*6，如果在电脑上运行，最好不要超过 8*8，否则可能造成内存不足而程序崩溃或者运行很长时间而没有结果。
invert	invert(1) : invert(x)返回方块 2 维矩阵 x 的逆矩阵，注意 x 中的元素可以为复数。
left_recip	left_recip(1) : left_recip(x)计算 x 的左除倒数，注意当前仅支持数值或二维矩阵。
rank	rank(1) : rank(matrix)返回矩阵的秩。比如，rank([[1,2],[2,4]])将返回 1。注意这里的矩阵不见得非要是方阵。
recip	recip(1) : recip(x)计算 x 的倒数，注意当前仅支持数值或二维矩阵。

这里需要注意几点：

第一，对于二维方阵来讲，recip、left_recip 和 invert 实际上是同一个函数，都是求取方阵 x（也就是这些函数的参数）的倒数，相当于计算表达式 $1/x$ 。

第二，Det 和 deter 是同一个函数的不同名字，换句话说，它们的使用方法和计算结果完全一样。

第三，所有的上述函数都支持复数矩阵。

以下是上述函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 math libs 子目录中的 examples.mfps 文件中找到）：

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

程序的运行结果如下：

```

[0. 170665027141073449368678271683842020455634872094878445912311433
5 +
3. 2610379517684265529500566832795301074088302158506118107692861942
i,
1. 0352846399044520080844175395063017457192992181908246617872107954
+
2. 0718315370680215726468326567580658394348957576054643511780204727
i, 1], [1, 1, -
1. 4604755542403403454627560718567442382313962988887573353544007743
+
0. 5989937399782723545797394498168180189451427644426167079112280372
i]],
[[1. 67711197606403032341325625275715448117055072235332757018694377
36 -
1. 6789870982611406187191411771858177830523877216751181685694659201
i, 0, 0], [0,
0. 0328711162943186887885582860890803241425541594114635568073761979
+
0. 5055670540754347904151221100125363969735197371948964385418564962
i, 0], [0, 0, -
0. 7099830923583490122018145388462348053131048817647911269943199715
-
8. 3365799558142941716959809328267186139211320155197782699723905761
i]]]

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. 6771119760640303234132562527571544811705507223533275701
869437736 -
1. 6789870982611406187191411771858177830523877216751181685694659201
i,
0. 0328711162943186887885582860890803241425541594114635568073761979
+
0. 5055670540754347904151221100125363969735197371948964385418564962
i, -

```

```
0. 7099830923583490122018145388462348053131048817647911269943199715
```

```
-
```

```
8. 3365799558142941716959809328267186139211320155197782699723905761  
i]
```

```
rank([[1, 2, 3], [4, 5, 8]]) = 2
```

第 6 节 表达式和微积分函数

将表达式和微积分函数放在一起，是因为这些函数都是对一个基于字符串的 MFP 表达式进行处理，函数的列表如下：

函数名	函数帮助信息
deri_ridders	<p>deri_ridders(4) :</p> <p>deri_ridders(expr, var, val, ord) 返回基于变量 var 的表达式 expr 在 var 等于 val 的时候的 ord 阶导数值。这个函数使用 Ridders 法进行计算。比如， deri_ridders("x**2+x", "x", 3, 2) 返回 2。</p>
derivative	<p>derivative(2) :</p> <p>derivative(expression, variable) 返回基于变量 variable 的表达式 expression 的导数表达式。注意 expression 和 variable 均为字符串。比如， derivative("x**2+x", "x") 返回一个字符串表达式 "2*x+1"。</p> <p>derivative(4) :</p> <p>derivative(expr, var, val, method) 返回基于变量 var 的表达式 expr 在 var 等于 val 的时候的导数值。参数 method 用于选择计算方法，如果是 true，意味着使用 Ridders 法进行计算，如果是 false，则只是简单地计算导数表达式在 val 时候的值。比如， derivative("x**2+x", "x", 2, true) 返回 5。</p>

evaluate	<p>evaluate(1...):</p> <p>evaluate(expr_string,var_string1,var_value1,var_string2,var_value2,...) 返回当基于字符串的变量 var_string1 等于 var_value1, var_string2 等于 var_value2, ...时, 基于字符串的表达式 expr_string 的值。注意变量 var_string1, var_string2, ...的值可以为任意类型。变量的数目可以为 0, 也就是说, evaluate("3+2")是合法的。</p>
integrate	<p>integrate(2):</p> <p>integrate(x,y)返回基于变量 y 的表达式 x 的不定积分, 表达式 x 和变量 y 均为字符串。注意如果表达式 x 不存在不定积分, 或者 x 过于复杂无法解出其不定积分, 本函数将会抛出异常。</p> <p>integrate(4):</p> <p>integrate(x,y,z,w)返回表达式 x 在变量 y 从 z 到 w 的积分值。表达式 x 和变量 y 均为字符串, z 和 w 可以为实数, 复数或字符串。注意本函数采用的积分算法是自适应 Gauss-Kronrod 积分法。</p> <p>integrate(5):</p> <p>integrate(x,y,z,w,v)返回表达式 x 相对于变量 y 从 w 到 z 的积分。计算时每一步步长为(w-z)/v。表达式 x 和变量 y 均为字符串, z 和 w 可以为实数, 复数或字符串, v 必须为正整数。注意如果 v 是 0, 则相当于执行 integrate(x,y,z,w)。</p>
product_over	<p>product_over(3):</p> <p>函数 product_over(x, y, z)计算基于字符串的表达式 x 中的变量取值从整数 y 逐步变化到整数 z 的值的乘积。y 和 z 必须是字符串的形式, 其中, y 必须写成一个赋值表达式, 比如 "a=10", 这里, a 是变量名。一个函数的例子为 product_over("x+1", "x=1", "10")。</p>

sum_over	<p>sum_over(3) :</p> <p>函数 <code>sum_over(x, y, z)</code> 计算基于字符串的表达式 <code>x</code> 中的变量取值从整数 <code>y</code> 逐步变化到整数 <code>z</code> 的值的总合。<code>y</code> 和 <code>z</code> 必须是字符串的形式，其中，<code>y</code> 必须写成一个赋值表达式，比如 "<code>a=10</code>"，这里，<code>a</code> 是变量名。一个函数的例子为 <code>sum_over("x+1", "x=1", "10")</code>。</p>
-----------------	---

这些函数中，`deri_ridders` 是使用 Ridders 法对一个函数求某一点的一阶，二阶或者三阶导数值。在列表中的例子 `deri_ridders("x**2+x", "x", 3, 2)` 表示求表达式 `x**2+x` 在 `x` 等于 3 时的二阶导数值。

`derivative` 用于求取一个函数的一阶导数的表达式或者一阶导数在某一点的值。列表中的第一个例子 `derivative("x**2+x", "x")` 表示求表达式 `x**2+x` 的一阶导数表达式，列表中的第二个例子 `derivative("x**2+x", "x", 2, true)` 表示求表达式 `x**2+x` 在 `x` 等于 2 时的一阶导数的值。需要注意的是最后一个参数 `true` 表示使用 Ridders 法求导数值，`false` 表示直接将 `x` 的值代入导数表达式中求值。如果不给出最后一个参数，其缺省值为 `true`。

`derivative` 也可以用于求取函数的高阶导数表达式。这需要对 `derivative` 函数进行连环调用，比如 `derivative(derivative("x**2+x", "x"), "x")` 给出表达式 `x**2+x` 的二阶导数表达式。

`sum_over` 相当于数学中的求和符号 Σ ，它的使用方法也和 Σ 完全一样。比如，列表中的例子 `sum_over("x+1", "x=1", "10")`，表示对表达式 `x+1` 求和，`x` 的变化范围是从整数 1 变化到整数 10，相当于数学表达式 $\sum_{x=1}^{10} (x+1)$ 。

类似 `sum_over`，`product_over` 相当于数学中的求积符号 Π ，它的使用方法也和 Π 完全一样，比如列表中的例子 `product_over("x+1", "x=1", "10")`，表示对表达式 `x+1` 求积，`x` 的变化范围是从整数 1 变化到整数 10，相当于数学表达式 $\prod_{x=1}^{10} (x+1)$ 。

`Evaluate` 函数是将一个字符串视为 MFP 表达式，然后求取该表达式的值。注意如果该表达式中有一个或多个未知变量，`evaluate` 函数可以增加参数给未知变量赋值，以便最后求得结果，比如

`evaluate("x+y+1", "x", 3, "y", 4)` 用于计算 $x+y+1$ 的值，并且给出了 x 的值为 3， y 的值为 4，所以最后的计算结果为 8。当然，如果没有未知变量，就不必增加用于赋值的参数了。

`Evaluate` 计算的 MFP 表达式还可以包括函数，无论是系统自带的函数还是用户定义的函数，比如 `evaluate("sind(30)")` 得到值为 0.5。

`Integrate` 函数则用于计算定积分或者不定积分。事实上，可编程科学计算器的计算积分的组件就是调用的这个函数。如果用这个函数计算不定积分比较简单，需要两个参数，第一个是被积分表达式，第二个是被积分变量，它们都是基于字符串的，比如

```
integrate("cos(x)", "x")
```

就是对 $\cos(x)$ 积分，得到的不定积分结果表达式也是一个字符串，也就是 " $\sin(x)$ "。

如果用于计算定积分，也有两种用法，第一种是采用高斯克朗德（Gauss-Kronrod）法，采用这种积分办法，能够处理起始和终止积分点为无穷的情况，也能够处理高频震荡函数，甚至能够处理某些奇异点的情况。代价就是计算速度比较慢。比如

```
integrate("exp(x)", "x", -inf, 0)
```

返回 $\exp(x)$ 从负无穷到 0 的积分（结果为 1），而

```
integrate("log(x)", "x", 0, 1)
```

返回 $\log(x)$ 从 0 到 1 的积分，注意这里 0 是奇异点，返回的结果为 -1.00000018，注意这里有计算误差。

对于大部分函数，如果没有奇异点，也不是高频震荡，积分范围也不包括无穷大，则可以使用普通的梯形求和积分法。这时，用户需要指定积分步数，如果积分步数为 0，或者积分范围包括无穷大，`integrate` 又自动回到了采用高斯克朗德法进行积分。一个普通的梯形求和积分法的例子为：

```
integrate("x**2+1", "x", -3+4i, 7-9i, 100)
```

，这个例子中，积分的起止点位于复数域（integrate 支持复数域上积分），积分步数为 100 步，最后的结果为 $-481.7345 - 225.69505 * i$ ，理论的结果为 $-481.6666667 - 225.6666667 * i$ 。可见，对于诸如 x^{**2+1} 之类的线性或者接近线形的函数，只要步数足够大，误差会很小。

通过嵌套 integrate 函数还可以实现高次积分，比如计算 $x*y$ 的积分，x 从 1 到 6，y 从 -4 到 3，计算表达式为：

```
integrate("integrate(\"x*y\", \"x\", 1, 6, 100)", "y", -4, 3, 100)
```

，结果为 -61.25。需要注意，由于高斯克朗德法速度比较慢，高次积分强烈不建议采用该积分法，这也是为什么在上述表达式中指定积分步骤的原因。

当然，也有可能，integrate 函数无法得到积分结果（比如，有些不定积分过于复杂或者不可积，或者有些定积分无法收敛），这时，integrate 函数会抛出异常，比如在命令提示符中运行下述语句块（直接将下述语句块拷贝粘贴到基于 JAVA 的可编程科学计算器的命令提示符中，按回车键即可运行）：

```
try  
print("integrate(\"e**(x**2)\", \"x\")", integrate("e**(x**2)", "x"))  
catch  
print("e**(x**2) cannot be integrated")  
endtry
```

用户可以看到 integrate 函数抛出异常，最后的结果为打印提示

```
e**(x**2) cannot be integrated
```

也就是 e 的 x 平方次方不可积。

最后要注意，以上介绍的所有表达式和积分函数，都能够读取已定义的变量的值，比如，在某个程序中已经声明了一个变量 b，它的值为 3，那么用户调用 evaluate("x+b", "x", 9)，就会返回 12。由于这个原因，用户在使

用上述函数的时候，最好能够保证函数内部定义的变量和程序中的变量的名字不同，免得发生冲突。这个在进行高次积分的时候特别要注意。

以下是上述函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `math libs` 子目录中的 `examples.mfps` 文件中找到）：

```
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 exprecalculus()

    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

    // 测试高次导数

    print("\nderivative(derivative(\"x*sin(x)\", \"x\"), \"x\") = " _

        + derivative(derivative("x*sin(x)", "x"), "x"))

    // test derivative value

    // 测试求取导数值

    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.

// 测试不可积分

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

    // 测试高次积分

    print("\nintegrate("\nintegrate("\\\"x*y\\\"", "\\\"x\\\"", 1, 6, 100)", "\\\"y\\\"", -4, 3, 100)
= ")

    print(integrate("integrate(\"x*y\", "x", 1, 6, 100)", "y", -4, 3, 100))

endf

```

上述例子的运行结果如下：

```

derivative("1/x**2*log(x) + 9", "x") = (-2)*log(x)*x**(-3)+x**(-3)

derivative("tanh(x)**-1", "x") = -(-
0.5)*2.71828182845904523536028747135266249775724709369995957496696
76277**x*(sinh(x)/cosh(x))**(-2)*sinh(x)*cosh(x)**(-2)+(-
0.5)*2.71828182845904523536028747135266249775724709369995957496696
76277**(-x)*(sinh(x)/cosh(x))**(-2)*sinh(x)*cosh(x)**(-2)+(-
0.5)*2.71828182845904523536028747135266249775724709369995957496696
76277**x*(sinh(x)/cosh(x))**(-2)/cosh(x)+(-
0.5)*2.71828182845904523536028747135266249775724709369995957496696
76277**(-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) =
1.91287092917527690172363463716465048491954803466796875

```

```

deri_ridders("x**0.5+sqrt(sin(x**2))", "x", 0.3, 3) =
7.1575232288636571107632429280365926329437264758531027037489228909

sum_over("1/(x-10)", "x=1", "9") = -
2.828968253968253968253968253968253968253968253968253968253968254

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) =
0.49999999999999800759152415811779636542182411236778807349767932

integrate("1/x**2", "x", 2, 50, 100) =
0.4847465087006575124658317917505673256758819785394978030710821748
e**(x**2) cannot be integrated

integrate("integrate(\"x*y\", \"x\", 1, 6, 100)", "y", -4, 3, 100) = -
61.25

```

注意，从 1.7 版的可编程科学计算器开始，引用空间的概念被加入，所以，在算微分和积分表达式时，给出的答案包括每个函数完整的引用路径。例如在旧版本中的 `log(x)` 在 1.7 版中为`::mfp::math::log_exp::log(x)`。这让答案显得有些冗长并且难于阅读。1.7.1 版改进了这个问题，在算微分和积分表达式时，给出的答案仅仅包括函数的最短引用路径。如果该函数在缺省引用空间内并且没有同名函数，则仅仅给出不包括引用路径的函数名。

第 7 节 统计、随机和排序函数

可编程科学计算器提供了用统计和随机过程以及排序的函数。这些函数和数学上的表示方法基本相同，用户可以很方便地使用：

函数名	函数帮助信息
avg	avg(0...) : 函数 avg(...) 返回任意个数参数的平均值。
beta	beta(2) : 函数 beta(z1, z2) 返回复数 z1 和 z2 的 beta 函数值。注意 z1 和 z2 的实部必须是正数。
gamma	gamma(1) : 函数 gamma(z) 返回复数 z 的 gamma 函数值。注意 z 的实部必须是正数。
gavg	gavg(0...) : 函数 gavg(...) 返回任意个数参数的几何平均数值。
havg	havg(0...) : 函数 havg(...) 返回任意个数参数的调和平均数值。
max	max(0...) : 函数 max(...) 返回任意数目参数中的最大值。
med	med(0...) : 函数 med(...) 返回任意数目参数的中位数。如果参数的个数为偶数个，返回中间两个参数的平均值。
min	min(0...) : 函数 min(...) 返回任意数目参数中的最小值。

quick_sort	quick_sort(2) : 函数 quick_sort (desc, original_list) 将拥有至少一个元素的向量 original_list 用快速排序法进行排序并返回排序后的向量。如果 desc 是 true 或者 1，按照从大到小 排序，否则 (false 或者 0) 按照从小到大排序。比如，输入 quick_sort (1, [5, 6, 7, 9, 4]) 得到 [9, 7, 6, 5, 4] 而输入 quick_sort (0, [5, 6, 7, 9, 4]) 的结果是 [4, 5, 6, 7, 9]。
ncr	ncr(2) : 函数 nCr(x, y) 计算有 x 个元素的集合 S 的 k 个元素组合的个数。注意 x, y 都是非负整数，x >= y。
npr	npr(2) : 函数 nPr(x, y) 计算有 x 个元素的集合 S 的 k 个元素排列的个数。注意 x, y 都是非负整数，x >= y。
rand	rand(0) : rand() 函数返回一个大于等于 0 小于 1 的随机浮点数。
stdev	stdev(0...) : 函数 stdev(...) 返回任意个数参数的标准差，注意这些参数是一个大的集合中的采样。
stdevp	stdevp(0...) : 函数 stdevp(...) 返回任意个数参数的标准差。
sum	sum(0...) : 函数 sum(...) 返回任意个数参数的总合。

这里需要注意 stdev 函数和 stdevp 函数的区别。假设这两个函数的参数都是同样的实数序列 $x_1, x_2, x_3, \dots, x_N$ ，那么 stdev 返回的是

$$\sqrt{\frac{1}{N-1}((x_1 - u)^2 + (x_2 - u)^2 + (x_3 - u)^2 + \dots + (x_N - u)^2)}$$

而 stdevp 返回的是

$$\sqrt{\frac{1}{N}((x_1 - u)^2 + (x_2 - u)^2 + (x_3 - u)^2 + \dots + (x_N - u)^2)}$$

这里， u 是 $x_1, x_2, x_3, \dots, x_N$ 的平均值。

以下是上述函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `math libs` 子目录中的 `examples.mfps` 文件中找到）：

```
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("\nnncr(8, 3) = " + ncr(8, 3))
print("\nnmpr(8, 3) = " + npr(8, 3))
print("\nrrand() = " + rand())

```

endf

上述例子程序运行结果如下：

```

avg(1, 5, 9, -6, 3, -18, 7) =
0.1428571428571428571428571428571428571428571428571428571429

beta(3.71, 23.55) =
0.0000279537392314725872716390423881975646941670888511331711318296

gamma(5.44 - 10.31i) =
0.0015360621732035695620552936894943183717820318136617456390043119
-
0.0279816213196075726360710743099268272949427989554500480691282345
i

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`

第 8 节 信号处理函数

为了方便电子电气工程师，MFP 提供了三个信号处理函数：conv（卷积），FFT（快速傅立叶变换）和 iFFT（快速傅立叶变换的逆变换），用法和示例如下：

函数名	函数帮助信息
<code>conv</code>	<p><code>conv(2) :</code></p> <p><code>conv(input_a, inputb)</code>返回 <code>input_a</code> 和 <code>input_b</code> 的卷积。<code>Input_a</code> 和 <code>input_b</code> 要么都是一维向量，要么都是二维矩阵。当前本函数仅仅支持一维和二维卷积，比如：</p> <p><code>conv([4,8,2,9],[5,3,8,9,6,7,8])=[20, 52, 66, 151, 139, 166, 181, 132, 79, 72]</code></p> <p><code>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</code></p>

	$+ 8 * i, 156 - 4 * i, 8], [1.2, 17.2, 28.8, 69.6, 0, -16]]$
FFT	<p>FFT(1...) :</p> <p>FFT(a, ...) 返回对一个数值向量作快速傅立叶变换后的值。注意数值向量中数值的个数必须是 2 的整数次方。如果参数 a 是一个数值序列，则本函数只可能拥有一个参数，返回值为对序列 a[0], a[1], ..., a[N-1] 作快速傅立叶变换的返回值。如果参数 a 仅仅是一个实数或者虚数，则本函数最少包含 2 个参数，而返回序列 a, optional_params[0], ... , optional_params[number_of_optional_params - 2], optional_params[number_of_optional_params - 1] 快速傅立叶变换后的值。注意返回值总是一个数组。</p> <p>函数例子：</p> <p>FFT(1, 2, 3, 4) 返回 [10, -2+2i, -2, -2 - 2i];</p> <p>FFT([1, 2, 3, 4]) 同样也是返回 [10, -2+2i, -2, -2 - 2i];</p>
IFFT	<p>IFFT(1...) :</p> <p>IFFT(a, ...) 返回对一个数值向量作快速傅立叶变换的逆变换后的值。注意数值向量中数值的个数必须是 2 的整数次方。如果参数 a 是一个数值序列，则本函数只可能拥有一个参数，返回值为对序列 a[0], a[1], ..., a[N-1] 作快速傅立叶变换德逆变换的返回值。如果参数 a 仅仅是一个实数或者虚数，则本函数最少包含 2 个参数，而返回序列 a, optional_params[0], ... , optional_params[number_of_optional_params - 2], optional_params[number_of_optional_params - 1] 快速傅立叶变换的逆变换后的值。注意返回值总是一个数组。</p> <p>函数例子：</p> <p>IFFT(10, -2 + 2i, -2, -2 - 2i) 返回 [1, 2, 3, 4];</p> <p>IFFT([10, -2 + 2i, -2, -2 - 2i]) 同样也是返回 [1, 2, 3, 4];</p>

以下是上述函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 `math libs` 子目录中的 `examples.mfps` 文件中找到) :

```
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
```

上述例子的运行结果如下:

```
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]
```

第9节 阶乘求值函数、判断质数函数和多项式求根函数

MFP 中求取一个非负整数的阶乘的函数为 factor，比如 factor(3) 得到 6。注意这个函数只有一个参数，如果该参数不是整数，将会被先截断转换为整数在进行计算，如果参数的值小于 0 或者不能被转换为整数，将会出错。

MFP 判断一个数是否为质数的函数式为 is_prime，比如 is_prime(3.3) 得到 false 而 is_prime(97) 得到 true。注意这个函数只有一个参数，并且该参数必须为实数，如果参数不是实数，将会报错。

MFP 一元多项式求根的函数为 roots。roots(a, ...) 返回一个多项式的根数列。如果 a 是一个包含 N 个元素的实数或虚数数列，则返回多项式

```
a[0] * x**N-1 + a[1] * x**N-2 + ... + a[N-2] * x + a[N-1] == 0
```

的根数列。如果 a 是一个单一的实数，此函数则必须拥有至少两个参数，返回多项式

$a * x^{n-a} + \sum_{i=1}^{n-a} a_i x^{n-i}$

的根数列。

需要注意的是，如果该多项式次数大于等于 4，根的计算是通过牛顿拉夫逊法给出的近似值。由于牛顿拉夫逊法需要迭代计算，运算时间会比较长（取决于设备的性能）。

例如，如果要计算多项式 $3 * x^2 - 4 * x + 1 = 0$ 的根，在命令提示符中输入命令：roots([3, -4, 1]) 获得的结果是 [1, 0.33333333]；

如果要计算多项式 $(1+2i) * x^3 + (7-6i) * x^2 + 0.54 * x - 4.31 - 9i = 0$ 的根，输入命令：roots(1+2i, 7-6i, 0.54, -4.31-9i) 获得的结果是 [0.79288607 + 3.9247084 * i, -0.56361748 - 0.78399569 * i, 0.7707314 + 0.85928729 * i]。

roots 函数和用 solve 程序块求解一元多项式得出的结果是一样的，但是由于 roots 语句不必对程序块的语法结构进行分析，它比用 solve 程序块的效率要高。

以下是上述函数的例子程序。本例子可以在本手册自带的示例代码所在目录中的 math libs 子目录中的 examples.mfps 文件中找到）：

```
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

```

上述例子的运行结果如下：

```
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
```

小结

MFP 编程语言提供了完整的数学和科学的函数。这些函数符合它们在相关领域的使用方法，简单易学。

MFP 编程语言还提供了用于表达式和微积分的函数，这些函数的共同特点是使用基于字符串的数学公式表达式作为参数。和后面将会提到的绘图函数不同的是，这些函数可以接受参数数学公式字符串中含有已经声明的变量。对于这种情况，这些函数将自动把已经声明的变量的数值代入表达式进行计算。

第5章 用MFP编程语言绘制图形

在第一章中，已经介绍了用智慧计算器或者专门的绘图工具绘制各种图形。这种绘图方法适合于广大非程序员，对编程不是很了解的用户。但必须指出的是，这种提供用户输入界面，由用户输入参数来绘图的办法，实质上只是将用户输入的参数转化为MFP的绘图函数，然后调用绘图函数进行绘图。如果用户能够直接操作这些绘图函数，对它们进行编程，可以绘制出更复杂，更适合用户需要的图形。

MFP的绘图函数，既可以在基于安卓的可编程科学计算器中运行，也可以在台式机或者笔记本电脑上基于JAVA的可编程科学计算器中运行，并且运行效果完全一样。为了文档撰写的方便，在本章中，所有的例子给出来的示例图都是在基于JAVA的可编程科学计算器中运行得到的结果，但是用户把代码拷贝到安卓上面运行，也可以得到相同的图像。

本章中，给出了调用可编程科学计算器绘图的很多实例，由于这些实例都需要调用一次MFP语言的绘图函数，也就是用一条MFP语句实现，所以，本章中的所有例子都实现在不带参数的函数`::mfpexample::plotGraphs`函数中。该函数的代码可以在本手册自带的示例代码所在目录中的`graph libs`子目录中的`examples.mfps`文件中找到）：

第1节 绘制表达式的图像

绘制表达式图像对用户来讲是最简单的一种绘图方式，相当于打开智慧计算器，输入表达式，然后点击绘图按钮。对表达式绘图，用户不用管绘制图形的范围，因为这个范围在图像绘制完成之后可以由用户动态调整，也不用管图像的颜色，标题，点或线的形态，这些由函数根据表达式的个数，特征自动地设置，更不用管绘制的图像的类型（是二维，三维还是极坐标图像），这个由函数根据变量的个数自动地加以判断。

MFP提供的绘制表达式的图形的函数为`plot_exprs`，具体用法如下：

函数`plot_exprs`分析最少1条，最多8条表达式以绘制2维或者3维图形（取决于表达式中未知变量的个数）。输入的表达式可以是一个等式，比如“`4*x+9 == y +z**2`”和“`log(x*y) == x`”，也可以是一个左侧为未知变量的赋值表达式，比如“`k= 3+ 7 * sin(z)`”，还可以是一个可以被看作为赋

值表达式的表达式，比如“ $9*\log(y)$ ”可以被看作“ $x = 9 * \log(y)$ ”。注意所有表达式中未知变量的总数不多于 3，每一条表达式中的未知变量的个数不能少于未知变量的总数减一。未知变量的起始范围可以在设置中设定，缺省是从 -5 到 5，但是使用者可以在图形绘制出来之后动态调整每个未知变量的范围。如果有两个未知变量并且其中一个是希腊字母 α 、 β 、 γ 或者 θ ，则 绘制极坐标图形而不是普通 2 维图形。本函数的一个例子是 `plot_exprs("4*x+sin(y)", "x*log(x)/log2(z)==y")`，`"4-y**2==(x**2 + z**2)"`。需要指出的是，如果是绘制二维隐函数表达式，这个函数最多能绘制出 4 个表达式解；如果绘制的是 3 维隐函数表达式，这个函数会根据情况，可能会求每一个变量的最多两组解，绘制出最多 6 个解表达式图形，这样一来，整个求解绘图过程会花费比较长的时间。并且，由于 `plot_exprs` 函数有内在的，最多绘制 8 条曲线（面）的限制，对隐函数绘图时，一个隐函数就可能需要 6 条曲面来绘制，所以，`plot_exprs` 的参数不能包含太多的隐函数表达式，否则会报出无法绘制太多曲线的错误。

比如，用户想在同一幅图上绘制两条曲线如下：

$$f(x) = \ln(x) + 2x - 6$$

以及

$$g(x) = x^3 + 0.9x$$

，调用函数时，注意到这两个表达式都并非隐函数，所以，表达式参数不必输入 $f(x)$ 和 $g(x)$ ，只用输入纯表达式部分（这里是等号右边的部分）既可。

但是，要注意 MFP 的表达式，操作符，包括乘号和函数调用时的括号，不可以省略，所以，表达式在 MFP 中必须写为 $\ln(x)+2*x-6$ ，而应该写成 $x**3+0.9*x$ ，注意在 MFP 中，次方符号是 $**$ 而不是 $^$ 。

最后，由于 `plot_exprs` 输入的参数必须为字符串，所以不要忘记给表达式加上引号，整个函数的调用语句为（第一种调用方法）：

```
plot_exprs("ln(x)+2*x-6", "x**3+0.9*x")
```

用户也可以将函数的调用语句写为（第二种调用方法）：

```
plot_exprs("y==ln(x)+2*x-6", "x**3+0.9*x==y")
```

，或者（第三种调用方法）：

```
plot_exprs("y=ln(x)+2*x-6", "y=x**3+0.9*x")
```

。第二种和第三种调用方法都设定了 y 变量，这里的 y 变量指的是待绘图的表达式在 y 坐标轴上的值，相当于待绘图的表达式中的 $f(x)$ 和 $g(x)$ 。使用单一的 y 变量而不是两个不同的变量（比如 yf 和 yg ）来代表 $f(x)$ 和 $g(x)$ 的原因在于，我们希望在同一张图上绘制这两个表达式，所以 $f(x)$ 和 $g(x)$ 的值的投影都在同一个坐标轴（也就是 y 代表的坐标轴）上。如果用两个不同的变量分别代表 $f(x)$ 和 $g(x)$ ，意味着 $f(x)$ 和 $g(x)$ 的值的投影在不同的坐标轴上，这样就不是绘制二维曲线，而是三维曲线了。

用户可能还注意到，在第二种调用方法中，我们使用的是等号（ $=$ ）， y 既可以在等号的左边，也可以在等号的右边；而在第三种调用方法中，我们使用的是赋值号（ $=$ ）， y 只能在赋值号的左边。这种要求完全符合 MFP 的规则——我们可以让一个表达式等于某个变量的值（变量和表达式分置等号两侧，不管谁左谁右），或者把表达式的值赋给某个变量（变量位于赋值号的坐标，表达式位于右边），但我们不能够把变量的值赋给表达式。

函数绘制出来的图像如下。用户可以拖动图像对图像进行上下左右的平移，也可以点击放大缩小按钮来对图像进行缩放，注意，无论平移还是缩放，都会造成绘图范围发生变化，一些点将会被重新计算，所以如果是在手机上运行，用户可能会感觉到时延，如果图像很复杂，计算量很大，用户甚至可能会觉得有点卡。但在电脑上，对这些图像的处理，应该是轻而易举。

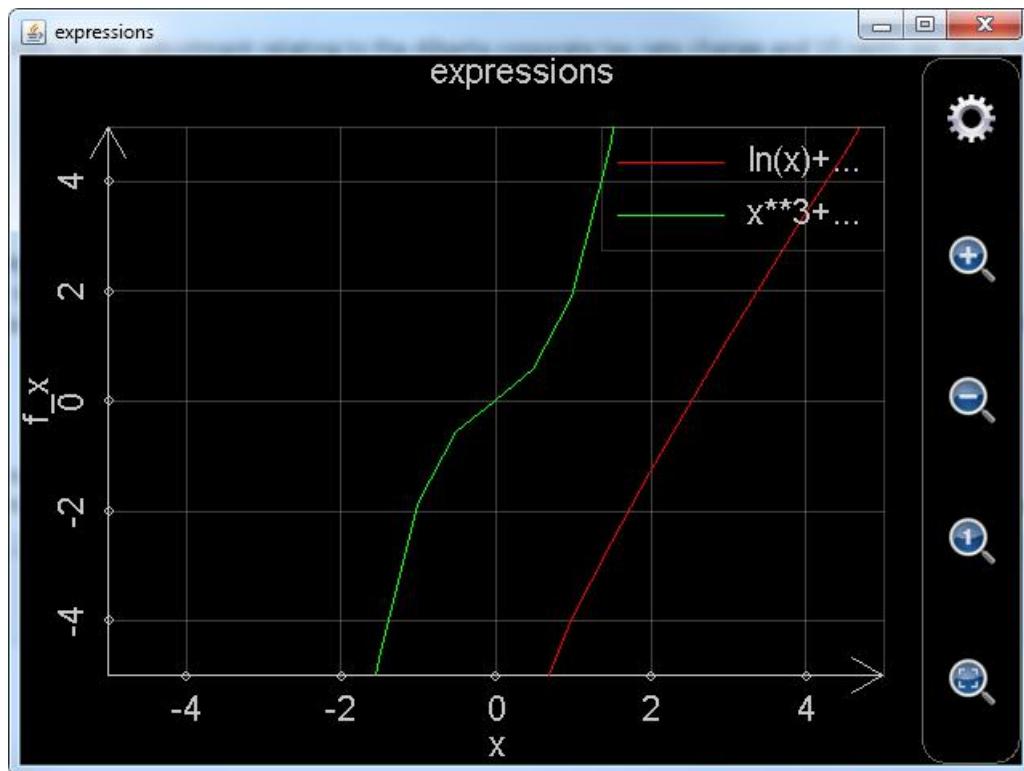


图 5.1：调用 `plot_exprs` 函数绘制表达式二维图形。

用户可能还会觉得，绘制出的图像，特别是绿色的曲线，不够平滑。这时就需要点击图上的齿轮按钮，对图像设置进行调整，点击齿轮按钮后出现的设置对话框参见下图。图中，红色长方形圈出的部分的左边用于设定绘制图形的步数，也就是在绘制范围内计算多少个点，步数越多，图像越平滑，但计算花费的时间越长，用 `plot_exprs` 函数绘制二维图像，缺省步数为 20 步；红色长方形圈出的部分的右边用于设定是否自动侦测奇异点，显然，侦测奇异点需要花费更多的计算时间，用 `plot_exprs` 函数绘制二维图像，缺省是不侦测奇异点的。如果用户用智慧计算器作图，缺省步数为 100 步，并且会自动侦测奇异点，所以，智慧计算器做出的表达式图像会更平滑，更逼真，当然，这也是有时候用户抱怨用智慧计算器作图比较“卡”的原因。

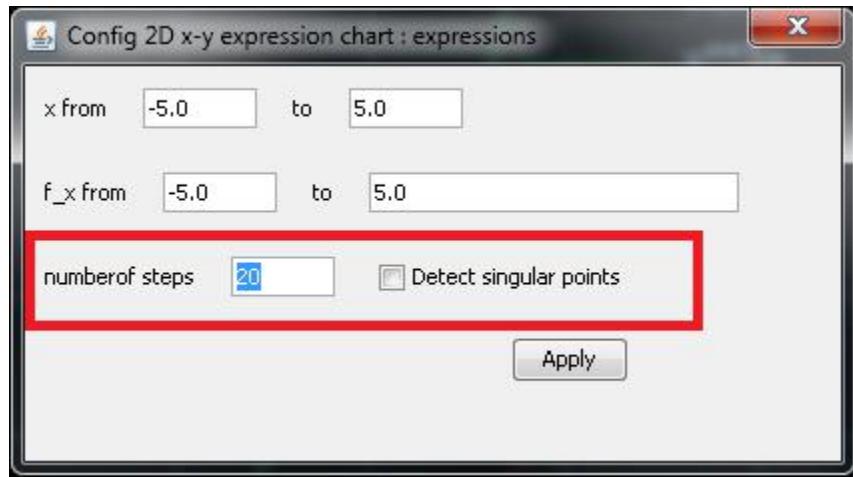


图 5.2：设置二维图像。

再比如，我们想绘制如下表达式的图像：

$$r^2 = \alpha^2 + 9$$

$$r = \cos \alpha$$

$$\alpha = \sin r$$

则可以调用函数

```
plot_exprs("r**2==alpha**2+9", "r==cos(alpha)", "alpha==sin(r)")
```

或者写成

```
plot_exprs("r**2==alpha**2+9", "cos(alpha)", "sin(r)")
```

在这里， $r == \cos(\alpha)$ 和 $\alpha == \sin(r)$ 的左侧变量和等于号都可以省略，MFP 会根据三个表达式所包含的总的变量个数和名称将表达式 $\cos(\alpha)$ 和 $\sin(r)$ 自动补全，但 $r**2 == \alpha**2 + 9$ 的任何一个部分都不能省略，原因是 $r**2 == \alpha**2 + 9$ 是隐函数等式。

函数绘制出来的图像如下。由于表达式中含有希腊字母 α ，并且总的未知变量的个数为 2，所以最终绘制出来的图像是极坐标图形。注意下图已经对图像进行缩小处理，并且绘图步数也增加到 200，否则图像会比较难看。

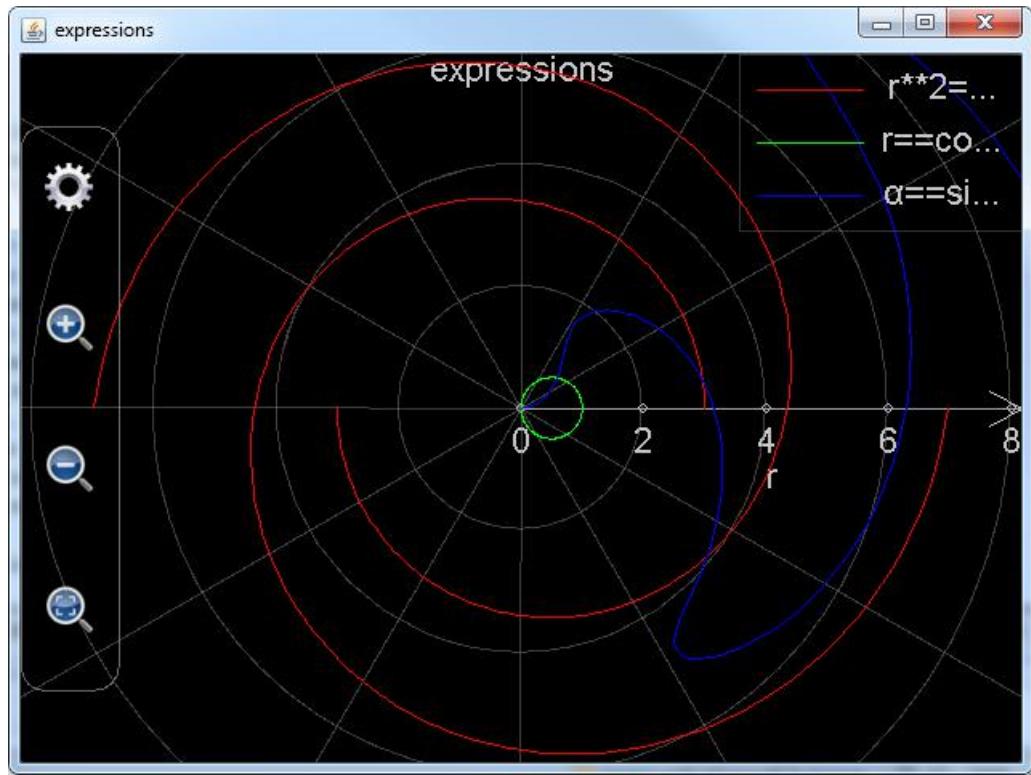


图 5.3：调用 `plot_exprs` 函数绘制表达式极坐标图形。

用户还需注意的是，在调整极坐标图形绘制范围的时候，幅值（也就是 r ）方向的绘制范围会发生改变，但是幅角（也就是 α ）的绘制范围永远是从 -2π 到 2π ，不会发生改变。这也符合极坐标图形的特点：放大缩小图像不会对幅角的范围有任何影响。

再给出绘制三维图形的例子。比如用户想绘制出一个椭球体

$$x^2 + 2y^2 + z^2 = 20$$

，和一个切割球体的曲面

$$z = \ln(3x^2 + y^2 + 2y + 2) \sin\left(\frac{xy}{10}\right)$$

，则输入的表达式为

```
plot_exprs("x**2+2*y**2+z**2=20", "z=ln(3*x**2+y**2+2*y+2)*sin(x*y/10)")
```

或者省去非隐函数的变量部分（如上所述，隐函数任何部分都不能省略）

```
plot_exprs("x**2+2*y**2+z**2==20", "ln(3*x**2+y**2+2*y+2)*sin(x*y/10)")
```

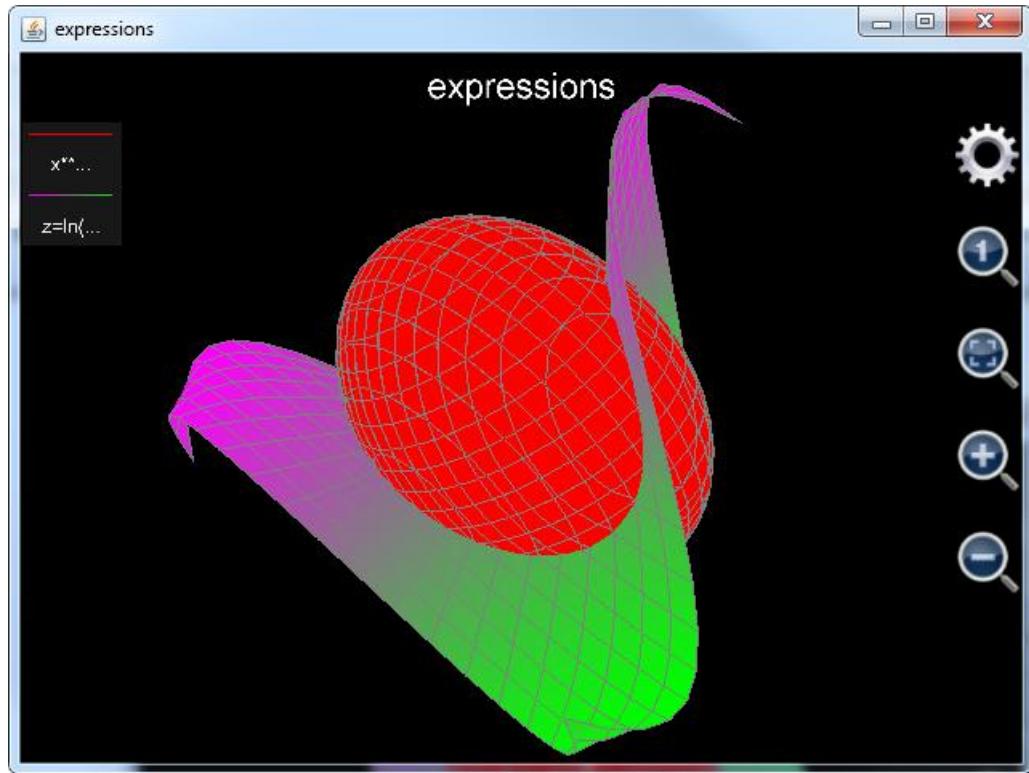


图 5.4：调用 plot_exprs 函数绘制表达式三维图形。

和二维图像类似，用户可以放大缩小图像，但是，用户用鼠标或者单个手指在图像按住并滑动不是拖动图像，而是旋转图像。所以，如果用户想要设置绘图范围，唯一的办法是点击齿轮按钮。点击齿轮按钮出现的设置对话框如下：

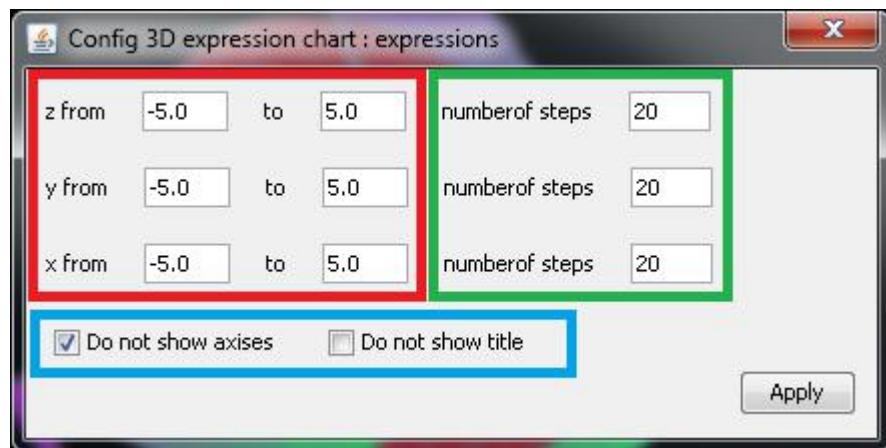


图 5.5： 设置表达式三维图形。

在设置对话框中，红色矩形所包含的部分用于设置在各个坐标轴方向的绘图范围，绿色矩形所包含的部分为每个坐标轴方向上的步数，也就是每个坐标轴方向上计算并绘制多少个点，蓝色举行所包含的部分为设置是否隐藏坐标轴和标题，以方便用户观察图像。注意在 1.6.7 版以前，用户只能选择同时隐藏坐标轴和标题或者同时显示坐标轴和标题，默认状态为显示坐标轴和标题；从 1.6.7 版开始，用户可以分开设置坐标轴和标题的隐藏或显示状态，默认状态为显示标题但隐藏坐标轴。

还要注意，如果是对隐函数作图，图像有可能只有一种颜色，而如果是对非隐函数作图，图像曲面的颜色是渐变的。但无论是哪种情况，用户都无法具体决定曲面使用哪种颜色。

最后给出另外一个绘制三维图形的例子：

```
plot_exprs("x**2-z**2==20", "x**2-y**2==6")
```

，绘制的图形如下（坐标轴和标题已经隐藏）：

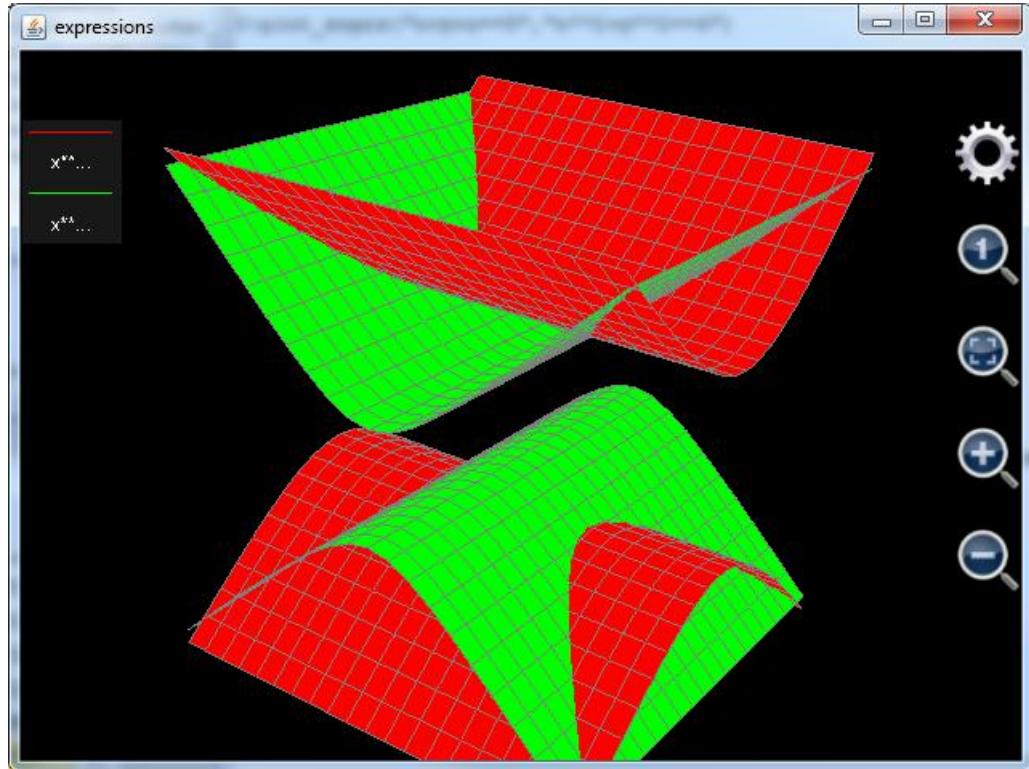


图 5.6： 根据总的未知变量的个数绘制出三维图形。

。照道理来讲， $x^{**2}-z^{**2}=20$ 和 $x^{**2}-y^{**2}=6$ 实际上都是二维曲线，如果调用 `plot_exprs` 分别单独绘制它们的图像，用户看到的是两条在二维平面上的双曲线。但是，由于这里 `plot_exprs` 要一起绘制这两个表达式，而这两个表达式中，总的变量的个数为 3（包括 x 、 y 和 z ），所以，最后绘制的图像是三维的而不是二维的。

最后还要指出一点，`plot_exprs` 函数是用 MFP 语言编写的函数，`plot_exprs` 函数的各个参数（也就是待绘图的表达式）只是原封不动地传递给了更底层的函数。由于这个原因，和第 4 章第 6 节所介绍的表达式和微积分函数不一样的是，用户在调用函数之前所定义的变量如果出现在 `plot_exprs` 的表达式中，`plot_exprs` 函数不会自动地对其求值。比如用户定义了一个变量 a ，它的值为 3，然后调用 `plot_exprs("x+a")` 绘制图形，MFP 不会把“ $x+a$ ”视为“ $x+3$ ”而在二维平面上绘制出一条直线，MFP 的做法是将 a 看作和 x 一样的一个变量而在三维空间中作出一个平面。所以，如果用户想动态地绘制 $x+a$ 的图像，每次调用 `plot_exprs` 时，就要动态地更新表达式字符串，比如，可以调用 `plot_exprs("x"+a)`，这样， a 当前的数值就会被自动地添加到字符串“ $x+$ ”后面从而得到新的字符串为“ $x+3$ ”。同样地，`plot_exprs` 函数内部只能看到 MFP 的缺省引用空间，也就是顶级引用空间和`::mfp` 以及其下级引用空间。调用 `plot_exprs` 的函数所能看到的引用空间无法传递到 `plot_exprs` 函数内部。这样一来，在调用 `plot_exprs` 的函数看来完全没有问题的函数参数，传递给 `plot_exprs` 后就可能会出现找不到函数的错误。比如用户自定义了一个引用空间，名字为`::aaaaa`，在这个引用空间内用户定义了一个函数叫做 `aaaaaF()`，然后用户在调用 `plot_exprs` 作图之前通过 `using citingspace` 语句声明使用引用空间`::aaaaa`，但这个声明并不能够简化 `plot_exprs` 的调用，用户还是必须在 `plot_exprs` 的字符串参数中给出 `aaaaaF()` 函数的绝对路径，也就是，

```
plot_exprs("::aaaaa::aaaaaF(x)")
```

，而 `plot_exprs("aaaaaF(x)")` 则会出错，原因是在 `plot_exprs` 函数内部引用空间`::aaaaa` 并不在搜索列表上。

除了 `plot_exprs`，后面的章节中介绍的绘图函数，也有很多是用 MFP 编写然后调用更底层的函数的，所以，建议用户在绘制包含预定义变量的表达

式的图像时，总是使用上述办法，并且在给出用户自定义函数的绝对引用空间路径，以保证代码的一致性。

第 2 节 绘制常规坐标系下的二维图像

绘制常规坐标系下的二维图形，MFP 提供了以下函数：

函数名	函数帮助信息
plot2dex	<p>plot2dex(6...) :</p> <p>函数 plot2DEX 调用 plot_multi_xy 以绘制由最多 8 条 2-D 曲线所构成的图像。其包括以下参数：1.图像名字（图像文件名）；2.图像标题；3.X 轴标题；4.Y 轴标题；5.图像背景色；6.是否显示网格；7.曲线标题；8.曲线数据点颜色；9.曲线数据点形状；10.曲线数据点大小；11.曲线连接线颜色；12.曲线连接线类型；13.曲线连接线粗细；14.t 起始位置；15.t 终止位置；16.t 的间隔；17.以 t 为变量的 X 的表达式；18.以 t 为变量的 Y 的表达式....。其中，每增加一条新的曲线，需要添加 12 个参数（也就是参数 7 到 18），最多定义 8 条曲线。另外要注意图像背景色，曲线数据点的大小，曲线连接线颜色以及曲线连接线类型还没有被实现，曲线连接线粗细仅支持 0（意味着没有连接线连接数据点）和非 0（意味着有连接线连接数据点）。本函数的一个例子为：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)")。</p>
plot_2d_curves	<p>plot_2d_curves(6...) :</p> <p>函数 plot_2d_curves 绘制由最多 1024 条 2-D 曲线所构成的图像。其包括以下参数：1.图像名字（图像文件名）；2.图像标题；3.X 轴标题；4.Y 轴标题；5.图像背景色；6.是否显示网格（注意这个参数是一个字符串，其值为 "true" 或者 "false"）；7.曲线标题；8.曲线数据点颜色；9.</p>

	<p>曲线数据点形状； 10.曲线数据点大小； 11.曲线连接线颜色； 12.曲线连接线类型； 13.曲线连接线粗细； 14.内部变量的名字（通常为"t"）； 15.内部变量的起始位置； 16.内部变量的终止位置； 17.内部变量的每一步变化间隔； 18.基于内部变量的 X 的表达式； 19.基于内部变量的 Y 的表达式...。其中，每增加一条新的曲线，需要添加 13 个参数（也就是参数 7 到 19），最多定义 1024 条曲线。另外要注意图像背景色，曲线数据点的大小，曲线连接线颜色以及曲线连接线类型还没有被实现，曲线连接线粗细仅支持 0（意味着没有连接线连接数据点）和非 0（意味着有连接线连接数据点）。本函数的一个例子为： <code>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)")</code>。</p>
<code>plot_2d_data</code>	<p><code>plot_2d_data(16)</code>：</p> <p>函数 <code>plot_2d_data</code> 分析最少 1 组，最多 8 组数值向量，每组数值向量将会被绘制为一条曲线。输入的参数个数可以为 1 个（绘制一条曲线），2 个（绘制一条曲线），4 个（绘制 2 条曲线），6 个（绘制 3 条曲线），8 个（绘制 4 条曲线），10 个（绘制 5 条曲线），12 个（绘制 6 条曲线），14 个（绘制 7 条曲线），16 个（绘制 8 条曲线）。每一个参数都是一个数值向量（也就是一维矩阵）。如果只有一个参数，该参数中的每一个元素将会是绘制出的曲线中的一个点，否则，奇数号参数决定曲线中的每一个点的 x 值，偶数号参数决定每一个点的 y 值。注意决定 x 值的参数中包含的元素个数应该和决定 y 值的参数中包含的元素相同。函数例子包括 <code>plot_2d_data([5.5, -7, 8.993, 2.788])</code> 以及 <code>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])</code>。</p>
<code>plot_multi_xy</code>	<p><code>plot_multi_xy(2...)</code>：</p> <p><code>plot_multi_xy</code>(包含≥ 2 个参数) 用于绘制 2 维或极坐标图</p>

像，每个图像最多包括 1024 条曲线。参数 1 为图像名字，参数 2 为图 像设置，该参数是一个字符串，比如 "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"。注意 chart_type 的值是 multiXY（用于绘制二维图像）或者 multiRangle（用于绘制极坐标图像），x_labels 和 y_labels 分别代表 x 和 y 轴上有多少刻度标记（对于极坐标图像是 R 轴有多少刻度标记，幅角的刻度标记不可设）。从参数 3 开始，每 3 个参数定义一条曲线，在这 3 个参数中，第一个参数为曲线设置，第二个参数是包含所有 x 数值（或者 R 数值）的向量，第三个参数是包括所有 y 值（或者幅角值）的向量。曲线设置参数为一个字符串，比如 "curve_label:cv2;point_color:blue;point_style:point;point_size:1;line_color:blue;line_style:solid;line_size:1"。另外注意 x 和 y 的每一个数值都必须为实数，x 和 y 的数值个数必须一致。本函数不返回数值。本函数的一个例子是 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])。

其中，plot2dex 和 plot_2d_curves 是用来在指定范围内会制 2 维表达式曲线，plot_2d_data 用于绘制二维数据图像。Plot_multi_xy 为更底层的函数，它被 plot2dex 和 plot_2d_data 函数所调用。

需要注意的是，plot_2d_curves 是从 1.6.7 版中才将接口暴露给用户的函数，这个函数在 1.6.6 及其以前版本中也存在，但是绘制曲线数目最多只

能有 8 条。从 1.6.7 版开始，plot_2d_curves 可以绘制曲线的数目增加到 1024 条，并且由于 plot_2d_curves 由 JAVA 实现，速度远比 plot2dEx 快，强烈建议用户使用 plot_2d_curves 来取代 plot2dEx。

以上 4 个函数，也包括从本节开始往后的所有绘图函数，和 plot_exprs 都有一个很大的区别。Plot_exprs 绘制表达式图像时不设定绘图范围，而是根据用户平移缩放图像，动态地调整绘图范围，图像上的点在绘图范围调整时自动重新计算。而以上 4 个函数，也包括从本节开始往后的所有绘图函数，在图像生成的时候，绘图的范围已经确定，即便用户平移缩放图像，绘图的范围也不会改变，图像上的点也不会重新计算。由于这个原因，用户操作由这些函数绘制出来的图形，会很平滑，不会有“卡”的感觉。

Plot_2d_curves 函数事实上就是安卓上的可编程科学计算器的独立的“绘制图形” -> “绘制二维图像”工具的函数版。它所需要的参数中，前 6 个参数用于设置图像本身，分别为图像的文件名（文件扩展名.mfpc 会被自动加上，不用用户输入），图像的标题，x 轴的名字，y 轴名字，背景色和是否绘制网格。注意，这些参数均为字符串，其中参数是否绘制网格为是一个字符串代表的布尔值（“true”或者“false”），而背景颜色所支持的字符串包括“white”（白色），“black”（黑色），“red”（红色），“green”（绿色），“blue”（蓝色），“yellow”（黄色），“cyan”（青色），“magenta”（紫红色），“dkgray”（深灰色）以及“ltgray”（浅灰色），缺省为黑色。

从第 7 个参数开始，每 13 个参数用于设置一条曲线。它们分别是

1. 曲线的名字（基于字符串）；
2. 曲线上的点的颜色（基于字符串，颜色选择范围和背景色一样）；
3. 曲线上的点的形状（基于字符串，可选的形状为“point”（点），“circle”（圆圈），“triangle”（三角形），“square”（方形），“diamond”（菱形）以及“x”（对角叉））；
4. 曲线上的点的大小（这个设置项必须为正整数，但还没有实现，用户随便填一个正整数即可）；

5. 点和点之间的连接线的颜色（基于字符串，颜色选择范围和背景色一样）；
6. 点和点之间的连接线的形态（基于字符串，但还没有实现，用户填入“solid”即可）；
7. 点和点之间的连接线的粗细（必须是一个非负整数，如果为 0，线将不会被绘制）；
8. 内部变量的名字（通常为“t”）
9. 变量 t（如果内部变量名字被命名为 t 的话，否则就是其他的变量名）的变化范围的起始值（必须是一个实数。这里的 t，和前面用基于安卓的可编程科学计算器上独立的“绘制图形”工具绘制二维图像时所使用的 t 是一样的。关于 t 的详细说明，参见第 1 章第 4 节中对于 t 的解释和定义）；
10. 变量 t 的变化范围的终止值（必须是一个实数）；
11. 变量 t 的变化的步长（必须是一个实数，t 的变化范围除以变化步长加 1 就是要绘制的点的个数。注意用户可以将其设置为 0，设置为 0 意味着由函数来决定步长）；
12. X 坐标变化对于 t 的函数（基于字符串。关于 X(t) 的详细说明，参见第 1 章第 4 节中的解释）；
13. Y 坐标变化对于 t 的函数（基于字符串。关于 Y(t) 的详细说明，参见第 1 章第 4 节中的解释）；

由于 plot_2d_curves 函数最多可以绘制 1024 条曲线，它的参数个数最多可以达到 $6+13*1024$ 等于 13318 个。事实上，用户完全可以对 plot_2d_curves 的参数进行编程，绘制出比较复杂的图形。

举个例子，比如用户想用 plot_2d_curves 绘制一个椭圆和一条抛物线，椭圆的函数为

$$4x^2 + y^2 = 16$$

，抛物线的函数为

$$y = \frac{x^2}{2.5} - 4x + 6$$

。对于绘制椭圆，用户可以设置 x 等于 $2\cos(t)$ ， y 等于 $4\sin(t)$ ， t 的变化范围是从 0 到 2π 步长为 0.02π 。对于绘制抛物线，用户可以设置 x 等于 t ， y 等于 $t^2/2.5 - 4t + 6$ ， t 的变化范围是从 -5 到 5，步长为 0.3。整个函数的调用如下：

```
Plot_2d_curves("chart 1", "plot2dEx 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)")
```

。绘制出的图形如下。用户可以随意拖动或者缩放图像，图像的变化非常平滑，没有任何迟滞。但要注意，在绘图范围之外（对于抛物线图像， t ，事实上也就是 x 小于 -5 或者大于 5 时），是没有图形被绘制的，尽管事实上抛物线的伸展范围远远超过图像的绘图范围）。

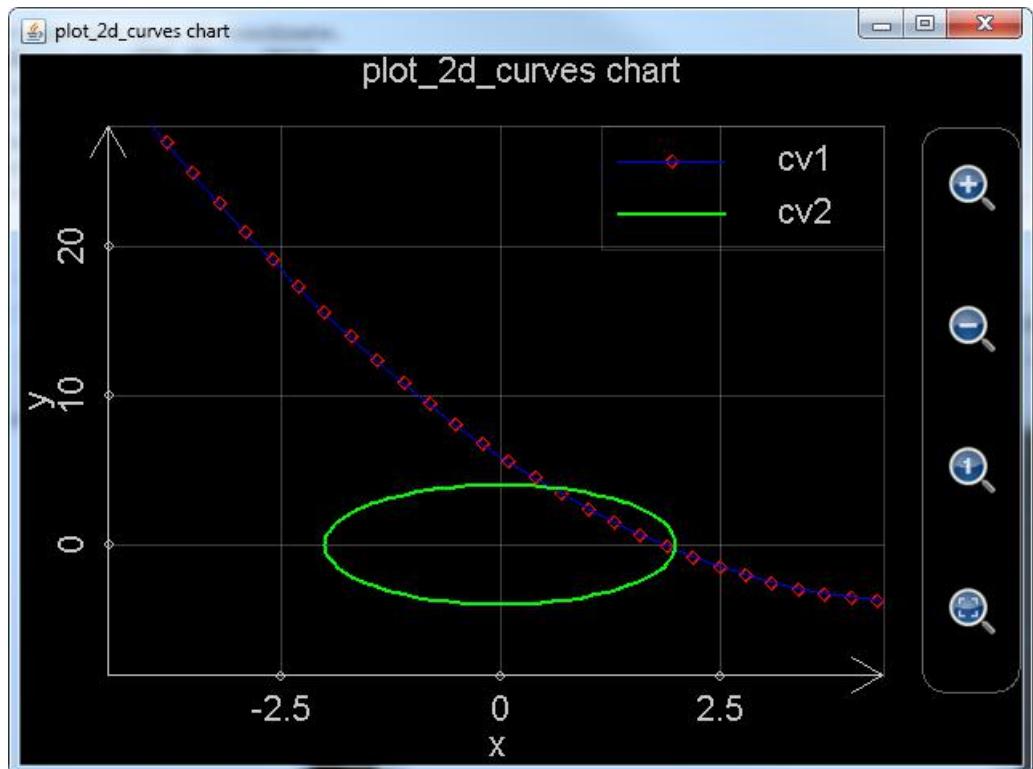


图 5.7：调用 `plot_2d_curves` 绘制指定范围的二维图像。

还要注意，这幅图像已经被可编程科学计算器自动保存以方便用户日后再次打开。如果这幅图像是在安卓系统上绘制的，那么用户点击主界面的“管理图形文件”图标，则可以进入图像文件夹。由于在调用 plot_2d_curves 时已经给图形文件命名为 chart 1，所以生成的图形文件的文件名应该是 chart 1.mfpc。用户找到它，长按该文件所对应的图标，则可以再次打开它。如果这幅图像是用基于 JAVA 的可编程科学计算器绘制的，在电脑上，我们可以找到 JAVA 的可编程科学计算器所在的 AnMath 目录的 charts 子目录，chart 1.mfpc 保存在该目录中。用户如果想打开它，需要在基于 JAVA 的图形界面可编程科学计算器中，选择“工具”菜单→“观看图像”子菜单，或者按下 Ctrl+O 快捷键来打开此文件。

再举个例子。比如，用户想要绘制正三角形。正三角形的三条边的表达式分别为

$$y = \sqrt{3}x + 2 \dots \dots \dots (-\sqrt{3} \leq x \leq 0)$$

$$y = -\sqrt{3}x + 2 \dots \dots \dots (0 \leq x \leq \sqrt{3})$$

以及

$$y = -1 \dots \dots \dots (-\sqrt{3} \leq x \leq \sqrt{3})$$

。那么可以调用 plot_2d_curves 绘制三条线作为三角形的三条边。第一条线的表达式为 $x=t$ 和 $y=\sqrt{3}t+2$ ， t 从 $-\sqrt{3}$ 到 0 步长为 0.02 ；第二条线的表达式为 $x=t$ 和 $y=-\sqrt{3}t+2$ ， t 从 0 到 $\sqrt{3}$ 步长为 0.02 ；第三条线的表达式为 $x=t$ 和 $y=-1$ ， t 从 $\sqrt{3}$ 到 $\sqrt{3}$ 。整个调用语句如下：

```
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")
```

。绘制出来的图形如下：

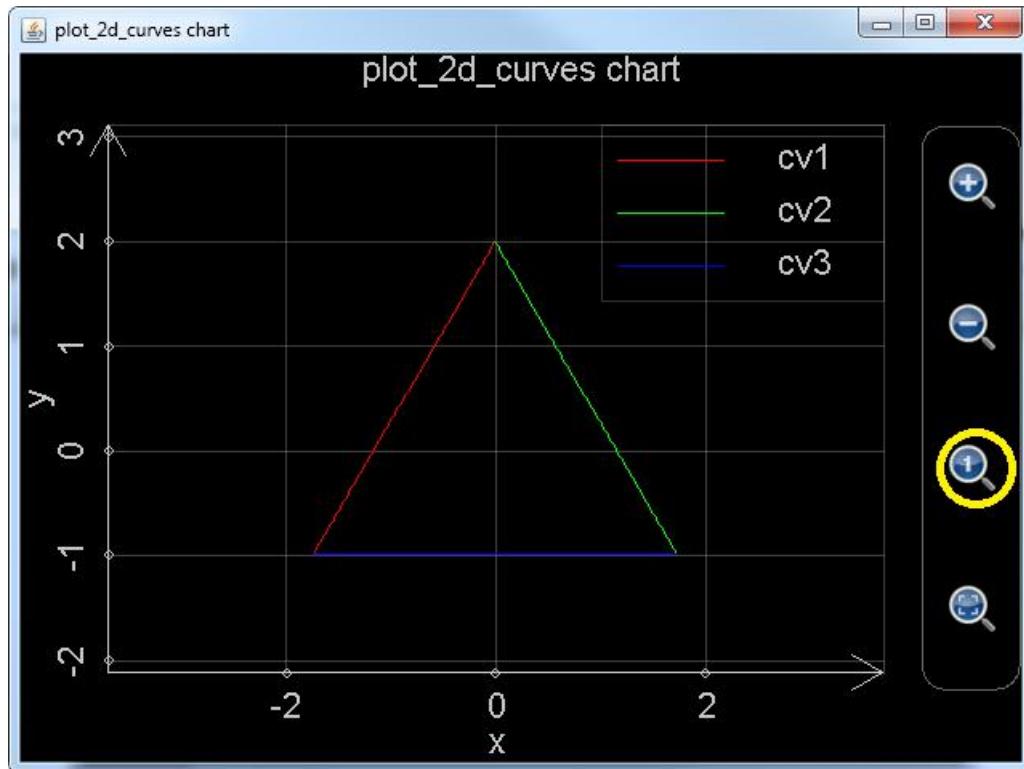


图 5.8：调用 plot_2d_curves 绘制三角形。

需要注意，图像刚刚生成时，由于 x 轴和 y 轴的坐标单位的长度不同，所以看起来并不像一个正三角形，这时，用户可以点击下图黄色圈圈中的放大镜里面有一个小 1 的按钮，自动调整 x 和 y 轴的坐标单位。这样得到的图形，就是标准的正三角形了。

Plot_2d_data 函数则是用于绘制二维数据曲线图。它的每一个参数都必须是一个一维数组，数组的每一个元素都必须是实数。如果参数只有一个，那么该数组的第一个元素所对应的数据点在 x 轴的坐标为 1，在 y 轴的坐标为第一个元素的值，第二个元素所对应的数据点在 x 轴的坐标为 2，在 y 轴的坐标为第二个元素的值，…，以此类推，有多少个元素就有多少个数据点。比如，以下命令绘制出一条折线，折线的连接点为 (1, 1)，(2, 7)，(3, 8)，(4, 6)：

```
Plot_2d_data([1, 7, 8, 6])
```

，绘制出的图形如下：

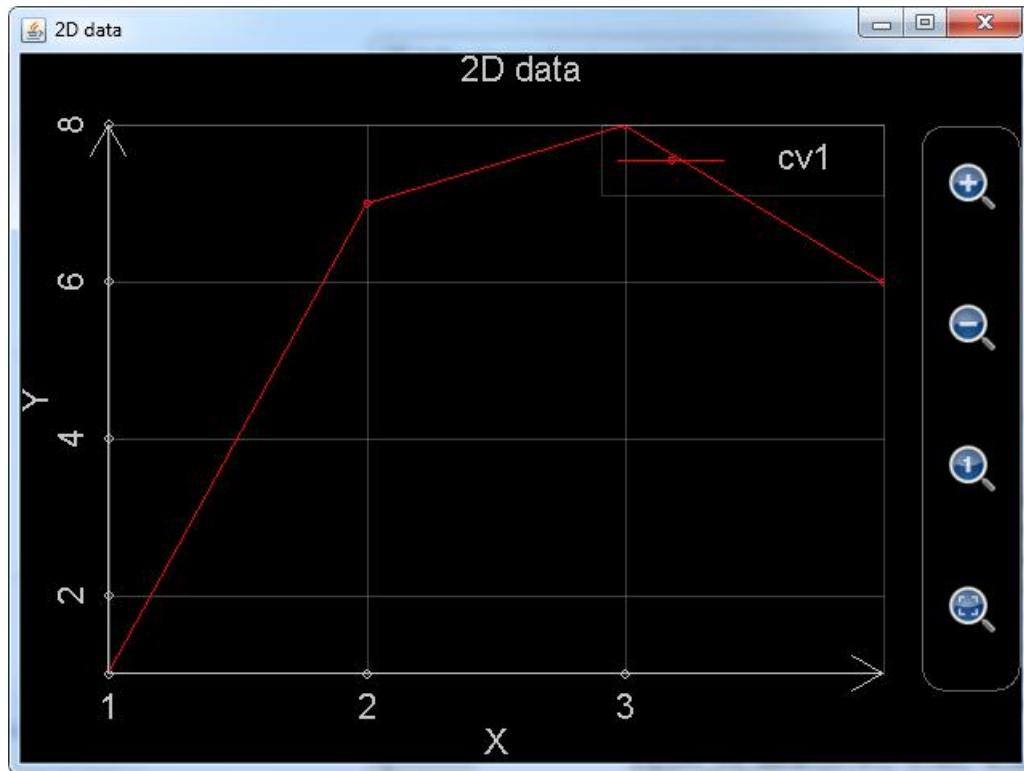


图 5.9：调用 `plot_2d_data`，仅使用一个参数绘制数据图。

如果参数多于一个，那么参数个数必须为偶数个，并且是两个两个一组，每一组中的两个参数为元素个数相同的一维数组，分别对应该组参数所代表的数据点集合的 x 坐标和 y 坐标集，比如，想绘制数据点集 $(-1.71, 6.24)$ ， $(8.93, -7.08)$ ， $(3.11, 5.85)$ ， $(4.28, -5.76)$ 以及 $(5.99, -3.24)$ ，所需要的两个参数为 $[-1.71, 8.93, 3.11, 4.28, 5.99]$ 和 $[6.24, -7.08, 5.85, -5.76, -3.24]$ 。

以下语句绘制了两个数据集，这两个数据集中的点的个数并不相同，并且第二个数据集种，y 的值有一个为 Nan（无定义数），Nan 的效果相当于数据集的点之间的连接线在这个点断开：

```
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])
```

绘制出的图形如下：

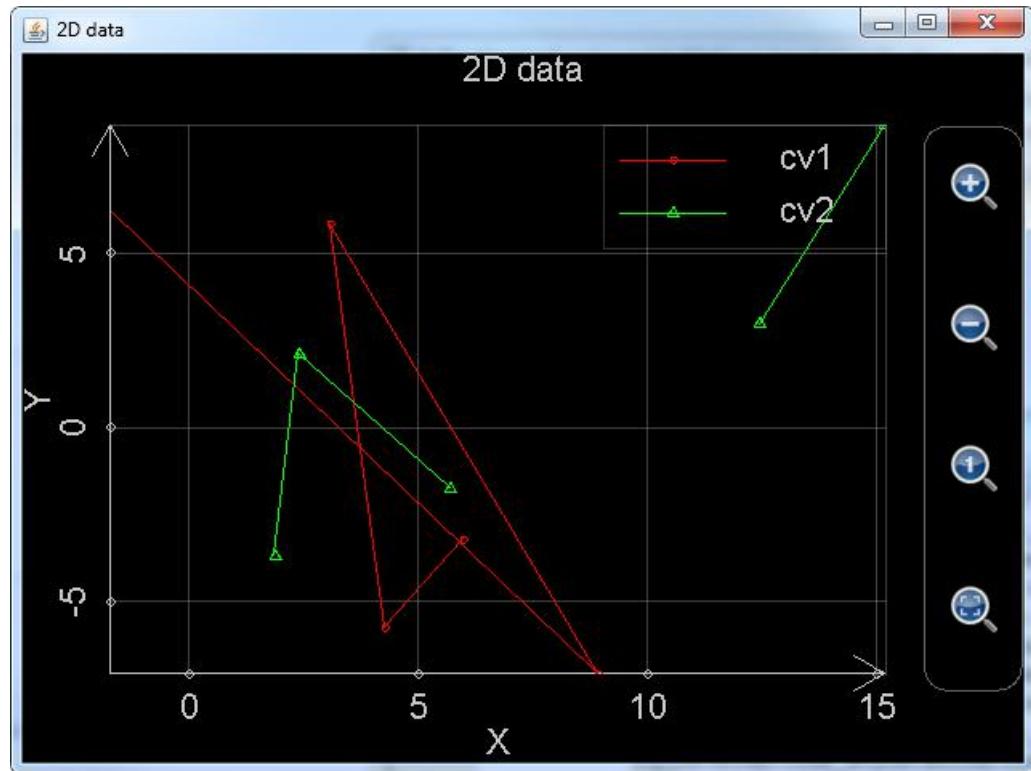


图 5.10：调用 `plot_2d_data`，使用 2 组参数同时绘制 2 个数据集，注意其中的绿色数据集的点之间的连接线有间断。

需要注意的是，用户虽然可以缩放拖动 `plot_2d_data` 所绘制出来的图像，但是无法设置每个数据集所使用的颜色以及点和线的形态，也无法保存所生成的图形。

`Plot_multi_xy` 则是非常底层的函数。这个函数和 `plot_2d_data` 一样是直接绘制数据，而不是根据表达式求得每个点的数据值然后加以绘制。`Plot_2d_data` 直接调用这个函数。而 `plot2dEx` 则是先将表达式的每一个数据点的值都计算出来，然后再调用 `plot_multi_xy`。

`Plot_multi_xy` 的第一个参数为基于字符串的图像的名字，也就是去掉扩展名.`.mfpc` 的图像文件名。

`Plot_multi_xy` 的第二个参数为图像的设置，这个参数是把所有的图像级别的设置放在一个字符串里面，每一个设置都基于

设置项目：设置项目的值；

的模式，然后将各个设置的首尾连接在一起。比如：

```
"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"
```

，这里的 chart_type 是图形类型，必须为 multiXY， chart_title 为图像标题， x_title 为图像 x 轴的名称， x_min 为最开始显示 x 轴的范围的最小值， x_max 为最开始显示 x 轴的范围的最大值， x_label 表示 x 轴上有多少个刻度标记， y_title 为图像 y 轴的名称， y_min 为最开始显示 y 轴的范围的最小值， y_max 为最开始显示 y 轴的范围的最大值， y_label 表示 y 轴上有多少个刻度标记， background_color 为背景色， show_grid 为是否显示网格。

Plot_multi_xy 的第 3 个参数为一条曲线的设置。这个参数是把所有的该曲线的设置放在一个字符串里面，每一个设置都基于

设置项目：设置项目的值；

的模式，然后将各个设置的首尾连接在一起。比如：

```
"curve_label:cv2;point_color:blue;point_style:circle;point_size:3;
line_color:blue;line_style:solid;line_size:1"
```

，这里 curve_label 是曲线标题， point_color 是曲线点的颜色， point_style 是曲线上点的形状（圆形，方形等）， point_size 是点的大小（和 plot_2d_curves 一样，这一项还没有实现，用户随便设置一个正整数即可）， line_color 是曲线的点的连接线的颜色， line_style 是连接线的类型（和 plot_2d_curves 一样，这一项还没有实现，用户设置为 solid 就好）， line_size 是线的宽度（必须为非负整数）。

Plot_multi_xy 的第 4 个参数为该曲线上每一个点在 x 轴上的坐标，注意这个参数必须为一个一维数组，数组中的每一个元素必须是实数，和 plot_2d_data 一样，如果该元素是 Nan，则曲线的连接线将在该点断开。

Plot_multi_xy 的第 5 个参数为该曲线上每一个点在 y 轴上的坐标，注意这个参数必须为一个一维数组，数组的长度和第 4 个参数必须一致，数组中的每一个元素必须是实数，和 plot_2d_data 一样，如果该元素是 Nan，则曲线的连接线将在该点断开。

如果用户想要绘制不止一条曲线，则需要输入另外一组参数 3, 4 和 5。用户最多可以绘制 1024 条曲线（对于 1.6.6 及以前版本，最多只能绘制 8 条曲线），所以 plot_multi_xy 最多支持 $2+1024*3$ 等于 3074 个参数。

Plot_multi_xy 的例子如下：

```
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:bu  
le;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])
```

上述例子绘制的图像为

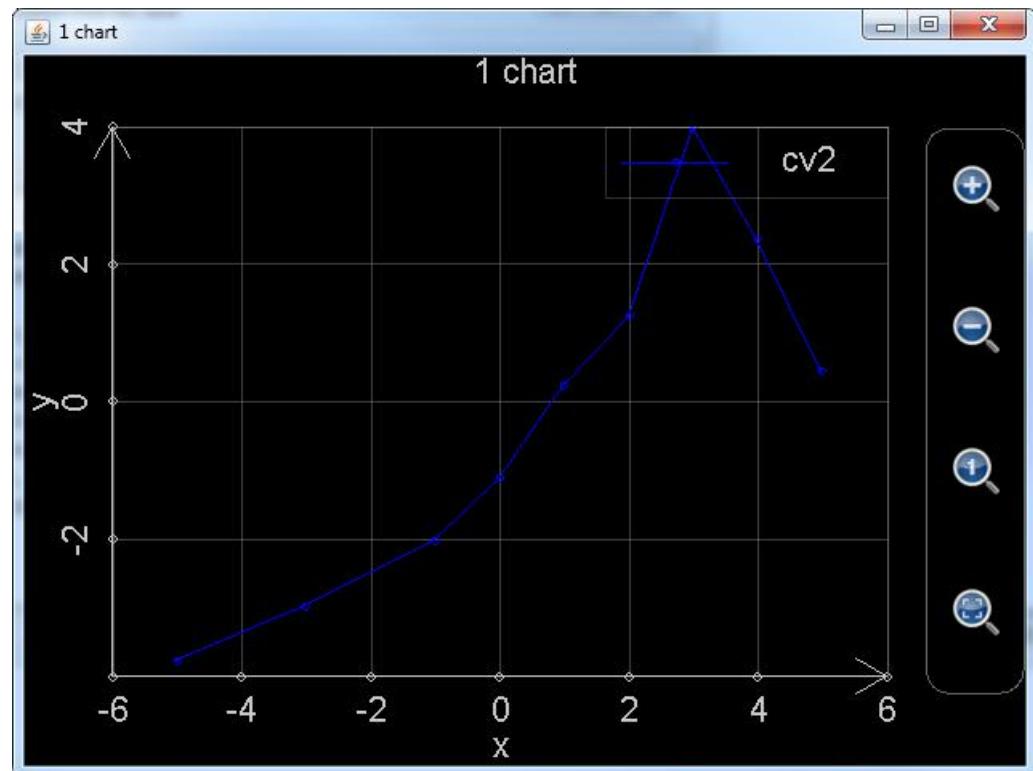


图 5.11：调用 plot_multi_xy 绘制图形。

注意，和 plot_2d_curves 一样，由于 plot_multi_xy 给出了图像文件的文件名，用户也可以在可编程科学计算器中再次打开它所生成的图像。

第 3 节 绘制极坐标系下的二维图像

绘制极坐标系下的二维图形，MFP 提供了和绘制普通坐标系下二维图形的类似的函数如下：

函数名	函数帮助信息
plot_polar	<p>plot_polar(6...) :</p> <p>函数 plot_polar 调用 plot_multi_xy 以绘制由最多 8 条极坐标曲线所构成的图像。其包括以下参数：1.图像名字（图像文件名）；2.图像标题；3.R 轴标题；4.幅角标题；5.图像背景色；6.是否显示网格；7.曲线标题；8.曲线数据点颜色；9.曲线数据点形状；10.曲线数据点大小；11.曲线连接线颜色；12.曲线连接线类型；13.曲线连接线粗细；14.t 起始位置；15.t 终止位置；16.t 的间隔；17.以 t 为变量的 R 的表达式；18.以 t 为变量的幅角的表达式...。其中，每增加一条新的曲线，需要添加 12 个参数（也就是参数 7 到 18），最多定义 8 条曲线。另外要注意图像背景色，曲线数据点的大小，曲线连接线颜色以及曲线连接线类型还没有被实现，曲线连接线粗细仅支持 0（意味着没有连接线连接数据点）和非 0（意味着有连接线连接数据点）。本函数的一个例子为：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")。</p>
plot_polar_curves	<p>plot_polar_curves(6...) :</p> <p>函数 plot_polar_curves 绘制由最多 1024 条极坐标曲线所构成的图像。其包括以下参数：1.图像名字（图像文件名）；2.图像标题；3.幅度轴标题；4.幅角标题（需要注意幅角标题在图中实际上不会被显示）；5.图像背景</p>

色；6. 是否显示网格（注意这个参数是一个字符串，其值为“true”或者“false”）；7. 曲线标题；8. 曲线数据点颜色；9. 曲线数据点形状；10. 曲线数据点大小；11. 曲线连接线颜色；12. 曲线连接线类型；13. 曲线连接线粗细；14. 内部变量的名字（通常为“t”）；15. 内部变量的起始位置；16. 内部变量的终止位置；17. 内部变量的每一步变化间隔；18. 基于内部变量的幅度的表达式；19. 基于内部变量的幅角的表达式...。其中，每增加一条新的曲线，需要添加13个参数（也就是参数7到19），最多定义1024条曲线。另外要注意图像背景色，曲线数据点的大小，曲线连接线颜色以及曲线连接线类型还没有被实现，曲线连接线粗细仅支持0（意味着没有连接线连接数据点）和非0（意味着有连接线连接数据点）。本函数的一个例子为：

```
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(16)：

函数 plot_polar_data 分析最少1组，最多8组数值向量，每组数值向量将会被绘制为一条极坐标曲线。输入的参数个数可以为2个（绘制一条曲线），4个（绘制2条曲线），6个（绘制3条曲线），8个（绘制4条曲线），10个（绘制5条曲线），12个（绘制6条曲线），14个（绘制7条曲线），16个（绘制8条曲线）。每一个参数都是一个数值向量（也就是一维矩阵）。奇数号参数决定曲线中的每一个点的 R 值，偶数号参数决定每一个点的幅角值。注意决定 R 值的参数中包含的元素个数应该和决定幅角值的参数中包含的元素相同。函数一个例子为 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])。

不难发现，plot_polar, plot_polar_curves 和 plot_polar_data 函数与前一节中详细介绍的 plot2dEx, plot_2d_curves 和 plot_2d_data 函数是一一对应的，它们甚至参数的输入都是几乎一模一样。并且，和前面用户 2D 绘图的版本一样，plot_polar_curves 由 JAVA 实现，最多能绘制 1024 条曲线，无论绘图速度还是曲线数量都远远强于 plot_polar 函数，所以将在未来的版本中逐步取代 plot_polar 函数。

Plot_polar_curves 和上一节中的 plot_2d_curves 的区别在于，第三个参数对于 plot_polar_curves 来讲是 R 轴（也就是极坐标中的幅角轴）的名称，而对于 plot_2d_curves 来讲是 x 轴的名称，第四个参数对于 plot_polar_curves 来讲是幅角的名称，对于 plot_2d_curves 来讲是 y 轴的名称。Plot_polar_curves 和 plot_2d_curves 一样，定义一条曲线也需要 13 个参数，不同的是，定义每条曲线的 13 个参数的倒数第二个参数对于 plot_polar_curves 函数来讲是幅值对内部变量（通常变量名称是 t）的函数，倒数第一个参数是幅角对内部变量的函数，而定义每条曲线的 13 个参数的倒数第二个参数对于 plot_2d_curves 函数来讲是 x 轴的坐标对内部变量的函数，倒数第一个参数是 y 轴坐标对内部变量的函数。

以下是使用 plot_polar_curves 函数绘制莲花和蝴蝶形状的例子。莲花形状的方程式为

$$r = \sin \theta + \sin^3(2.5\theta) \dots \dots \dots 0 \leq \theta \leq 4\pi$$

，那么，如果将 θ （也就是幅角）对 t 的表达式设置为 t， r （也就是幅值）对 t 的表达式则是 $r(t) == \sin(t) + \sin(2.5*t)**3$ ，t 的变化范围是 0 到 $4*\pi$ ，我们可以设定步长为 0.05。

蝴蝶形状的极坐标方程式为

$$r = 0.6e^{\sin \theta} - 2 \cos 4\theta + \sin^5 \frac{2\theta - \pi}{24}$$

，那么，如果将 θ （也就是幅角）对 t 的表达式设置为 t， r （也就是幅值）对 t 的表达式则是 $r(t) == 0.6 * \exp(\sin(t)) - 2 * \cos(4*t) + \sin((2*t - \pi)/24)**5$ ，t 的变化范围是 $-\pi$ 到 π ，我们可以设定步长为 0.02。

整个绘图函数的调用语句为：

```
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")
```

绘制出来的图像如下：

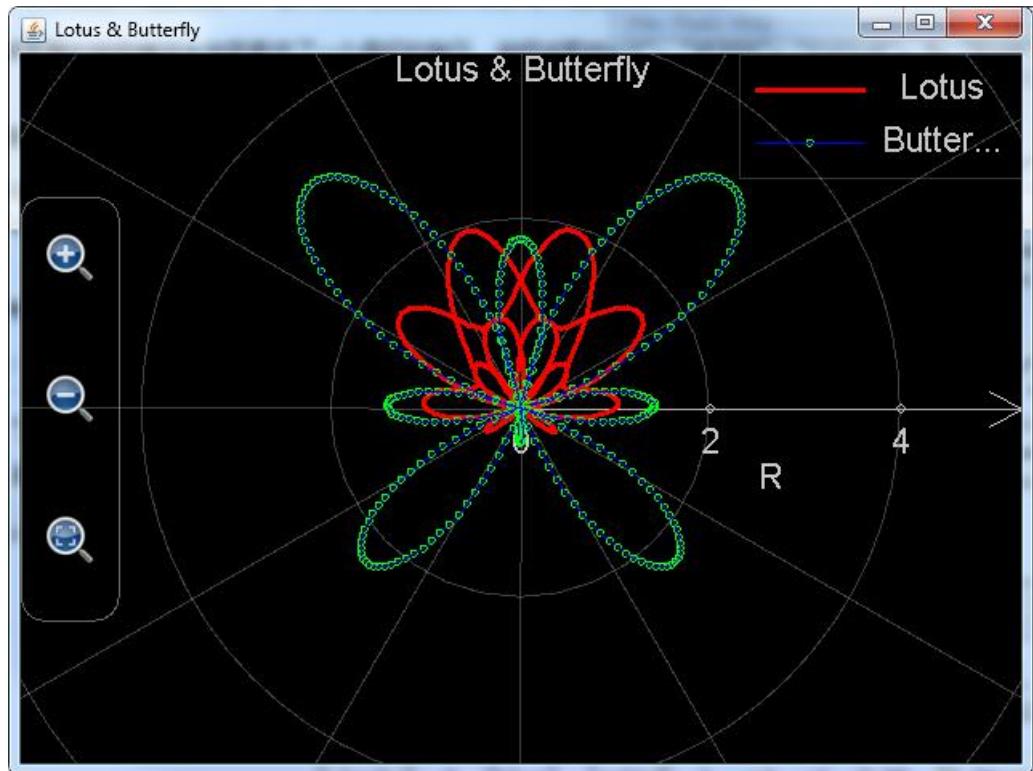


图 5.12：调用 `plot_polar_curves` 绘制莲花和蝴蝶。

和 `plot_2d_curves` 函数一样，上述绘制出来的图案被保存为文件（文件名为 `plot_polar_curves` 的第一个参数加上 `.mfpc` 扩展名，也就是 `LotusAndButterfly.mfpc`）存放 `AnMath` 目录下的 `charts` 子目录中。用户可以在可编程科学计算器中打开。

`Plot_polar_data` 和上一节中的 `plot_2d_data` 的参数输入的区别在于，`plot_polar_data` 不能够仅仅只输入一个一维数组作为参数。`Plot_polar_data` 输入的参数必须是成对的，每对参数对定义一组数据点，每对参数包括两个一维数组，，数组中的元素必须为实数或者 `Nan`，每对参数中两个数组的长度必须相符。其中，奇数号数组定义了该组数据点的幅值，偶数号数组定义了数据点的幅角。

下述例子用于在极坐标系中绘制两组数据，第一组数据中含有 4 个点，第二组中有 6 个点。但要注意第二组数据中有一个点的幅值为 Nan，所以，第二组数据各点之间的连线在这一点断开。

```
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])
```

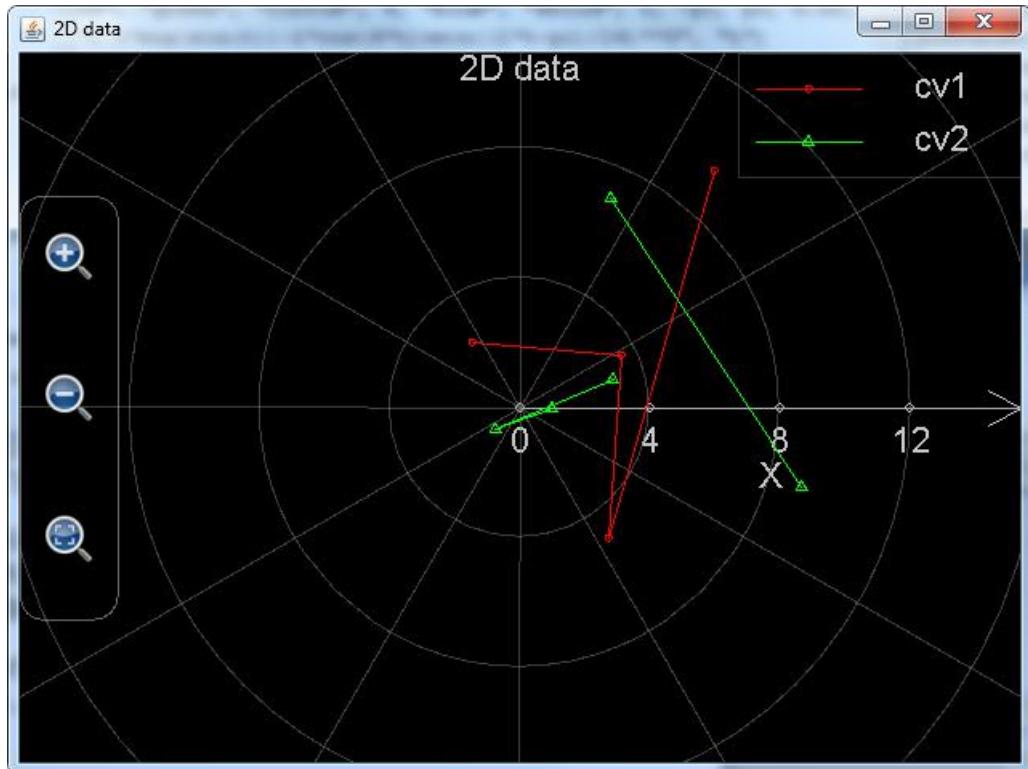


图 5.13：调用 plot_polar_data 在极坐标系中绘制数据图。

Plot_polar 和 plot_polar_data 实际上都是使用 MFP 语言本身编写的绘图函数，它们进行绘图时，都调用了上一节中介绍的 plot_multi_xy 函数。在上一节的函数表中，对 plot_multi_xy 的介绍已经很清楚了：plot_multi_xy(包含至少 2 个参数)用于绘制 2 维或极坐标图像，每个图像最多包括 1024 条曲线。参数 1 为图像名字，参数 2 为图像设置，该参数是一个字符串，比如 "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: 4.126871578884345; y_labels: 10; background_color: black; show_grid: true"。注意这里 chart_type 的值不再是 multiXY (用于绘制二维图像) 而是 multiRangle (用于绘制极坐标图像) 了。x_labels 对于极坐标

图像来讲是 R 轴有多少刻度标记，y_labels 对于极坐标幅值的设定毫无影响，因为幅角的刻度标记的个数永远是 8 个，不可以更改。所以用户只用给 y_labels 随便填入一个正整数即可。

那么，如果把上一节中在常规坐标系下调用 plot_multi_xy 绘制的图形搬到极坐标系下会是怎样的呢，运行下述语句：

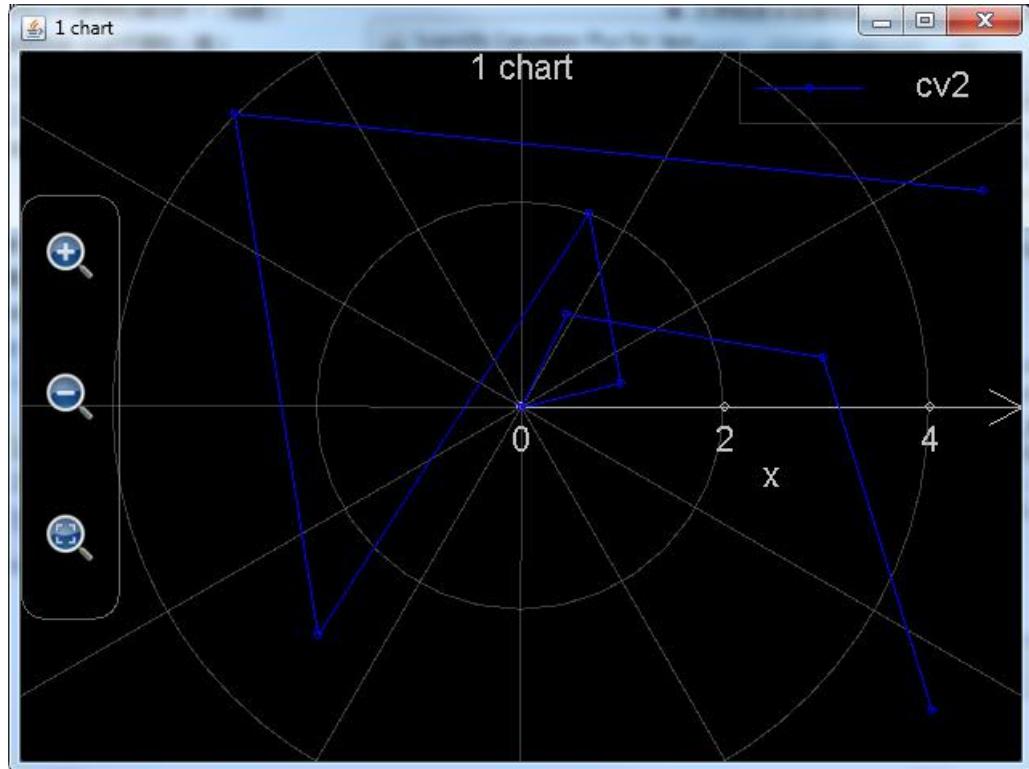


图 5.14：调用 plot_multi_xy 在极坐标系中绘制数据图形。

```
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:bu  
le;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])
```

，可以看到绘制出的图形为参见上图。

上述例子在极坐标系上的图案似乎杂乱无章，对比图 5.11：，不难得出结论，极坐标系和常规坐标系是展现数据的两种不同方式，有时候在一个坐

标系下找不到规律的数据，放到另外一个坐标系下，就能得到很好的分析效果。

第4节 绘制三维图像

MFP 提供了以下函数用于绘制三维图形：

函数名	函数帮助信息
plot3d	<p>plot3d(5...) :</p> <p>函数 plot3D 调用 plot_multi_xyz 以绘制由最多 8 条 3-D 曲面所构成的图像。其包括以下参数：1.图像名字（图像文件名）；2.图像标题；3.X 轴标题；4.Y 轴标题；5.Z 轴标题；6.曲线标题；7.是网格还是填充表面（true 是网格， false 是填充表面）；8.z 值最小的时候的颜色；9.最小的 z 值（注意如果是 null， 意味着让软件自己找到最小的 z 值）；10.z 值最大的时候的颜色；11.最大的 z 值（注意如果是 null， 意味着让软件自己找到最大的 z 值）；12.变量 u 的起始值；13.变量 u 的终止值；14.变量 u 的间隔（0 意味着间隔由软件决定）；15.变量 v 的起始值；16.变量 v 的终止值；17.变量 v 的间隔（0 意味着间隔由软件决定）；18.以 u, v 为变量的 X 的表达式；19.以 u, v 为变量的 Y 的表达式；20.以 u, v 为变量的 Z 的表达式；...。其中，每增加一条新的曲线，需要添加 15 个参数（也就是参数 6 到 20），最多定义 8 条曲线。该函数的一个例子为：</p> <pre>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)") 。</pre>
plot_3d_surfaces	<p>plot_3d_surfaces(5...) :</p> <p>函数 plot_3d_surfaces 绘制由最多 1024 条 3-D 曲面所构成的图像。其包括以下参数：1.图像名字（图像文件名）；2.图像标题；3.X 轴标题；4.Y 轴标题；5.Z 轴标题；6.曲线标题；7.是网格还是填充表面（这是一个布</p>

	<p>尔值, true 是网格, false 是填充表面) ; 8.z 值最小的时候的正面的颜色; 9.z 值最小的时候的反面的颜色; 10.最小的 z 值 (注意如果是 null, 意味着让软件自己找到最小的 z 值) ; 11.z 值最大的时候的正面的颜色; 12.z 值最大的时候的反面的颜色; 13.最大的 z 值 (注意如果是 null, 意味着让软件自己找到最大的 z 值) ; 14.第一个内部变量的名字 (通常为"u") ; 15.第一个内部变量的起始值; 16.第一个内部变量的终止值; 17.第一个内部变量的每一步变化的间隔 (0 意味着间隔由软件决定) ; 18.第二个内部变量的名字 (通常为"v") ; 19.第二个内部变量的起始值; 20.第二个内部变量的终止值; 21.第二个内部变量的每一步变化的间隔 (0 意味着间隔由软件决定) ; 22.基于前述两个内部变量的 X 的表达式; 23.基于前述两个内部变量的 Y 的表达式; 24.基于前述两个内部变量的 Z 的表达式; ...。其中, 每增加一条新的曲面, 需要添加 19 个参数 (也就是参数 6 到 24), 最多定义 1024 条曲面。该函数的一个例子为: plot_3D_surfaces("chartI", "first chart", "x", "y", "z", "Curve1", false, "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)")。</p>
plot_3d_data	<p>plot_3d_data(24):</p> <p>函数 plot_3d_data 分析最少 1 组, 最多 8 组数值矩阵, 每组数值矩阵将会被绘制为一条曲面。输入的参数个数可以为 1 个 (绘制一条曲面), 3 个 (绘制一条曲面), 6 个 (绘制 2 条曲面), 9 个 (绘制 3 条曲面), 12 个 (绘制 4 条曲面), 15 个 (绘制 5 条曲面), 18 个 (绘制 6 条曲面), 21 个 (绘制 7 条曲面), 24 个 (绘制 8 条曲面)。如果只有一个参数, 该参数必须是一个 2 维矩阵, 矩阵中的每一个元素将会是绘制出的曲线中的一个点的 z 值, 否则, 每 3 个参数组成一个参数组, 在每个参数组中, 第一个参数必须是一个一维矩阵, 参数中的元素值决定的决定曲面中各点的 x 值, 第二个参数也必须是一个一维矩阵, 参数中的元</p>

	<p>素值决定的决定曲面中各点的 y 值，第三个参数必须是一个二维矩阵，矩阵中的元素值决定每一个点的 z 值。注意决定 x 值的参数中包含的元素个数以及决定 y 值的参数中包含的元素应该和决定 z 值的参数中包含的元素个数相符。函数例子包括 <code>plot_3d_data([[2.47, 3.53, 4.88, 9.42], [8.49, 6.76, 5.31, 0.88], [-9, -7, -5, -3, -1]])</code> 以及 <code>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]])</code>。</p>
<code>plot_multi_xyz</code>	<p><code>plot_multi_xyz(2...)</code> :</p> <p><code>plot_multi_xyz</code>(包含至少 2 个参数)用于绘制 3 维图像，每个图像最多包括 1024 条曲面。参数 1 为图像名字，参数 2 为图像设置，图像设置参数是一个字符串，比如"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"。需要注意的是 <code>chart_type</code> 的值必须是 <code>multiXYZ</code>，<code>x_labels</code>，<code>y_labels</code> 和 <code>z_labels</code> 分别代表 x，y 和 z 轴上有多少刻度标记。从参数 3 到参数 34，每 4 个参数定义一条曲线，在这 4 个参数中，第一个参数为曲线设置，第二个参数是包含所有 x 数值的矩阵，第三个参数是包括所有 y 值的矩阵，第四个参数是包括所有 z 值的矩阵。曲线设置参数为一个字符串，比如"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"。另外注意 x，y 和 z 的每一个数值都必须为实数，x，y 和 z 的矩阵的尺寸必须一致。本函数不返回数值。本函数的一个例子为： <code>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:-5;y_max:6;y_labels:6;z_title:Z;z_min:-1;z_max:1;z_labels:6")</code>。</p>

```

6;y_max:6;y_labels:3;z_title:Z;z_min:-
3;z_max:1;z_labels:4",
"curve_label:c1;min_color:blue;min_color_1:green;max_c
olor: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]])。

```

其中，plot3d 和 plot_3d_surfaces 是用来在指定范围内绘制 3 维表达式曲面或者曲线，plot_3d_data 用于绘制三维数据图像。Plot_multi_xyz 为更底层的函数，它被 plot3d 和 plot_3d_data 函数所调用。和 plot2dEx 以及 plot_polar 函数一样，plot3d 函数由 MFP 语言写成，最多只能在一张图上绘制 8 条曲面，它的 JAVA 实现的对应版本 plot_3d_surfaces 可以最多绘制 1024 条曲面（在 1.6.6 及其以前的版本中 plot_3d_surfaces 函数接口不对用户开放，并且最多也只能绘制 8 条曲面），计算速度也快很多，强烈建议用户逐步放弃使用 plot3d 函数而转移到 plot_3d_surfaces 函数上。

Plot_3d_surfaces 函数事实上就是安卓上的可编程科学计算器的独立的“绘制图形”→“绘制三维图像”工具的函数版。它所需要的参数中，前 5 个参数用于设置图像本身，分别为图像的文件名（文件扩展名.mfpc 会被自动加上，不用用户输入），图像的标题，x 轴的名字，y 轴名字和 z 轴名字。这些参数都是字符串。图像的背景色永远是黑色，不能够由用户设定。从第 6 个参数开始，每 19 个参数用于设置一条曲线（或者曲面）。它们分别是：

1. 曲线（面）的名字（基于字符串）；
2. 曲面是网格还是填充表面（注意该参数不是字符串，而是布尔值。
true 是网格，false 是填充表面。如果绘制的曲面，这两种都可以。如果绘制的是曲线，强烈建议设置为 true，也就是网格，否则，曲线和坐标轴的颜色一样，都是灰色的，看都看不清）；
3. Z 坐标方向上最小值对应的曲面的正面颜色（基于字符串，包括 "white"（白色），"black"（黑色），"red"（红色），"green"（绿色），"blue"（蓝色），"yellow"（黄色），"cyan"（青色），"magenta"（紫红色），"dkgray"（深灰色）以及"ltgray"（浅灰色），如果字符串不是上述任何一种，将使用白色）；

4. Z 坐标方向上最小值对应的曲面的反面颜色（基于字符串，包括 "white"（白色），"black"（黑色），"red"（红色），"green"（绿色），"blue"（蓝色），"yellow"（黄色），"cyan"（青色），"magenta"（紫红色），"dkgray"（深灰色）以及"ltgray"（浅灰色），如果字符串不是上述任何一种，将使用白色）；
5. 最小的 z 值（注意，这个最小的 z 值不见得是曲面在 Z 坐标方向上的最小值，它只是定义了颜色的变化，也就是，图案中任何小于最小 z 值的部分都被涂以最小 z 值对应的颜色，图案中大于最小 z 值的部分的颜色从最小 z 值对应的颜色向最大 z 值所对应的颜色渐变。如果该值设置为 null，则让软件自己寻找曲面或曲线的最小 z 值）；
6. Z 坐标方向上最大值对应的曲面正面的颜色（基于字符串，包括 "white"（白色），"black"（黑色），"red"（红色），"green"（绿色），"blue"（蓝色），"yellow"（黄色），"cyan"（青色），"magenta"（紫红色），"dkgray"（深灰色）以及"ltgray"（浅灰色），如果字符串不是上述任何一种，将使用白色）；
7. Z 坐标方向上最大值对应的曲面反面的颜色（基于字符串，包括 "white"（白色），"black"（黑色），"red"（红色），"green"（绿色），"blue"（蓝色），"yellow"（黄色），"cyan"（青色），"magenta"（紫红色），"dkgray"（深灰色）以及"ltgray"（浅灰色），如果字符串不是上述任何一种，将使用白色）；
8. 最大的 z 值（注意，这个最大的 z 值不见得是曲面在 Z 坐标方向上的最大值，它只是定义了颜色的变化，也就是，图案中任何大于最大 z 值的部分都被涂以最大 z 值对应的颜色，图案中小于最大 z 值的部分的颜色从最大 z 值对应的颜色向最小 z 值所对应的颜色渐变。如果该值设置为 null，则让软件自己寻找曲面或曲线的最大 z 值）；
9. 第一个内部变量的名字，通常是 u；
10. 第一个内部变量，也就是变量 u（如果被命名为 u 的话）的变化范围的起始值（必须是一个实数。这里的 u 和前面用基于安卓的可编程科学计算器上独立的“绘制图形”工具绘制三维图像时所

使用的 u 是一样的。关于 u 的详细说明，参见第 1 章第 4 节中对于 u 的解释和定义)；

11. 变量 u 的变化范围的终止值（必须是一个实数）；
12. 变量 u 的变化的步长（必须是一个实数，u 的变化范围除以变化步长加 1 就是在 u 方向上要绘制的点的个数。注意用户可以将其设置为 0，设置为 0 意味着变化的步长由软件决定）；
13. 第二个内部变量的名字，通常为 v；
14. 第二个内部变量，也就是变量 v（如果被命名为 v 的话）的变化范围的起始值（必须是一个实数。这里的 v 和前面用基于安卓的可编程科学计算器上独立的“绘制图形”工具绘制三维图像时所使用的 v 是一样的。关于 v 的详细说明，参见第 1 章第 4 节中对于 v 的解释和定义）；
15. 变量 v 的变化范围的终止值（必须是一个实数）；
16. 变量 v 的变化的步长（必须是一个实数，v 的变化范围除以变化步长加 1 就是在 v 方向上要绘制的点的个数。注意用户可以将其设置为 0，设置为 0 意味着变化的步长由软件决定）；
17. X 坐标变化对于 u 和 v 的函数（基于字符串。关于 $X(u, v)$ 的详细说明，参见第 1 章第 4 节中的解释）；
18. Y 坐标变化对于 u 和 v 的函数（基于字符串。关于 $Y(u, v)$ 的详细说明，参见第 1 章第 4 节中的解释）；
19. Z 坐标变化对于 u 和 v 的函数（基于字符串。关于 $Z(u, v)$ 的详细说明，参见第 1 章第 4 节中的解释）；

由于 `plot_3d_surfaces` 函数最多可以绘制 1024 条曲线，它的参数个数最多可以达到 $5+19*1024$ 等于 19461 个。

下面给出一个例子用于绘制彩色 6 面立方体。绘制立方体是所有三维绘图软件必包含的例子。MFP 一样也可以做到。绘制立方体的思路是，立方体的六个面每个面是 `plot_3d_surfaces` 函数的一个待绘制曲面，假设立方体的

边长为 2，那么可以定义 u 从 -1 到 1 步长为 2（步长为 2 意味着曲面上不会有网格线出现，因为曲面本身就是网格中的一个格），v 从 -1 到 1 步长也为 2，由于立方体的任意一个面必然与某一个坐标轴在 -1 或者 1 垂直相交而平行于另外两个坐标轴，所以 x, y 和 z 中，必然有一个等于 -1 或者 1，另外两个一个等于 u，另一个等于 v。整个调用语句如下：

```
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")
```

绘制出的图形效果如下：

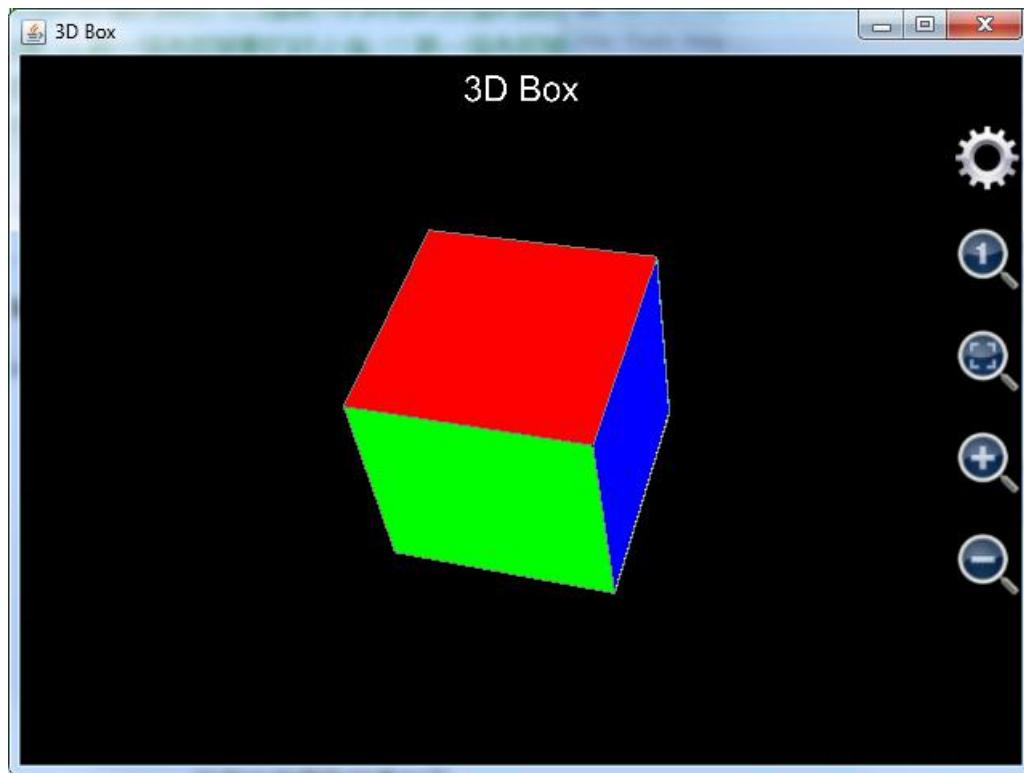


图 5.15：调用 plot_3d_surfaces 函数绘制立方体。

注意到在上述例子中由于我们设置曲面的标题为空字符串，各个曲面的图例说明不会在图中显示。

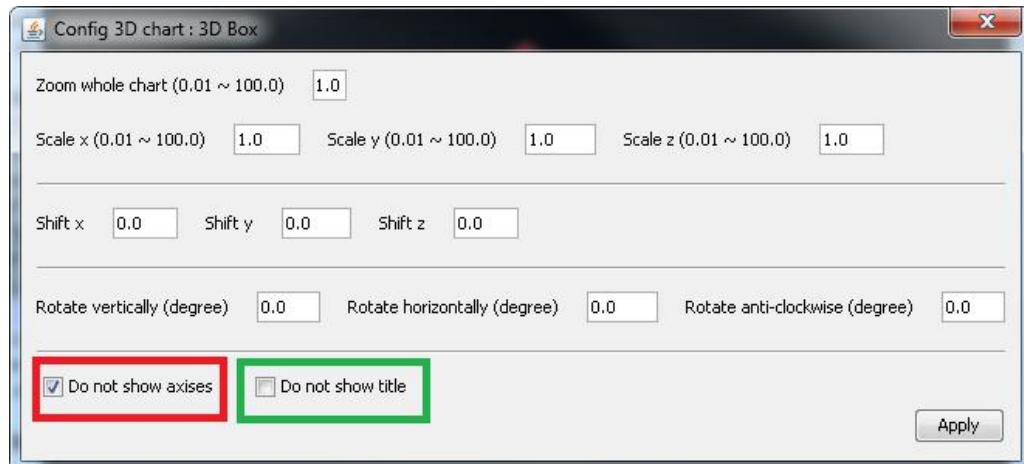


图 5.16：设置三维图形，隐藏坐标轴和标题。

用户可以用手指或者鼠标拖动图形进行旋转，还可以点击放大缩小图标调整图形大小（在安卓上，用户还可以用捏合扩张的手势进行同样的操作），如果我们想要显示坐标轴或者觉得标题太碍眼想隐藏标题，则可以点击齿轮按钮，不选中隐藏坐标轴（红色的方框），但选择隐藏标题（绿色方框），参见上图。

需要注意的是，在可编程科学计算器 1.6.6 版中，用户只能选择同时隐藏坐标轴和标题或者同时显示坐标轴和标题。在更老的版本中，用户是无法隐藏坐标轴和标题的。

用户不选中红色长方形中的选择框，但是选中绿色长方形中的选择框，则坐标轴显示但标题被隐藏。图像效果参见下图：

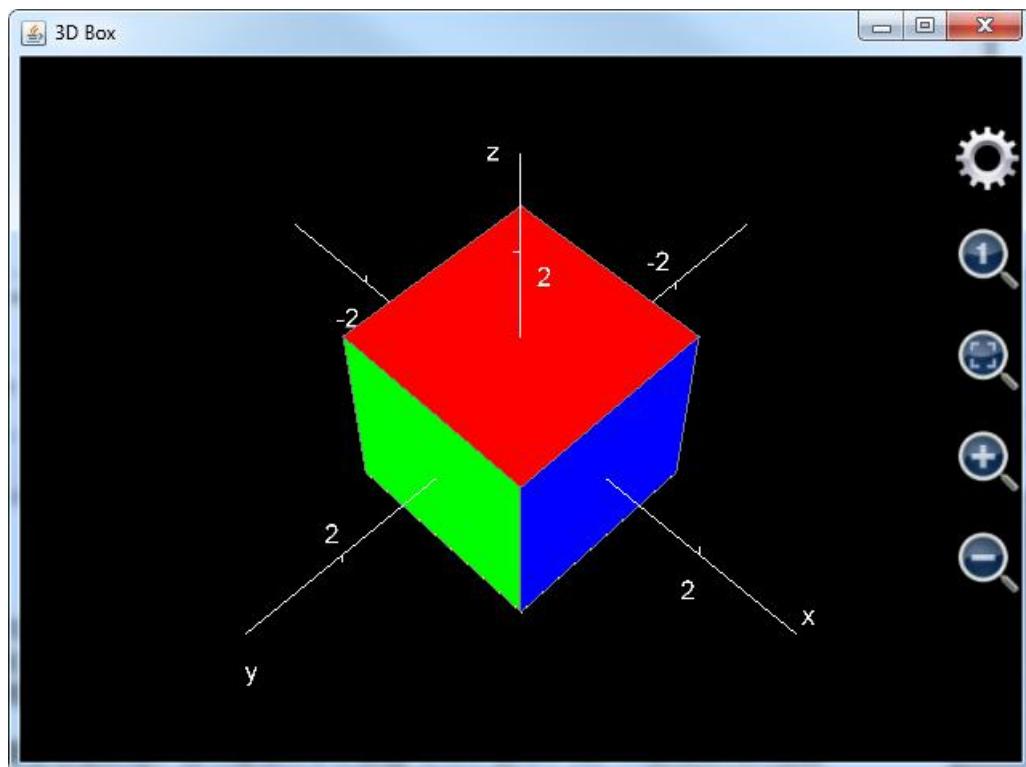


图 5.17：隐藏坐标轴和标题后的三维立方体。

和 `plot_2d_curves` 函数一样，由于用户在调用函数时指定了文件名，生成的图像会被自动保存在 `AnMath/charts` 目录中方便日后用户在可编程科学计算器中打开查看。

绘制立方体，只能算是 MFP 强大的三维绘图功能的开胃甜点，以下例子向用户展示如何用 `plot_3d_surfaces` 函数绘制北京的地标建筑鸟巢。鸟巢的照片如下：



图 5.18：北京的地标建筑鸟巢的实物照片。

绘制鸟巢，并非是把鸟巢上的每一根钢筋支架都画出来。一些细节的东西在图中将会被省略，绘制的关键在于鸟巢顶部圆环波浪形半开状态的屋顶，和鸟巢四周圆环形，但又是倾斜的围墙。

我们知道，如果用 u 和 v 作变量， u 看作是幅角， v 看作是半径，圆形的表达式为

$$x = v \cos u$$

$$y = v \sin u$$

这里， u 的变化范围是从 $-\pi$ 到 π ， v 是一个常数。

如果绘制的是圆环，相当于由很多同心圆层层叠加，这时， v 就不再是一个常数，而是一个范围，我们假设鸟巢屋顶的外径为 15，内径为 7.5，为了绘制鸟巢屋顶， v 的范围就是从 7.5 到 15。

再假设鸟巢的平均高度是 5，所以，鸟巢屋顶所在的曲面的 z 坐标平均值为 5。

问题在于，鸟巢的屋顶并不是平的，它是一个波浪形，也就是一个余弦曲线，从鸟巢的实物照片中可以看到，鸟巢屋顶的变化是从高到低再到高再到低，就相当于从 $-\pi$ 到 π 的范围内，该余弦曲线震荡了两次，所以，该余弦曲线应该用 $\cos(2*u)$ 来表示，所以鸟巢屋顶的实际大致（注意是大

致)高度应该是 $5*(1 + 0.3 * \cos(2*u))$, 这里的 0.3 表示余弦曲线的震荡幅度, 这个值是通过看图估计出来的。

还注意到, 鸟巢屋顶高度波浪形的变化并非是从里到外均匀一致的, 越靠近边缘, 屋顶高度的波浪形变化越明显, 所以, 鸟巢屋顶高度的震荡部分还要乘以一个和半径相关的系数, 所以, 鸟巢屋顶高度, 也就是鸟巢屋顶的 z 坐标的最终表达式为 $5*(1 + 0.3 * \cos(2*u)*v/15)$, 这里 5 为鸟巢屋顶的平均高度, 0.3 为屋顶高度的震荡幅度, 15 为鸟巢的最大半径。

本来, 如果鸟巢是一个标准的圆柱体, 鸟巢屋顶的高度变化对屋顶上每个点的 x 坐标和 y 坐标不会有影响。但是问题在于, 鸟巢并不是标准的圆柱体。由于鸟巢的外墙是从上往下向内倾斜, 所以, 鸟巢实际上是一个倒置的圆椎体的一部分。这样一来, 屋顶的高度发生变化, 会引起屋顶边缘所在点的半径发生畸变, 屋顶的高度变高, 屋顶边缘的半径就会变长, 屋顶的高度变低, 边沿的半径就会变短。我们假设鸟巢外墙的倾斜度(斜率)为 5, 由于已经假定鸟巢顶部的直径为 15, 高度为 5, 所以鸟巢底部的直径应该是 $15 - 5/5 = 14$ 。换句话说, 鸟巢所在椎体的上下半径的差距为 1, 是最大半径的 $1/15$ 。这就意味着, 鸟巢边缘半径随高度的变化而变化, 变化幅度为高度变化幅度的 $1/15$ 。由于鸟巢屋顶的 z 坐标的最终表达式为 $5*(1 + 0.3 * \cos(2*u)*v/15)$, 所以鸟巢屋顶的 x 坐标的最终表达式为 $v*(1 + 0.02 * \cos(2*u)*v/15)*\cos(u)$, 而 y 坐标的最终表达式为 $v*(1 + 0.02 * \cos(2*u)*v/15)*\sin(u)$ 。这里, 0.02 就是半径变化的幅度, 它是 0.3 的 $1/15$ 。

下一步就是绘制鸟巢四周的围墙, 上面已经假设鸟巢的顶部最大半径为 15, 底部最大半径为 14, 高度为 5。所, 可以逆推得到, 鸟巢所在的圆椎体的顶点所在位置为 $z = -14 / (15 - 14) * 5 = -70$, 圆锥体表面的斜率为 5。如果只是想绘制这个圆锥体从半径等于 14 到半径等于 15 的带状部分, 可以将变量 v 看作高度, 变化范围从 0 到 5, u 看作幅角, 变化范围从 $-\pi$ 到 π , 则 x、y 和 z 的表达式为

$$x = (14 + \frac{v}{5} \cos u)$$

$$y = (14 + \frac{v}{5} \sin u)$$

$z = v$

但是，由于屋顶的波浪形变化，造成了鸟巢外墙的顶部高度发生畸变，这种畸变从顶部向底部传递的过程中逐渐减弱，外墙底部是完全平坦的。由于 v 的顶部是 5，底部是 0，我们可以简单地给 z 乘以屋顶边缘高度的震荡系数，也就是 $z=v*(1+0.3*\cos(2*u))$ ，这里的 0.3 就是上面提到的屋顶高度的震荡幅度。

由于屋顶高度发生变化，外墙上每个点的 x 和 y 的坐标也会有相应的畸变，由于鸟巢所在圆锥体的表面斜率为 5，所以 x 和 y 坐标的畸变幅度应该是 z 方向畸变幅度的 $1/5$ 。这样一来，鸟巢外墙的 x 坐标的表达式为

$$x=(14 + v * (1 + 0.3 * \cos(2*u))/5) * \cos(u)$$

$$y=(14 + v * (1 + 0.3 * \cos(2*u))/5) * \sin(u)$$

鸟巢的底部就是一个简单的圆形，半径为 14，高度为 0， u 看作是幅角，变化范围是从 $-\pi$ 到 π ， v 看作是半径，变化范围从 0 到 14，那么鸟巢底部 x ， y 和 z 的表达式为

$$x = (14 \cos u)$$

$$y = (14 \sin u)$$

$$z = 0$$

所以，最终绘制鸟巢图案的语句为：

```
Plot_3d_surfaces("birdnest", "Bird's Nest", "x", "y", "z", _           //设定图像的名字
  "", false, "red", "blue", null, "cyan", "ltgray", null, "u", -pi, pi, 0, "v",
  0, 5, 0, "(14 + v * (1 + 0.3 * \cos(2*u))/5) * \cos(u)", "(14 + v * (1 + 0.3 *
  \cos(2*u))/5) * \sin(u)", "v * (1 + 0.3 * \cos(2*u))", _ // 绘制鸟巢的四周围墙

  "", false, "magenta", "red", null, "yellow", "green", null, "u", -pi, pi, 0,
  "v", 7.5, 15, 0, "v*(1 + 0.02 * \cos(2*u)*v/15)*\cos(u)", "v*(1 + 0.02 *
  \cos(2*u)*v/15)*\sin(u)", "5*(1 + 0.3 * \cos(2*u)*v/15)", _ // 绘制鸟巢的顶部

  "", false, "green", "cyan", null, "green", "cyan", null, "u", -pi, pi, 0,
  "v", 0, 14, 0, "v*\cos(u)", "v*\sin(u)", "0") // 绘制鸟巢的底部
```

图像绘制结果参见下图。注意图像刚开始显现的时候，图形的尺寸比较小，用户可以点击绿色方框中的放大按钮，将图形放大，并用鼠标拖动图形进行旋转，调整观察角度，以获取最好的视觉效果。

还要注意，如果用户使用的是 1.6.6 版本的可编程科学计算器，图像刚开始显现的时候，看起来并不像鸟巢，原因在于图形太大，并且 x、y 和 z 轴的单位长度不成比例，用户可以点击红色方框中的放大镜中有一个小 1 的按钮，将 x、y 和 z 轴比例调整为 1: 1: 1，然后再点击黄色方框中的缩小按钮，便可以看到逼真的鸟巢图形。

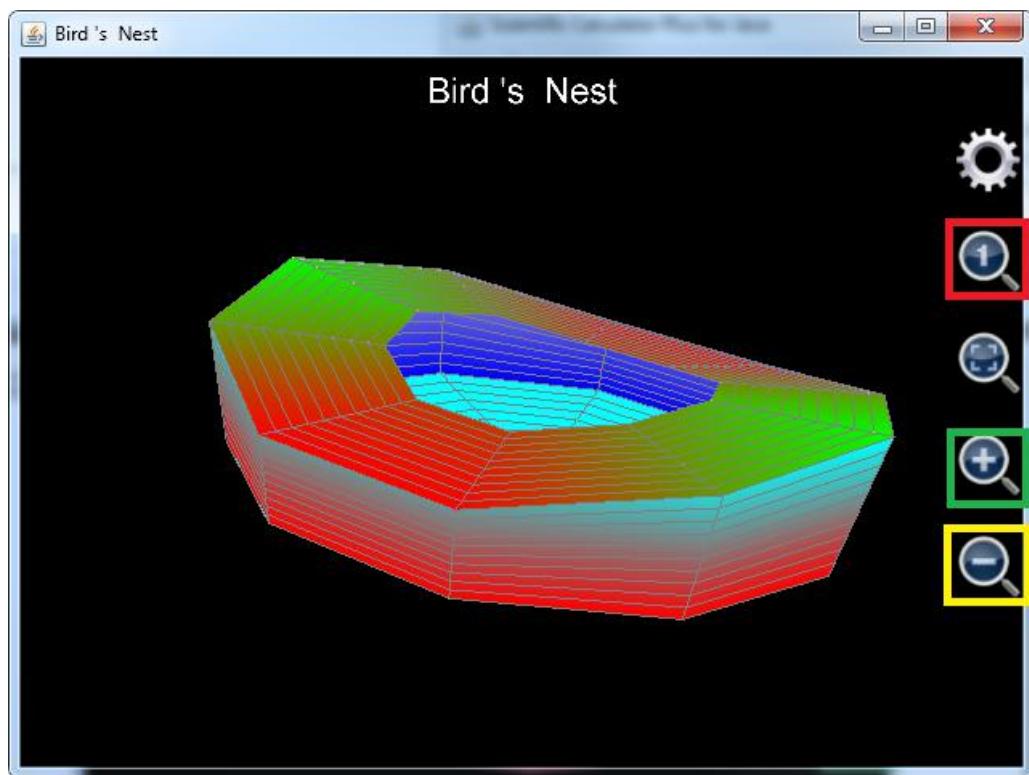


图 5.19：鸟巢三维图像。

如果用户使用的是 1.6.6 及其以前版本的可编程科学计算器，在调用 `plot3d` 函数绘制包含多个曲目的 3 维图形时，往往会被 `plot3d` 函数最多支持 8 组 x, y 和 z 的表达式的限制所困扰，但是，完全可以用 `plot3d` 函数绘制出复杂的，包括多于 8 个曲面的立体图像。其办法是，调用 `iff` 函数，在 u (或者 v) 处于某一个范围时绘制出一个曲面，在 u (或者 v) 处于另外一个范围时绘制出另一个曲面。为了避免两个曲面相连，u (或者 v) 处于这两个范围之间时，x 或者 y 或者 z 设置为 `Nan`。这种办法也可以

用于 `plot_3d_surfaces`、`plot2dEx`、`plot_2d_curves`、`plot_polar` 和 `plot_polar_curves` 函数，当然由于从 1.6.7 版开始，`plot_3d_surfaces`、`plot_2d_curves` 和 `plot_polar_curves` 函数一次均可绘制出 1024 条曲线，使用这个办法对它们来说没有意义。

在第 1 章第 4 节中展示了如何使用“绘制三维图形”工具绘制上海东方明珠电视塔。为了方便 1.6.6 及其以前版本的可编程科学计算器用户，在这里给出直接使用 `plot3d` 函数绘制工具绘制上海东方明珠电视塔的代码和在个人电脑上的绘图效果（在安卓平台上也可以同样的函数绘制出同样的效果图），可以看到，使用 MFP 编程绘制复杂图形，比在手机上输入方便得多。另外需要注意的是 `plot3d` 函数是如何使用一组表达式绘制多个曲面的，比如，在绘制 3 根大小球之间的连接柱体时，`plot3d` 函数设置 `u` 的变化范围从 0 到 8（也就是幅角从 0 到 8π 变化）；`v` 的变化范围从 20 到 70（也就是立柱的高度为 20 到 70）；`x` 的表达式为 `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)`，这里，`u` 从 0 到 2（也就是幅角从 0 到 2π ）用于绘制第一个柱体，`u` 从 3 到 5（也就是幅角从 3π 到 5π ）用于绘制第二个柱体），`u` 从 6 到 8（也就是幅角从 6π 到 8π ）用于绘制第三个柱体，`iff` 函数给出了在这三个范围内 `x` 的不同的表达式。

需要注意，在这三个范围之间，也就是 `u` 从 2 到 3 和从 5 到 6，`x` 的表达式值为 `Nan`，其原因，如上所述，是为了保证这三根柱体的表面不是相互连接的。

同理，`y` 的表达式为 `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)`，而 `z` 由于仅仅表示高度，表达式相对简单，就是 `v`。

绘制上海东方明珠电视塔的代码如下：

```
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 (绘制底部三根支撑斜柱体)
```

```
 "", false, "green", null, "yellow", null, -  
 1, 1, 0.25, 0, 20, 20, "cos(u*pi)*2", "sin(u*pi)*2", "v", _ //plot connection column (绘制连接柱)
```

```
 "", 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 (绘制大球)
```

```
 "", 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 (绘制大球和小球之间的三根连接柱)
```

```
 "", 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 (绘制小球)
```

```
 "", 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 (绘制小球上部的立柱)
```

```
 "", 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 (绘制更小的球)
```

```
 "", false, "red", null, "1tgray", 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 (绘制顶部天线)
```

图像绘制的效果如下图（注意在 1.6.6 版及其以前版本中，用户需要点击放大镜中有一个小 1 的按钮以调整 x, y 和 z 的比例为 1: 1: 1，还需要点击齿轮按钮隐藏坐标轴和标题）：



图 5.20：用 plot3d 函数绘制上海地标东方明珠电视塔。

Plot_3d_surfaces 函数不但可以绘制曲面，还可以绘制曲线。注意，在绘制曲线的时候， u 和 v 中只有一个能发挥作用，否则就是曲面了。比如要绘制一条螺旋线，半径为 5，可以将 u 看作幅角，变化范围从 -2π 到 2π ， x 的表达式为 $5\cos(u)$ ， y 为 $5\sin(u)$ ， z 为 u ，绘制语句为

```
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")
```

螺旋曲线绘制结果如下：

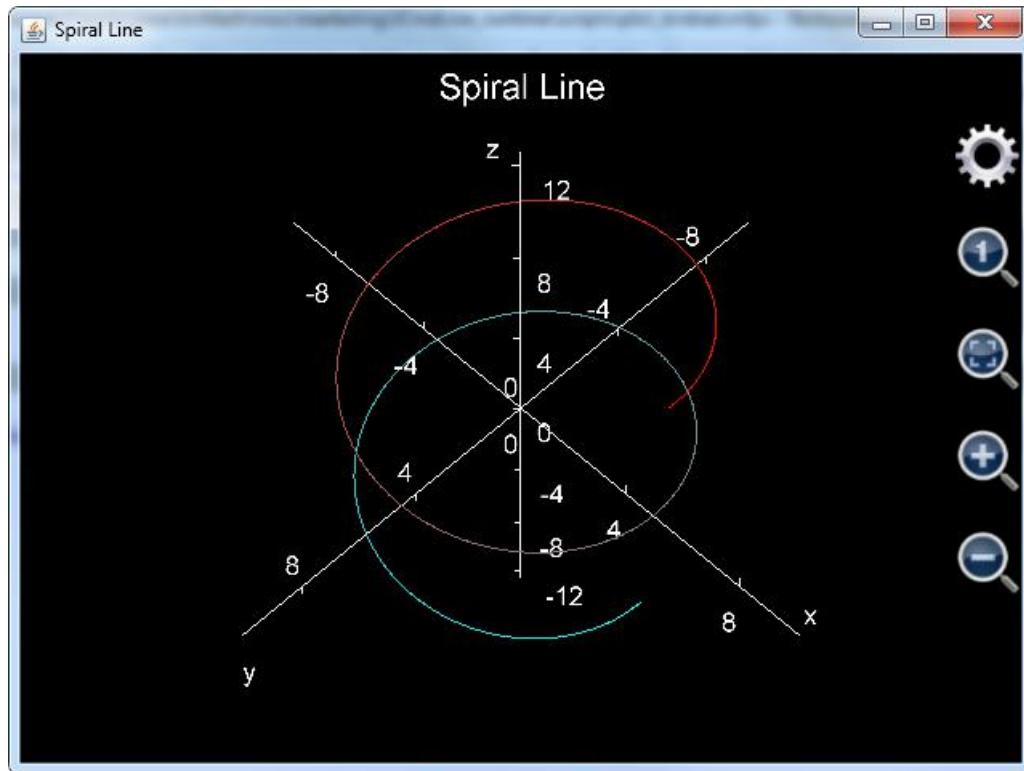


图 5.21：用 `plot_3d_surfaces` 函数绘制螺旋曲线。

这里需要注意几点：

第一点，虽然绘制曲线时，只有 u 起作用，但是 v 最好不要随意设置，因为 v 的步数乘以 u 的步数就是最终要绘制点的个数。如果想要图形尽可能快地绘制出来，可以设置 v 从 0 到 1 步长为 1，也就是只有一步；

第二点，如果是绘制曲线，参数是否仅仅绘制网格需要为 `true`，否则，曲线的颜色会变成和坐标轴一样的灰色而看不清楚；

第三点，绘制上图所示的螺旋线，步数要足够多，否则就不够平滑，基于这个原因，上面的例子中步长为 $\pi/50$ ，也就是总共有 200 步。

`Plot_3d_data` 则用于在三维坐标系中绘制数据图像。`Plot_3d_data` 有两种调用方式，第一种调用方式仅仅输入一个参数，用于绘制一个曲面。该参数必须是一个二维矩阵，矩阵中的每一个元素都必须是实数，表示绘制出

来的曲面上的一个点的 z 值，该元素在第一维的次序号对应于 x 值，在第二维的次序号对应于 y 值。比如：

```
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]])
```

对应的绘制出的图形为（注意已经将图形旋转放大以便于观察）：

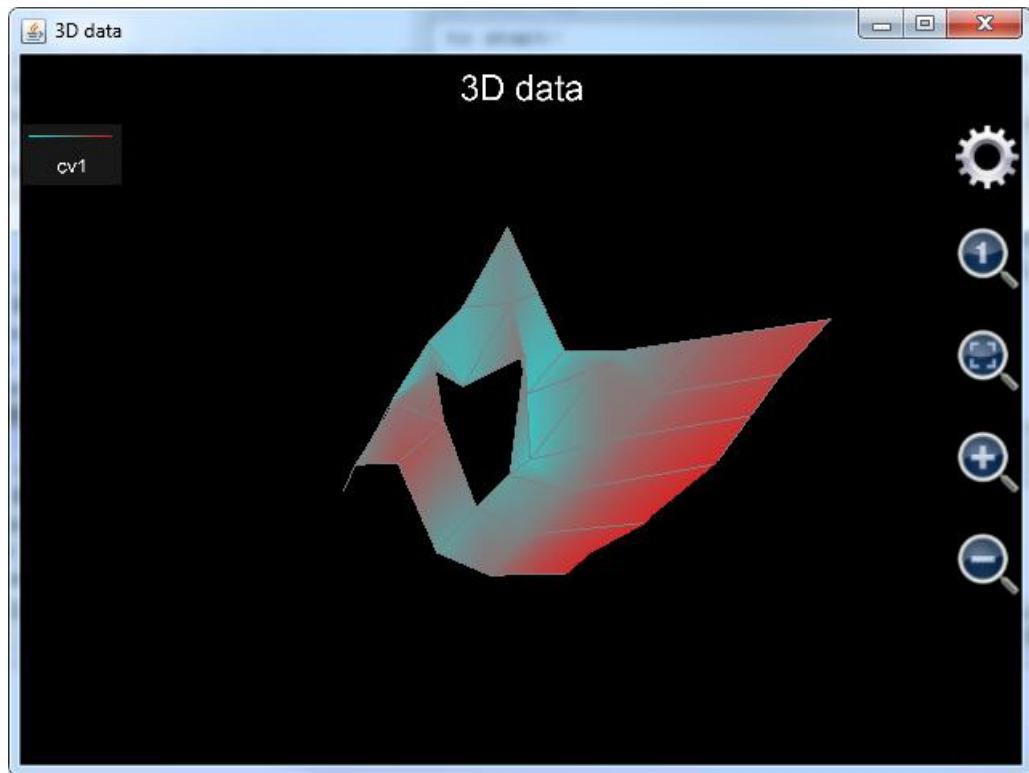


图 5.22：用 plot_3d_data 函数绘制单一曲面。

注意到曲面的中间有一个洞，这是因为曲面所对应的数据数组中有一个点的值为 Nan，Nan 是无法被绘制出来的，所以此点所在的位置形成了一个洞。

Plot_3d_data 也可以用来同时绘制多个曲面。在这种情况下，每绘制一个曲面需要 3 个参数组成一个参数组。在一个参数组中，第一个参数必须是一个一维数组，参数中的元素值决定的决定曲面中各点的 x 值，第二个参数也必须是一个一维数组，参数中的元素值决定的决定曲面中各点的 y 值，第三个参数必须是一个二维矩阵，矩阵中的元素值决定每一个点的 z

值。注意第三个参数的第一维的长度必须和第一个参数的长度一致，第三个参数的第二维的长度必须和第二个参数的长度保持一致。

在工程上，工程师经常需要观察比较数据在三维中的分布情况，这时，就可以使用 plot_3d_data 函数同时绘制多个三维曲面。比如，在金融工程中，风险分析师经常需要分析一组期权产品的波动率曲面（Volatility surface），并根据波动率曲面来决定某一个期权的定价，可以用调用如下语句同时绘制两个波动率曲面：

```
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], _ //曲面 1 的 x, y 坐标

[[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, 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, 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,  
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]], _ //曲面 1 的波动率二维矩阵
```

```
[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], _ //曲面 2x 和 y 坐标
```

```
[[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, 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, 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]]) // 曲面 2 的波动率二维矩阵

```

上述语句中，曲面 1 和 2 的 x 和 y 坐标均为一个一维向量，其中，x 表示期权的价值状态（Moneyness），y 表示期权的有效期，z 坐标则表示对应于某个价值状态和有效期的期权的理论波动率。注意，由于数据的问题，有时候计算出来的理论波动率是不合理的，对于这种坏的点，在金融分析上面必须舍弃，plot_3d_data 所具有的忽略值为 Nan 的点的特征，正好能够实现这个要求，这就是我们看到 z 坐标的二维矩阵内部有很多 Nan 的原因。

上述语句最后绘制出的图像如下，需要注意的是，如果用户使用的是 1.6.7 版及其以后版本的可编程科学计算器，在图形刚刚绘制出来的时候，由于图形本身在 x、y 和 z 坐标轴上的跨度不成比例，看起来就像一条带子，用户无法观察波动率分布的细节。这时，用户需要点击红色方框中的自适应

按钮，让软件自动调整 x、y 和 z 的比例，以便于观察，参见下图 5.23：；如果用户使用的是 1.6.6 及其以前版本的可编程科学计算器，则无需点击自适应按钮，图像绘制出来的初始状态就是自适应 x、y 和 z 的比例。

调整后的图形参见下图 5.24：，注意为了让两个波动率曲面正好位于图像正中，已经将 z 轴平移了 -0.3。并且坐标轴已经被设置为显示，以方便用户看清 x、y 和 z 轴单位长度的比例。

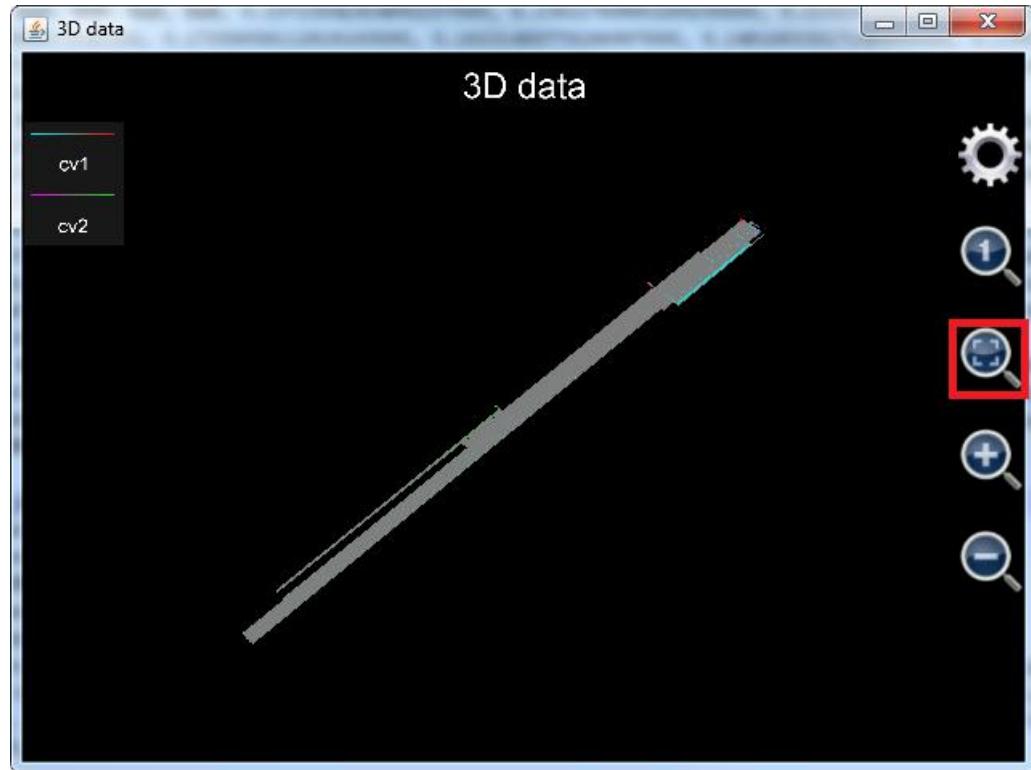


图 5.23：用 plot_3d_data 函数同时绘制两个波动率曲面（未经调整）。

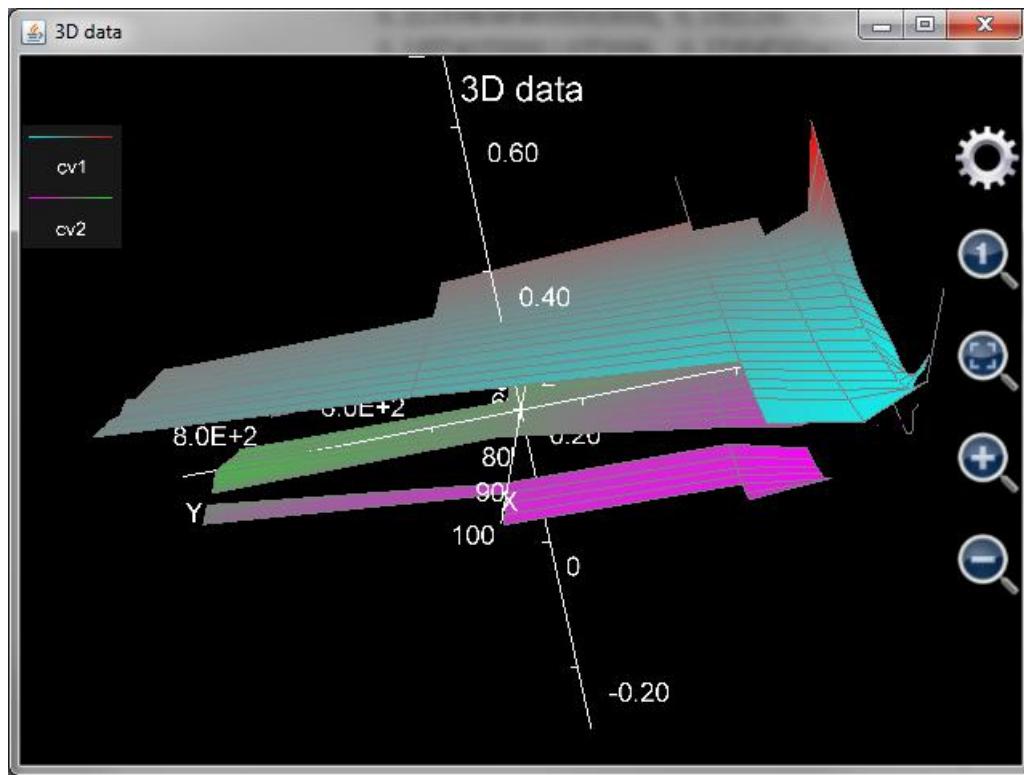


图 5.24： 调整后的两个波动率曲面。

最后需要注意，`plot_3d_data` 函数并不保存生成的图像，这和 `plot_2d_data` 是一样的。

`Plot_multi_xyz` 则是非常底层的函数。这个函数和 `plot_3d_data` 一样是直接绘制数据，而不是根据表达式求得每个点的数据值然后加以绘制。

`Plot_3d_data` 直接调用这个函数（但也要对数据做一些转换）。而 `plot3d` 则是先将表达式的每一个数据点的值都计算出来，然后再调用 `plot_multi_xyz`。

`Plot_multi_xyz` 的第一个参数为基于字符串的图像的名字，也就是去掉扩展名.`.mfpc` 的图像文件名。

`Plot_multi_xyz` 的第二个参数为图像的设置，这个参数是把所有的图像级别的设置放在一个字符串里面，每一个设置都基于

设置项目:设置项目的值；

的模式，然后将各个设置的首尾连接在一起。比如：

```
"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"
```

，这里的 chart_type 是图形类型，必须为 multiXYZ，chart_title 为图像标题，x_title 为图像 x 轴的名称，x_min 为最开始显示 x 轴的范围的最小值，x_max 为最开始显示 x 轴的范围的最大值，x_label 表示 x 轴上有多少个刻度标记，y_title 为图像 y 轴的名称，y_min 为最开始显示 y 轴的范围的最小值，y_max 为最开始显示 y 轴的范围的最大值，y_label 表示 y 轴上有多少个刻度标记，z_title 为图像 z 轴的名称，z_min 为最开始显示 z 轴的范围的最小值，z_max 为最开始显示 z 轴的范围的最大值，z_label 表示 z 轴上有多少个刻度标记。

Plot_multi_xyz 的第 3 个参数为一条曲线（面）的设置。这个参数是把所有的该曲线（面）的设置放在一个字符串里面，每一个设置都基于

设置项目：设置项目的值；

的模式，然后将各个设置的首尾连接在一起。比如：

```
"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"
```

，这里 curve_label 是曲线（面）标题，is_grid 表示是否仅仅绘制网格，min_color 表示正面 Z 坐标方向上最小值对应的颜色，min_color_1 表示反面 Z 坐标方向上最小值所对应的颜色，min_color_value 是最小的 z 值（注意，这个最小的 z 值不见得是曲面在 Z 坐标方向上的最小值，它只是定义了颜色的变化，也就是，图案中任何小于最小 z 值的部分都被涂以最小 z 值对应的颜色，图案中大于最小 z 值的部分的颜色从最小 z 值对应的颜色向最大 z 值所对应的颜色渐变。如果该值设置为 null，则让软件自己寻找曲面或曲线的最小 z 值），max_color 表示正面 Z 坐标方向上最大值对应的颜色，max_color_1 表示反面 Z 坐标方向上最大值所对应的颜色，

`max_color_value` 是最大的 z 值（注意，这个最大的 z 值不见得是曲面在 Z 坐标方向上的最大值，它只是定义了颜色的变化，也就是，图案中任何大于最大 z 值的部分都被涂以最大 z 值对应的颜色，图案中小于最大 z 值的部分的颜色从最大 z 值对应的颜色向最小 z 值所对应的颜色渐变。如果该值设置为 null，则让软件自己寻找曲面或曲线的最大 z 值）

`Plot_multi_xyz` 的第 4 个参数为该曲线（面）上每一个点在 x 轴上的坐标，注意这个参数必须为一个二维数组，数组中的每一个元素必须是实数，和 `plot_3d_data` 一样，如果该元素是 Nan，则曲线（面）的连接线将在该点断开。

`Plot_multi_xyz` 的第 5 个参数为该曲线（面）上每一个点在 y 轴上的坐标，注意这个参数必须为一个二维数组，数组的尺寸和第四个参数必须一致，数组中的每一个元素必须是实数，和 `plot_3d_data` 一样，如果该元素是 Nan，则曲线（面）的连接线将在该点断开。

`Plot_multi_xyz` 的第 6 个参数为该曲线（面）上每一个点在 z 轴上的坐标，注意这个参数必须为一个二维数组，数组的尺寸必须和第四个以及第 5 个参数的尺寸相符合，数组中的每一个元素必须是实数，和 `plot_3d_data` 一样，如果该元素是 Nan，则曲线（面）的连接线将在该点断开。

如果用户想要绘制不止一条曲线，则需要输入另外一组参数 3, 4, 5 和 6。用户最多可以绘制 1024 条曲线，所以 `plot_multi_xyz` 最多支持 $2+1024*4$ 等于 4098 个参数。

`Plot_multi_xyz` 的例子如下：

```
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:r  
ed", [[[-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]])
```

上述例子绘制的图像为一张类似于大致平放在 x 轴 y 轴所构成的平面上的纸，然后大致沿着 x 轴从正 y 轴方向向负 y 轴方向折叠，但并没有完全叠合起来。这个例子是为了展现 `plot_multi_xyz` 函数完全可以绘制扭曲的表面，甚至在该表面上同一个 x, y 坐标对应于两个不同的 z 值。

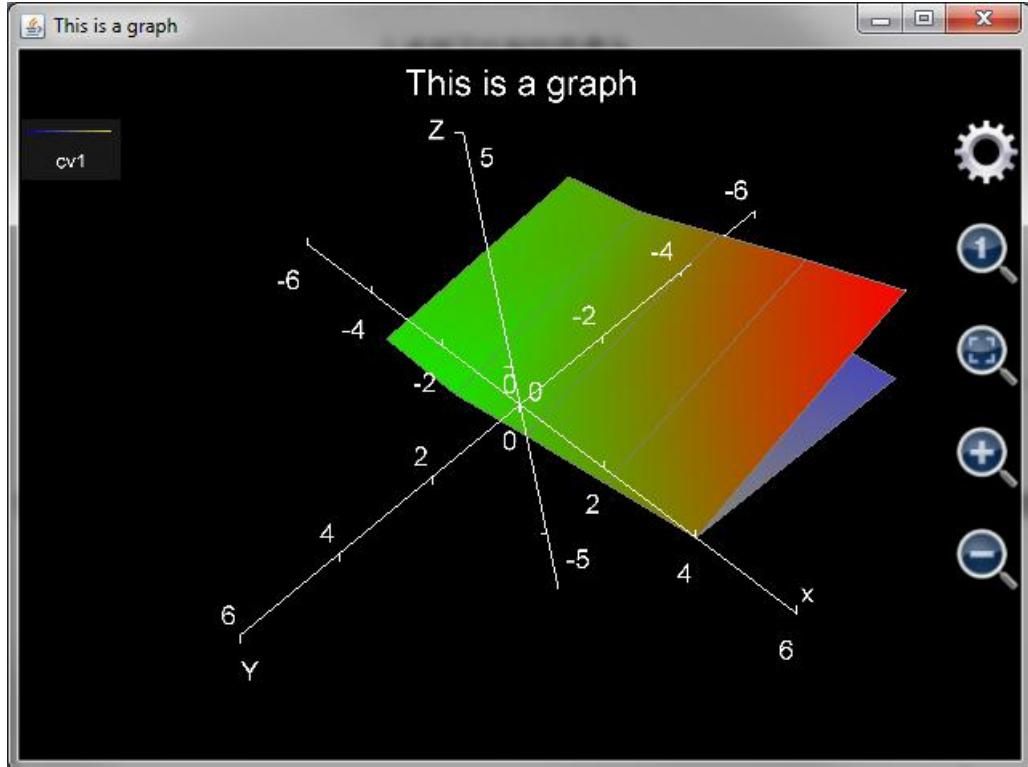


图 5.25：用 `plot_multi_xyz` 函数绘制折叠过的表面。

事实上，由于 `plot_multi_xyz` 函数只是对一组空间上的点进行绘图，而对这些点的 x, y 和 z 坐标没有要求，这意味着只要知道了图形的坐标点，用户可以在三维空间中绘制出任何想要的图形。

小结

本章详细介绍了调用 MFP 绘图函数进行绘图的方法。绘图函数可以分为 2 种。第一种是直接绘制表达式图形。用户在调用函数时无需输入变量变化范围，在图形生成之后，用户可以对绘图范围进行调整。这种绘图方法简单方便，但是代价是每次调整绘图范围都要部分或者全部重新计算每个绘制点的坐标，所以用户可能会感到时延。

第二种绘图函数是需要输入各个变量基于 t (二维图形) 或者 u, v (三维图形) 的变化表达式和变化范围。这样的绘图函数参数输入比较复杂，但是绘制图像完成之后，用户对图像进行拖拽或者缩放，不会有迟滞的感觉，此外，用这种绘图方式可以绘制出非常复杂的图形，比如北京的地标“鸟巢”和上海的地标“东方明珠”电视塔。从理论上讲，完全可以通过使用这些绘图函数绘制出任何用户想要的图形。

第6章 输入输出和文件操作

MFP 应用程序的输入输出是和用户交互的重要手段。MFP 编程语言实现了一组类似 C 语言的简单快捷的输入输出方式，用户可以方便地从命令提示符终端，字符串或者文件中读取信息，或者向命令提示符终端，字符串或者文件输出信息，但要注意输入输出的功能只在命令提示符和基于 JAVA 的可编程科学计算器中实现，在智慧计算器中不支持。

为了方便创建，复制，移动和删除文件，MFP 编程语言提供了一组类似于 Dos（或者 Unix）命令的文件操作函数，使用这些函数，用户可以轻松地查阅任何一个文件夹中的内容，并对任何一个文件和文件夹进行进行复制移动或者删除操作，就如同在 Dos 窗口或者 Linux 终端中一样。MFP 对文件操作的完整支持使得 MFP 编程语言从理论上成为一种无所不能的语言，用户可以通过 MFP 程序访问操作系统的任何一个角落。

第1节 在命令提示符中输入输出

在命令提示符中（或者运行在 JAVA 上的可编程科学计算器中）输入输出是用户和程序交互的一种重要方式。MFP 编程语言提供了以下支持函数：

函数名	函数帮助信息
input	<p>input(2) :</p> <p>input(prompt,input_type) 函数打印出提示符 prompt 并等待用户输入。第二个参数 input_type 可以省略。如果第二个参数不省略并且其为字符串"s"或"S"，用户的输入将被看成一个字符串，并且这个函数返回输入的字符串。否则，输入被当作一个表达式来处理，而这个函数返回表达式的值。如果输入的不是一个合法的表达式，本函数将再次打印出提示符，等待用户重新输入。用户如果要结束一次输入需要按回车键。如果一次输入多行，只有第一行会被处理。本函数的一个例子为，用户运行 input("\$", "S")，然后在提示符（也就是\$字符）后输入 4 + 3，按回车键，本函数将会返回字符串"4 + 3"。而如果用户运行 input("%")，然后在提示符（也就是%字符）后输入 4 + 3，按回车键，本函数将会返回 7。</p>

pause	<p>pause(1) :</p> <p>pause(message) 将暂停当前运行的程序，等待用户输入一个回车，然后继续。字符串参数 message 可以省略。如果不省略，message 将作为提示被打印在屏幕上。</p>
print	<p>print(1) :</p> <p>print(x) 向输出端打印任意数据类型 x 的值。</p>
printf	<p>printf(1...) :</p> <p>printf(format_string, ...), sprintf(format_string, ...) 和 fprintf(fd, format_string, ...) 和 C/C++ 中的对应函数工作方式相似。这些函数通过 format_string 和其后的数值参数构造出一个新的字符串，printf 函数将字符串打印到标准输出，sprintf 函数将字符串作为返回值返回，fprintf 函数则将字符串输出到文件号为 fd 的文本文档。字符串 format_string 支持输入整数 (%d、%i、%x 等)，浮点数 (%e、%f 等)，字符和字符串 (%c 和 %s) 等等。用户可以在 C 语言的帮助文档中找到 format_string 的构造方法。例如，printf("Hello world!%f", 3.14) 将会打印输出 "Hello world!3.140000"，而 sprintf("%c%d", "A", 9) 则返回 "A9"（注意 MFP 不支持单一字符数据类型，所以单一的字符将会存储为一个只包括一个字符的字符串）。</p>
scanf	<p>scanf(1) :</p> <p>scanf(format_string), sscanf(input_from, format_string) 和 fscanf(fd, format_string) 和 C/C++ 中的对应函数工作方式相似。scanf 读取用户的一行输入，sscanf 读取字符串 input_from，fscanf 从文件（文件号 fd）中读取文件内容。字符串 format_string 支持输入整数 (%d、%i、%x 等)，浮点数 (%e、%f 等)，字符和字符串 (%c 和 %s) 等等。用户可以在 C 语言的帮助文档中找到 format_string 的构造方法。但是，和 C 语言有所不同，MFP 中的这些函数不需要输入用于存储读取数值的参数。所有的读取的数值将会保存在一个数组中作为返回值返回。比如，sscanf("3Hello world!", "%d%c%c%s") 将会返回 [3, "H", "e", "llo"]（注意 MFP 不支持单一字符数据类型，所以单一的字符将会存储为一个只包括</p>

一个字符的字符串)。

其中, input, pause 和 print 函数的用法都很简单, 上述表格中的说明已经很详细, 就是要注意 print 函数的参数是一个字符串, 而字符串可以由一个字符串(哪怕是空字符串)和其他任何数据相加而得。比如要打印数组 [1, 2, 3+4i], 可以调用 print(""+[1, 2, 3+4i]), 这样 [1, 2, 3+4i] 就自动变成了字符串 [1, 2, 3 + 4i] 然后被打印输出到屏幕上。需要注意的是, 如果要调用 print 打印一些用于界定字符串或者表示特殊字符规避符, 比如双引号, 或者\, 需要在前面加上\。比如,

```
print("\\"\\")
```

会在屏幕上打印出"\。这个需要加双引号的特性不但适用于 print 函数, 也适用于所有对字符串操作的函数包括 input, pause 和后面要提到的 printf, scanf 等等。

下面给出了上述三个函数的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 io and file libs 子目录中的 examples.mfps 文件中找到:

```
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") //输入一个数字

    variable b = input("Please iinput a string:", "S") //请输入一个字符串
```

```
pause("Press ENTER to continue!") //按回车键继续

//print what has been input

//打印出输入的数字和字符串

print("You have input " + a + b)

//打印特殊字符

print("\n\\\"")

endf
```

用户在第一个提示符后输入 sind(30)*2+3i，在第二个提示符后输入“是一个复数”5个汉字（注意在安卓系统的命令提示符中输入汉字，需要点击菜单按钮，选择“弹出系统软件盘”，在基于JAVA的可编程科学计算器中无法输入汉字，不过可以拷贝粘贴汉字到输入提示符上），然后按ENTER键，可以看到打印输出为：

You have input 1 + 3i 是一个复数

"\

Printf 和 scanf 函数用于格式化输入输出，用法类似于 C 语言中的同名函数（当然没有 C 语言中的对应函数的用法那么全）。

printf() 函数的基本调用格式为：

```
printf("<格式化字符串>", <参量表>)
```

其中格式化字符串包括两部分内容：部分是正常字符，这些字符将按原样输出；另一部分是格式化规定字符，以“%”开始（如果仅仅是想用 printf 打印一个“%”字符，需要在前面加上规避符“%”，也就是“%%”），后跟一个或几个规定字符，用来确定输出内容格式。

参量表是需要输出的一系列参数，其个数必须与格式化字符串所说明的输出参数个数一样多，各参数之间用“,”分开，且顺序一一对应，否则将会出现意想不到的错误。

以下是 printf 函数所支持的格式化方式：

%d: 代表十进制整数;

%f: 代表浮点数;

%s: 代表字符串;

%e: 代表指数形式的浮点数;

%x 和%X: 以十六进制表示的整数, 注意输出以 16 进制表示的整数时, 没有 0x 打头, 比如输出十进制整数 255, 16 进制数值为 FF, 在 MFP 中表示 16 进制整数应该由 0x 打头, 也就是写成 0xFF 的形式, 但是 printf 输出时, 就是输出 FF, 没有 0x 打头;

%o: 以八进制表示的整数, 注意同 16 进制一样, 输出时没有 0 打头。比如以 8 进制输出 10 进制整数 63, MFP 中的表示方法应该是 077, 而 printf 输出时, 就是输出 77, 没有 0 打头;

%g: 为实数自动选择合适的表示法, 但无法处理字符串。

比如, 如果用户想要输出一个浮点数 8.776, 后面跟随一个逗号, 然后是一个字符串 "Hello", 后面跟随一个空格用十六进制输出一个整数 1234, 后面跟随 3 个字母 abc, 然后用普通十进制输出另外一个整数 255, 后面跟随一个英文单词 finish, 则可以调用 printf 语句如下:

```
printf("%f, %s %Xabc%dfinish", 8.776, "Hello", 1234, 255)
```

上述语句输出的结果为:

```
8.776000, Hello 4D2abc255finish
```

注意输出的结果中, 16 进制的整数没有 0x 打头 (也就是 1234 被直接输出为 4D2 而不是 0x4D2)。

用户 还需要注意的是, 需要输出的参数必须是能够使用 printf 所指定的格式化方式输出的数据类型, 或者通过数据类型自动转换, 转换为能够使用 printf 所指定的格式化方式输出的数据类型, 否则会出错, 比如

```
printf("%d", 123)
```

```
printf("%f", 123.1)
```

均为 printf 的正确调用方式，因为 123 是整数，而%d 正好是用于格式化输出整数；123.1 是浮点数，%f 正好用于格式化输出浮点数。

```
printf("%f", 123)
```

```
printf("%d", 123.1)
```

也都是对的，因为 123 可以被自动转换为浮点数，而 123.1 可以通过截断取整自动转换为整数。但是，

```
printf("%d", 123 + 123i)
```

```
printf("%f", [1.2, 2.3])
```

则会报错，因为 123+123i 是复数，复数无法自动转换为整数，而[1.2, 2.3]是数组，数组无法自动转换为数（包括浮点数）。

如果想正确地输出复数或者数组，可以用如下方式调用 printf 函数

```
printf("%d + %di", 123, 123) //输出 123 + 123i
```

```
printf("[%f, %f]", 1.2, 2.3) //输出[1.200000, 2.300000]
```

或者更简单的办法，使用字符串格式化输出方式，也就是%s：

```
printf("%s", 123 + 123i) //输出 123 + 123i
```

```
printf("%s", [1.2, 2.3]) //输出[1.2, 2.3]
```

。由于 MFP 中的任何数据类型都可以被转换为字符串，所以字符串格式化输出方式，也就是%s，可以说是万能的。

格式化输出方式的 "%" 字符和代表输出方式的字母之间还可以插入一些字符进行更精细的控制，比如，可以在 "%" 和字母之间插进数字表示最大宽度。例如：

%3d 表示输出 3 位整型数，不够 3 位右对齐；

%9.2f 表示输出场宽为 9 的浮点数，其中小数位为 2，整数位为 6，小数点占一位，不够 9 位右对齐；

%8s 表示输出 8 个字符的字符串，不够 8 个字符右对齐。

以上例子中，如果字符串的长度，或整型数位数超过说明的场宽，将按其实际长度输出。但对浮点数，若整数部分位数超过了说明的整数位宽度，将按实际整数位输出；若小数部分位数超过了说明的小数位宽度，则按说明的宽度以四舍五入输出。

另外，若想在输出值前加一些 0，就应在场宽项前加个 0。例如：%04d 表示在输出一个小于 4 位的数值时，将在前面补 0 使其总宽度为 4 位。

如果用浮点数表示字符或整型量的输出格式，小数点后的数字代表最大宽度，小数点前的数字代表最小宽度。例如：%6.9s 表示显示一个长度不小于 6 且不大于 9 的字符串。若长度大于 9，则第 9 个字符以后的内容将被删除。

另外，printf 还支持换行符\n，回车符\r 和缩进符\t，这些字符串可以用到 printf 的格式化字符串参数中，比如：

```
printf("abc\ndef")
```

的输出结果为两行，内容分别是 abc 和 def。其原因是因为\n 符号意味着换行。

Scanf 则和 printf 的功能正好相反。Scanf 从命令提示符终端上读取用户的一行输入（注意只能是一行，用户按 ENTER 键换行则输入终止）并返回一个数组，数组中的每一个元素是用户的输入的每一个数值。Scanf 仅有一个参数，是输入格式化字符串，表示用户输入的每一个元素将被 scanf 当作怎样的数据类型来处理。Scanf 支持以下格式化方式：

%d：代表输入十进制整数；

%f 和%e：代表输入浮点数；

%s：代表输入字符串；

`%x`: 代表输入以十六进制表示的整数（注意没有 `0x` 打头，比如 `0x10AB` 就应该写成 `10AB`，这和 MFP 中表示 16 进制的整数必须用 `0x` 打头不同，但是和 `printf` 的处理方式一致。还要注意，这里的 `x` 必须小写，`%X` 不被支持）；

`%o`: 代表输入以八进制表示的整数（注意没有 `0` 打头，比如 `017` 就应该写成 `17`，这和 MFP 中表示 8 进制的整数必须用 `0x` 打头不同，但是和 `printf` 的处理方式一致）；

`%g`: 代表输入一个十进制的实数。

在 `scanf` 的参数，也就是格式化字符串里，格式化方式表示符（也就是`%d`, `%f` 这些）之间，用户可以输入一些其他的字符用于分隔键盘输入。`Scanf` 在读取输入时，会跳过这些所指定的字符，然后读取下一个输入数。`Scanf` 所支持的分割字符可以包括空白字符（也就是空格，缩进符`\t`, 换行符`\n` 和回车符`\r`）和非空白字符（对于一些特殊的非空白字符，比如`%`或者`\`，在格式化字符串中需要用特殊的方式来表示：`%`必须被写为`%%`, `\`必须被写作`\\\`，“必须被写作`\”`，这和 `printf` 是类似的）。空白字符会使 `scanf()` 函数在读操作中略去输入中的一个或多个空白字符。而一个非空白字符会使 `scanf()` 函数在读入时剔除掉与这个非空白字符相同的字符。而如果在格式化方式表示符之间没有任何字符，`scanf` 函数自动将零个（注意是 0 个）、一个或多个相连的空格字符视作用户输入的数据之间的分隔符。

`Scanf` 的读取输入的策略是读到不能读为止，比如 `scanf` 的格式化字符串是`"%d%s"`，用户的输入是 `123abd`，那么 `scanf` 先输入一个十进制整数，`123` 被读入，但是当 `scanf` 遇到 `a` 时，`a` 显然不是十进制整数的一部分，所以，`scanf` 这时候就开始尝试用格式化字符串`%d` 之后的部分开始匹配，也就是`%s`, `%s` 是输入字符串，所以会读入 `a` 和 `a` 之后所有的字符，也就是`abd`。

如果 `scanf` 在读取过程中，发现某个格式化表示符所对应的数据无法被读出，则 `scanf` 会终止并返回一个数组包括所有已经读入的数据，比如，`scanf` 的格式化字符串为`"%f %d%s"`，用户输入 `1.23 xabd`，第一个格式化表示符`%f` 对应的输入为 `1.23`, `scanf` 能够读出，然后是空格，然后 `scanf` 在读取`%d` 所对应的整数时，找不到任何 10 进制数字，所以 `scanf` 终止返回数组`[1.23]`。

以下是使用 `scanf` 读取用户输入的一个例子，`scanf` 的调用语句如下

```
Scarf ("%f\n%x%dABC%s%d%s %e")
```

用户输入

```
8.67 6E8d 232ABC hello 12
```

最后的输出结果为[8.67, 28301, 232, "hello", 12, ""]。注意这里，ABC 之后需要输入一个字符串，但是 ABC 之后是空格，空格将会被自动当作格式化表示符之间的间隔而被 `scanf` 跳过，所以最后输入的字符串时"hello"而不是" hello"，而格式化字符串最后的格式化表示符`e`没有对应的输入，所以 `scanf` 终止返回。

`Scarf` 的格式化表示符还具有设置最多输入字符数目的功能，比如`%2s` 表示输入字符串时，最多输入两个字符，而诸如`%3f`，则表示输入浮点数时，最多输入三个字符，比如如下例子：

```
scanf ("%3f %2s")
```

， 用户输入为

```
1235.324 aerf
```

最后的输出结果为[123, "5."]。原因是，`%3f` 仅仅从用户输入中读取 3 个字符作为浮点数输入，所以输入了 123，然后 `scanf` 继续读取格式化字符串，发现是空格，而如上所述，格式化字符串中的空格对应于用户输入的 0 个，1 个或者多个连续的空格符，由于用户输入 3 之后没有输入空格，所以格式化字符串中的这个空格对应 0 个用户输入的空格，然后 `scanf` 继续读取格式化字符串发现是`%2s`，所以接着从用户输入中读取 2 个字符组成字符串"5."并返回。

最后还要注意，和 `printf` 函数不同，`scanf` 不能自动进行数据类型转换，当且仅当用户输入的数据符合格式化表示符的时候，输入才能成功，否则 `scanf` 将会终止。比如语句

```
Scarf ("%d,%f")
```

尝试从用户输入中读取一个整数，一个浮点数，如果用户输入为

12. 34, 5

, scanf 只能返回[12]，原因在于 12.34 是一个浮点数，浮点数不能被自动截断取整为整数，所以 scanf 为%d 读取整数输入时，仅仅读到小数点“.”之前就停止了，但是 scanf 发现%d 之后必须要有逗号“,”作为数据间的分隔字符，但小数点并非逗号，所以读取失败，最后返回[12]。

下面给出了上述 printf 和 scanf 函数的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 io 和 file libs 子目录中的 examples.mfps 文件中找到：

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 (现在测试printf) \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 00000000012.234215  
  
printf("\n")  
  
printf("abc\ndef")  
  
printf("\n")
```

```
printf("Now test scanf function (现在测试 scanf) \n")  
  
variable result_of_scanf  
  
printf("Please input (请用户输入) :\n123abd\n")  
  
result_of_scanf = scanf("%d%s")  
  
// remember, to simply print % using printf, we need to use %%
```

```
// 记住，如果想用 printf 输出一个%符号，需要使用%%

printf("scanf(\"%%d%%s\") returns " + result_of_scanf)

printf("\n")

printf("Please input (请用户输入) :\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 (请用户输入) :\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 (请用户输入) :\n1235.324 aerf\n")

result_of_scanf = scanf("%3f %2s")

print("scanf(\"%3f %2s\") returns " + result_of_scanf)

printf("\n")

printf("Please input (请用户输入) :\n12.34,5\n")

result_of_scanf = scanf("%d,%f")

print("scanf(\"%d,%f\") returns " + result_of_scanf)

printf("\n")
```

```
endf
```

在命令提示符中运行上述::mfpxexample::printfscanf() 函数，得到的结果如下，注意这里的结果也包括用户的输入：

Now test printf function (现在测试 printf)

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 (现在测试 scanf)

Please input (请用户输入) :

123abd

123abd

scanf("%d%s") returns [123, "abd"]

Please input (请用户输入) :

1. 23 xabd

1. 23 xabd

scanf ("%f %d%s") returns [1. 23]

Please input (请用户输入) :

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 (请用户输入) :

1235. 324 aerf

1235. 324 aerf

scanf ("%3f %2s") returns [123, "5."]

Please input (请用户输入) :

12. 34, 5

12. 34, 5

scanf ("%d, %f") returns [12]

第 2 节 对字符串输入输出

和 printf 与 scanf 相似, MFP 编程语言也提供了 sprintf 和 sscanf 用于对字符串输入输出。sprintf 的用法和上一节中介绍的 printf 基本一样, 唯一的区别在于, printf 没有返回值, 而是将格式化后的数据打印输出到命令提示符的屏幕上, sprintf 不是将格式化后的数据打印输出到命令提示符的屏幕上, 而是作为一个字符串返回。

sscanf 的用法和上一节中介绍的 scanf 也是基本一样，但需要注意两点不同。第一，sscanf 需要两个参数，第一个参数是待读入的字符串，这就相当于 scanf 中读取用户输入的来源——命令提示符，第二个参数才是格式化字符串；第二，sscanf 读取的字符串可能包括多行，每一个换行符都被当作一种空白字符处理，而 scanf 一次只能从命令提示符终端中读取一行。除去这两点，sscanf 的格式化字符串参数的定义和格式，以及返回数组的方式，都和 scanf 一模一样。

下面给出了使用 sprintf 和 sscanf 函数的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 io 和 file libs 子目录中的 examples.mfps 文件中找到：

```
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 (现在测试 sprintf) \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 00000000012.234215

    printf(result_str + "\n")

    result_str = sprintf("abc\ndef")

    printf(result_str + "\n")



    printf("Now test sscanf function (现在测试sscanf) \n")

    variable result_of_sscanf

    result_of_sscanf = sscanf("123abd", "%d%s")

```

```
// remember, to simply print % using printf, we need to use %%
```

```
// 记住, 如果想用 printf 输出一个%符号, 需要使用%%
```

```
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
```

```
// 读取包括多行的字符串
```

```
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\")\nreturns " + 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
```

在命令提示符中运行上述::mfexample::sprintfsscanf()函数，得到的结果如下，注意运行上述函数无需用户的输入：

Now test sprintf function (现在测试 sprintf)

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 (现在测试 sscanf)

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]
```

第3节 文件内容读写及其相关函数

MFP 编程语言提供了完整的一套类似 C 语言的文件内容读写函数，其中不但包含类似于 printf 和 scanf，用于读写文本文件的 fprintf 和 fscanf，还包括用于读取文本文件行的 freadline，用于读写二进制文件的 fread 和 fwrite，以及文件的打开关闭函数 fopen 和 fclose。具体地，这些文件说明如下表所示：

函数名	函数帮助信息
fclose	<p>fclose(1) :</p> <p>fclose(fd)关闭文件号 fd 所对应的文件。如果文件号不存在，返回-1，否则返回 0。</p>
feof	<p>feof(1) :</p> <p>feof(fd)用于确定是否已经到达文件号为 fd 的读模式文件的末尾。如果是，返回 true，否则返回 false。如果文件号不合法，抛出异常。</p>
fopen	<p>fopen(2) :</p> <p>fopen(path, mode)打开位于 path 路径的文件并返回文件号以进行后续读写操作。它和 C 以及 Matlab 中的同名函数用法相似。但它仅支持"r"、"a"、"w"、"rb"、"ab"和"wb"六种读写模式。例子包括 fopen("C:\\Temp\\Hello.dat", "ab") (Windows) 和 fopen("./hello.txt", "r") (Android)。</p> <p>fopen(3) :</p> <p>fopen(path, mode, encoding)用字符编码 encoding 打开位于 path 路径的文件并返回文件号以进行后续读写操作。由于只有文本文件支持字符编码，参数 mode 只能为"r"、"a"和"w"3 种读写模式。例子包括 fopen("C:\\Temp\\Hello.txt", "a", "LATIN-1") (Windows)</p>

	和 fopen("./hello.txt", "r", "UTF-8") (Android)。
fprintf	<p>fprintf(2...) :</p> <p>printf(format_string, ...), sprintf(format_string, ...) 和 fprintf(fd, format_string, ...) 和 C/C++ 中的对应函数工作方式相似。这些函数通过 format_string 和其后的数值参数构造出一个新的字符串，printf 函数将字符串打印到标准输出，sprintf 函数将字符串作为返回值返回，fprintf 函数则将字符串输出到文件号为 fd 的文本文件。字符串 format_string 支持输入整数（%d、%i、%x 等），浮点数（%e、%f 等），字符和字符串（%c 和 %os）等等。用户可以在 C 语言的帮助文档中找到 format_string 的构造方法。例如，printf("Hello world!%f", 3.14) 将会打印输出 "Hello world!3.140000"，而 sprintf("%c%od", "A", 9) 则返回 "A9"（注意 MFP 不支持单一字符数据类型，所以单一的字符将会存储为一个只包括一个字符的字符串）。</p>
fread	<p>fread(4) :</p> <p>fread(fd, buffer, from, length) 从文件（文件号 fd）中读取 length 个字节数据，并把读出的数据保存在数组 buffer 中（从 buffer 的索引 from 开始保存）。注意 from 和 length 必须非负，并且 from+length 必须不比 buffer 的容量大。参数 from 和 length 可以同时省略。如果它们被省略，意味着 fread 读取整个 buffer 容量的字节数据并保存在整个 buffer 中。Buffer 也可以省略，如果 buffer 省略，fread 读取一个字节并返回。如果 fread 在读取之前发现已经到达文件末尾，则返回 -1，否则返回读取字节的个数（如果 buffer 不省略）。如果文件不存在或非法或不可以访问，将会抛出异常。例子包括 fread(1)、fread(2, byte_buffer) 以及 fread(2, byte_buffer, 3, 7)。</p>
freadline	<p>freadline(1) :</p> <p>freadline(fd) 读取文本文件（文件号是 fd）的一行。如果 freadline 在读取之前发现已经到达文件末尾，它返回 NULL。否则，它返回基于字符串的这一行的内容，但不包括结尾的换行符。</p>
fscanf	fscanf(2) :

	<p>scanf(format_string), sscanf(input_from, format_string)和 fscanf(fd, format_string)和 C/C++中的对应函数工作方式相似。scanf 读取用户的一行输入，sscanf 读取字符串 input_from，fscanf 从文件（文件号 fd）中读取文件内容。字符串 format_string 支持输入整数（%d、%i、%x 等），浮点数（%e、%f 等），字符和字符串（%c 和 %s）等等。用户可以在 C 语言的帮助文档中找到 format_string 的构造方法。但是，和 C 语言有所不同，MFP 中的这些函数不需要输入用于存储读取数值的参数。所有的读取的数值将会保存在一个数组中作为返回值返回。比如，sscanf("3Hello world!", "%d%c%c%s")将会返回[3, "H", "e", "llo"]（注意 MFP 不支持单一字符数据类型，所以单一的字符将会存储为一个只包括一个字符的字符串）。</p>
fwrite	<p>fwrite(4) :</p> <p>fwrite(fd, buffer, from, length)向文件（文件号 fd）中写入 length 个字节数据。这些字节数据保存在数组 buffer 中（从 buffer 的索引 from 开始）。注意 from 和 length 必须非负，并且 from+length 必须不比 buffer 的容量大。参数 from 和 length 可以同时省略。如果它们被省略，意味着 fwrite 写入整个 buffer 的字节数据。Buffer 也可以是一个单独的字节，在这种情况下 fwrite 仅写入一个字节的数据。如果文件不存在或非法或不可以访问，将会抛出异常。例子包括 fwrite(1, 108)、fwrite(2, byte_buffer)以及 fwrite(2, byte_buffer, 3, 7)。</p>

1. 打开和关闭文件

和 C 语言类似，MFP 对文件内容进行读写操作有一个基本的流程，首先，第一步要打开文件。打开文件的函数是 fopen，该函数有两个或 3 个参数，第一个参数为一个基于字符串的文件名，该文件名包括路径，这里的路径可以是绝对路径（比如“/mnt/sdcard/a.txt”或者“C:\\temp\\b.exe”），也可以是相对路径（比如“a.txt”或者“temp\\b.exe”）。注意这里的相对路径是相对当前目录的路径，在启动安卓上的可编程科学计算器的时候，初始的当前目录是位于 SD 卡上的 AnMath 目录，在启动基于 JAVA 的可编程科学计算器时，初始的当前目录是启动 JCcmdLine.jar 文件的所在目录。在可编程科学计算器启动之后，用户可以通过 cd 函数改变当前目录。

Fopen 函数的第二个参数是打开文件的方式（也是一个字符串），fopen 函数支持 6 种打开文件的方式，它们是

"r": 表示用读模式打开一个文本文件（通常文件扩展名为. txt，也有可能是. xml 或者. c、. cpp、. mfps 等代码文件，这些文件都可以用记事本软件打开）；

"w": 表示用写模式打开一个文本文件，文件的原有内容将会被取代；

"a": 表示用添加模式打开一个文本文件，新输入的内容将会被添加到文件的原有内容之后；

"rb": 表示用读模式打开一个二进制文件（这种文件通常无法用记事本打开）；

"wb": 表示用写模式打开一个二进制文件，文件的原有内容将会被取代；

"ab": 表示用添加模式打开一个二进制文件，新输入的内容将会被添加到文件的原有内容之后；

注意这里文本文件和二进制文件读写方式是不一样的，后面将会介绍，文本文件的读写函数是 fprintf, fscanf 和 freadline，而二进制文件的读写函数是 fread 和 fwrite。如果用读写二进制文件的函数去读写文本文件，或者用读写文本文件的函数去读写二进制文件，会出错。

Fopen 函数除了上述两个参数之外，还有一个可选参数，也就是被操作文件的编码模式（也是一个字符串）。该参数仅对文本文件有效。用正确的编码模式打开一个文本文件对于确保正确读取或者写入文件内容非常重要，否则，有可能会造成文件里面是汉字文章，用 MFP 读出来的结果却是乱码。

Fopen 函数的编码模式参数的缺省值是操作系统的编码模式，对于中文 Windows，编码模式是 GB2312，这种编码模式是支持中文的，所以，如果在中文 Windows 平台上用 fopen 函数在不指定编码模式的条件下打开一个文本文件并写入一些中文，再用一些常用的文本编辑软件比如 NotePad++ 打开，看到的不会是问号或者乱码。但是，对于英文的 Windows，编码模式通常是 IBM437，这种编码模式不支持中文，所以，在英文 Windows 平台上用

fopen 函数在不指定编码模式的条件下打开一个文本文件并写入一些中文，再用 NotePad++ 打开，看到的都是乱码或者问号。

在安卓上，文本文件的编码问题则简单得多。安卓系统的编码模式都是 UTF-8，不论是什么语言。UTF-8 支持中文，所以，在安卓系统上创建的中英文文本文件，都可以在常用的文本编辑器上正确打开。

由于阅读本手册的读者一般都是使用中文的 Windows 和安卓系统，所以，大部分用户不用管 fopen 的这个可选参数，而只用第一个和第二个参数即可。但如果是在英文 Windows 平台上读写含有汉字的文件，建议 fopen 最后增加指定编码模式的参数并设置该参数值为“UTF-8”。

Fopen 函数的返回值是打开的文件号，这是一个非负整数，每一次调用 fopen 函数，不论是操作同一个文件，fopen 返回的文件号都不一样。Fopen 返回的文件号将作为后面文件读写函数和关闭文件函数的参数，告诉这些函数哪个文件需要被读写，那个文件需要被关闭。

在打开文件并对其进行了读写操作之后，文件不在被使用，这时，就需要将文件关闭。MFP 中，关闭文件的函数名称为 fclose(fd)，这里 fd 为前面调用 fopen 函数所返回的文件号。

关闭已经打开但是不再被使用的文件时非常重要不可忽略的步骤。如果不关闭文件，首先会造成内存的泄漏，其次，如果以后需要再次打开没有关闭的文件进行写操作，有可能会失败。

以下给出了使用 fopen 和 fclose 的例子：

```
variable fd1 = fopen("test.exe", "wb") // 打开二进制文件 test.exe 写  
fclose(fd1)  
  
variable fd2 = fopen("/mnt/sdcard/AnMath/test.txt", "r", "UTF-8") //  
用 UTF-8 编码模式打开文本文件 test.txt 读  
fclose(fd2)
```

2. 文本文件的读写

在打开文件后，就需要对文件内容进行读写。MFP 仅提供了顺序读取（或者写入）的功能。也就是，文件打开后，MFP 只能向文件尾的方向读取（或写入），而不能跳转到文件的中间。这样，当 MFP 编程语言读取文件时，如果到达了文件的尾部，读取必须停止。MFP 提供了函数 `feof` 用于判断是否到达文件的尾部。`feof` 仅需要一个参数，就是 `fopen` 函数返回的文件号。如果到达文件尾部，`feof` 返回 `true`，否则返回 `false`。

MFP 提供的文本文件读取函数包括 `fscanf` 和 `freadline`。其中，`fscanf` 和前面介绍的 `scanf` 函数以及 `sscanf` 函数的使用方法基本一样。`fscanf` 需要两个参数，第一个参数是 `fopen` 函数返回的文件号，第二个参数是格式化字符串，返回值是一个数组，里面的每一个元素是一个从文件中读入的数据。注意，和 `sscanf` 函数一样，这个被读取的文件可以包括很多行，文件中的换行符被当作一种空格符处理。

`freadline` 函数则用来一行一行地读取文件。`freadline` 使用方法更简单，它只需要一个参数，就是文件号。每调用一次，返回文件中的一行，然后将文件的读取指针指向该行后面的字符。如果到达文件尾，则返回空字符串。

以下给出了读取文本文件的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 `io and file libs` 子目录中的 `examples.mfps` 文件中找到。

为了运行这个例子，首先需要在放置代码的 `io and file libs` 子目录中创建文本文件 `test_read.txt`，其内容如下：

123 456

Good, $2*9=18$

Hello!

abc

读取该文本文件的代码如下：

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.  
    // 这里假设当前目录是 AnMath 目录, test_read.txt 文件保存在  
    // scripts/manual/io 和 file libs/ 文件夹中。  
    // Content of the file (文件内容如下: )  
    //123 456  
    //Good, 2*9=18  
    //Hello!  
    //abc  
  
    variable fd = fopen("scripts/manual/io and file libs/test_read.txt", "r")  
    // read two integers  
    // 读取两个整数  
    variable int_array = fscanf(fd, "%d%d")  
    // print what has been read  
    // 打印出读取的内容
```

```
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!  
  
// 如果还没有到达文件尾部，并且还没有读到字符串 Hello!  
  
while and(feof(fd) == false, read_str != "Hello!")  
  
    read_str = freadline(fd)  
  
    // print what has been read  
  
    // 打印出读取的内容  
  
    printf("read " + read_str + "\n")  
  
loop  
  
variable str_array = fscanf(fd, "%s")  
  
// print what has been read  
  
// 打印出读取的内容  
  
printf("read " + str_array + "\n")  
  
if (feof(fd))  
  
    // it is right if we are at the end of the file  
  
    // 到达文件尾部意味着程序工作正常  
  
    print("End of the file found!\n")  
  
else  
  
    // it is wrong if we are still not at the end of file  
  
    // 如果还没有到达文件尾部，意味着出错  
  
    print("Something wrong!\n")  
  
endif
```

```
fclose(fd) //close the file (别忘记关闭文件)
```

```
endif
```

上述程序的运行结果如下：

```
read [123, 456]
```

```
read
```

```
read Good, 2*9=18
```

```
read Hello!
```

```
read ["abc"]
```

```
End of the file found!
```

需要注意一点的是，在读出整数 123 和 456 之后，上述代码调用 `freadline` 函数在 `while` 循环中一行一行地读取文件内容。但是，`freadline` 读出的第一行并非是 `Good, 2*9=18`，而是一个空字符串，其原因是，在 `freadline` 函数被调用之间，`fscanf` 函数仅仅读到 456 就返回了而没有继续读下去，这就意味着 456 之后文件中的换行符还没有被读入，所以，`freadline` 开始读的时候，是从 6 之后开始读到换行符为止，但是 6 和换行符之间没有字符，所以 `freadline` 返回一个空字符串。

MFP 提供的文本文件写入函数为 `fprintf`。`fprintf` 函数的用法和 `printf` 函数以及 `sprintf` 函数基本类似，唯一的区别在于，`fprintf` 多一个参数，也就是，`fprintf` 的第一个参数是 `fopen` 返回的文件号，通过这个文件号，`fprintf` 对相应的文件进行写入操作。`fprintf` 的第二个参数才是格式化字符串，从第三个参数起（如果有的话），是需要写入的数据。

以下给出了读取文本文件的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 `io and file libs` 子目录中的 `examples.mfps` 文件中找到。

```
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.

    // 首先测试写模式。如果要打开的文件不存在，那么它将会被创建。如果文件已经

    // 存在，它的原有的内容将会被新写入的内容所覆盖。

variable fd = fopen("scripts/manual/io and file libs/test_write.txt", _

    "w", "UTF-8")

    // first inputs some numbers and string with prompt information

    // 首先输入一些数字和字符（包括一些提示信息）

fprintf(fd, "The first line includes %d, %f, %s\n", 123.71, 56.48, "hi")

    // then input 4 Chinese characters with prompt information

    // 然后输入四个汉字（包括一些提示信息）

fprintf(fd, "Now input some Chinese characters: " + "汉字中文\n")

fclose(fd) // close the file (关闭文件)

    //

    // 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.  
  
// 然后测试添加模式。如果要打开的文件不存在，那么它将被创建。如果文件  
// 已经存在，将在它的原有的内容后添加新的内容。  
  
fd = fopen("scripts/manual/io and file libs/test_write.txt", _  
"a", "UTF-8")  
  
// inputs some numbers and string with prompt information  
  
// 输入一些数字和字符（包括一些提示信息）  
  
fprintf(fd, "Now add a new line %d, %f, %s\n", -999, -48.73, "why?")  
  
fclose(fd) // close the file (关闭文件)  
  
endif
```

上述代码运行完成之后，用户会在本手册自带的示例代码所在目录中的 io and file libs 子目录中的找到一个 test_write.txt 文件，该文件的内容如下：

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?

注意由于用户使用的是 UTF-8 编码模式来写入文件，所以，上述汉字可以在一些常用的文本编辑器，比如 notepad++ 中打开。

通过文本文件的读写，用户还可以实现 MFP 语言上并不支持的全局变量的功能。用户可以将全局变量的值写入一个文本文件中，并在其他的函数中读取该文本文件获得变量值，或者写入该文本文件更改变量值，这样，该文本文件就相当于一个全局变量，文件的内容就相当于变量的值。

3. 二进制文件的读写

MFP 编程语言读取二进制的函数名字叫 fread。Fread 函数使用四个参数，第一个参数是文件号 (fd)，这个参数是前面调用 fopen 函数打开一个文

件的返回值；第二个参数是读取的内容的保存位子（buffer），这个参数必须是一个数组；第三个参数是读出的数据在 buffer 中的起始存放的位子；第四个参数是读取的字节数。

需要注意的是，读出的数据在 buffer 中的起始存放的位子加上读取的字节数必须不大于读取的内容的保存数组的长度。第三个和第四个参数可以省略，如果省略，缺省的读出的数据在 buffer 中的起始存放的位子为 0，读取的字节数为 buffer 数组的长度。第二个参数也可以省略，如果省略，`fread` 函数读取一个字节并返回读出的数值，如果不省略，`fread` 把读取的数据保存在 buffer 中并返回读出的字节的个数。如果 `fread` 在读取数据之前发现已经到达文件的尾部，则返回 -1。为了避免 `fread` 函数在到达文件尾部后还继续尝试读取文件，和读取文本文件一样，也可以使用 `feof` 函数判断是否到达文件的尾部。如果文件不存在或者无法访问，则抛出异常。

MFP 编程语言写二进制文件的函数叫做 `fwrite`。`fwrite(fd, buffer, from, length)` 向文件（文件号 `fd`）中写入 `length` 个字节数据。这些字节数据保存在数组 `buffer` 中（从 `buffer` 的索引 `from` 开始）。注意 `from` 和 `length` 必须非负，并且 `from+length` 必须不比 `buffer` 的容量大。参数 `from` 和 `length` 可以同时省略。如果它们被省略，意味着 `fwrite` 写入整个 `buffer` 的字节数据。`Buffer` 也可以是一个单独的字节（而不是数组），在这种情况下 `fwrite` 仅写入一个字节的数据。如果文件不存在或非法或不可以访问，将会抛出异常。

以下给出了读写二进制文件的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 `io and file libs` 子目录中的 `examples.mfps` 文件中找到。

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"

    // 记住写二进制文件需要用"wb"而不是"w"

    variable fd = fopen("scripts/manual/io_and_file_libs/test_rw.bin", "wb")

    fwrite(fd, 108) // write one byte whose value is 108

        // 写入一个字节，值为 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

    // 注意这里的缓存必须是一个字节数组（字节，这里必须是不大于 128 不小于-127

    // 的整数）。buffer 的最后一个元素 1000 大于 255，所以在写入二进制文件时，

    // 将会被强制转化为一个字节范围内的整数也就是-24，在此过程中它的高位比特

        // 信息丢失。

    variable buffer = [-18, 79, 126, -55, 48, -23, -75, 7, 98, 6, 0, -34, 1000]

    fwrite(fd, buffer) //write every thing in the buffer into the file

        //写入 buffer 中的所有内容

    fclose(fd)

    // remember append binary file should use "ab" not "a"

    // 记住向二进制文件尾部添加内容需要用"ab"而不是"a"
```

```
fd = fopen("scripts/manual/io and file libs/test_rw.bin", "ab")  
  
    // write 7 bytes from index 3 of buffer  
  
    // 向文件中写 7 个字节，这七个字节从 buffer 的索引 3（也就是第 4 个字符）开始。  
  
    fwrite(fd, buffer, 3, 7)  
  
    fclose(fd)  
  
    //  
  
    //print original buffer content  
  
    // 打印出最开始 buffer 的内容  
  
    print("Originally buffer includes " + buffer + "\n")  
  
    // remember read binary file should use "rb" not "r"  
  
    // 记住读二进制文件需要用"rb"而不是"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  
  
    // 从二进制文件中读入 5 个字节，并把读入的字节保存在 buffer 中，  
  
    // 保存位置从索引 2 开始，也就是第三个字符  
  
    fread(fd, buffer, 2, 5)  
  
    print("Now buffer is " + buffer + "\n") // 打印出 buffer 的内容  
  
    variable byte_value = fread(fd) // read 1 byte (读一个字节)  
  
    print("Read one byte which is " + byte_value + "\n")  
  
    variable read_byte_cnt = fread(fd, buffer) // try to read buffer length  
  
                                            // bytes  
  
                                            // 尝试读取 buffer 长度个字节
```

```
print("Read " + read_byte_cnt + " bytes" + "\n") // print how many  
// bytes read  
// 打印出读出的字节数  
print("Now buffer is " + buffer + "\n") //打印出buffer 的内容  
read_byte_cnt = fread(fd, buffer) // try to read buffer length bytes  
// again  
// 再次尝试读取 buffer 长度个字节  
print("Read " + read_byte_cnt + " bytes" + "\n") // print how many  
// bytes read  
// 打印出读出的字节数  
// check if we have arrived at the end of file  
// 检查是否到达文件尾了  
if (feof(fd))  
    // check how many bytes we can read if we have arrived at the  
    // end of file  
    // 检查到达文件尾后还能读多少字节  
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  
// 再次尝试读取 buffer 长度个字节  
print("Read " + read_byte_cnt + " bytes" + "\n") // print how  
// many bytes
```

```
// read  
// 打印出读出的字节数  
endif  
fclose(fd)  
endif
```

程序运行的结果如下：

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

第4节 文件属性操作函数

MFP 编程语言提供了一系列的函数帮助使用者获取甚至修改文件的属性，包括文件名称，路径，类型，大小以及最近的修改时间等。所有这些函数列表及其详细说明如下：

函数名	函数帮助信息

get_absolute_path	<p>get_absolute_path(1) :</p> <p>get_absolute_path(fd_or_path)返回文件号 fd_or_path (这里 fd_or_path 是一个整数) 或者 相对路径为 fd_or_path (这里 fd_or_path 是一个字符串) 所对应的文件的绝对路径 (从根目录开始的完整路径) 字符串。</p>
get_canonical_path	<p>get_canonical_path(1) :</p> <p>get_canonical_path(fd_or_path)返回文件号 fd_or_path (这里 fd_or_path 是一个整数) 或者 相对路径为 fd_or_path (这里 fd_or_path 是一个字符串) 所对应的文件的标准路径 (不依赖符号链接的绝对路径) 字符串。</p>
get_file_last_modified_time	<p>get_file_last_modified_time(1) :</p> <p>get_file_last_modified_time(path)返回基于字符串路径的 path 的文件或目录的上一次更改时间。该时间等于从 1970 年 1 月 1 日午夜开始到上一次更改时刻所经历的毫秒数。如果 path 不存在或者没有访问权限，返回-1。</p>
get_file_path	<p>get_file_path(1) :</p> <p>get_file_path(fd)返回文件号 fd (fd 是一个整数) 所对应的文件的路径字符串。</p>
get_file_separator	<p>get_file_separator(0) :</p> <p>get_file_separator()返回路径分割符。在 Windows 平台上返回字符串"\\"，在 Linux 和 Android 平台上返回字符串"/"。</p>
get_file_size	<p>get_file_size(1) :</p> <p>get_file_size(path)返回基于字符串路径的 path 的文件大小。如果 path 不是对应一个文件或者没有权限或者不存在，返回-1。</p>

is_directory	<p><code>is_directory(1) :</code></p> <p><code>is_directory(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否是一个目录。如果该文件或目录存在并且是一个目录返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_directory("E:\\")</code> (Windows) 和 <code>is_directory("/home/tony/Documents/cv.pdf")</code> (Android)。</p>
is_file_executable	<p><code>is_file_executable(1) :</code></p> <p><code>is_file_executable(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否可执行。如果该文件或目录存在并且可执行返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_file_executable("E:\\")</code> (Windows) 和 <code>is_file_executable("/home/tony/Documents/cv.pdf")</code> (Android)。</p>
is_file_existing	<p><code>is_file_existing(1) :</code></p> <p><code>is_file_existing(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否存在。如果存在返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_file_existing("E:\\")</code> (Windows) 和 <code>is_file_existing("/home/tony/Documents/cv.pdf")</code> (Android)。</p>
is_file_hidden	<p><code>is_file_hidden(1) :</code></p> <p><code>is_file_hidden(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否隐藏。如果该文件或目录存在并且隐藏返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_file_hidden("E:\\")</code> (Windows) 和 <code>is_file_hidden("/home/tony/Documents/cv.pdf")</code> (Android)。</p>

is_file_normal	<p><code>is_file_normal(1) :</code></p> <p><code>is_file_normal(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否是一个常规文件而不是目录。如果该文件或目录存在并且是一个常规文件而不是目录返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_file_normal("E:\\")</code> (Windows)和 <code>is_file_normal("/home/tony/Documents/cv.pdf")</code> (Android)。</p>
is_file_readable	<p><code>is_file_readable(1) :</code></p> <p><code>is_file_readable(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否可读。如果该文件或目录存在并且可读返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_file_readable("E:\\")</code> (Windows)和 <code>is_file_readable("/home/tony/Documents/cv.pdf")</code> (Android)。</p>
is_file_writable	<p><code>is_file_writable(1) :</code></p> <p><code>is_file_writable(path)</code>用于判断位于字符串 <code>path</code> 的文件（或者目录）是否可写。如果该文件或目录存在并且可写返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_file_writable("E:\\")</code> (Windows)和 <code>is_file_writable("/home/tony/Documents/cv.pdf")</code> (Android)。</p>
is_path_absolute	<p><code>is_path_absolute(1) :</code></p> <p><code>is_path_absolute(path)</code>用于判断位于字符串 <code>path</code> 是否是一个绝对路径（也就是从根目录开始而不是相对于当前目录的路径）。如果是返回 <code>true</code>, 否则返回 <code>false</code>。例子包括 <code>is_path_absolute("E:\\temp")</code> (Windows)和 <code>is_path_absolute("Documents/cv.pdf")</code> (Android)。</p>

is_path_parent	<p>is_path_parent(2) :</p> <p>is_path_parent(path1, path2)用于判断位于字符串 path1 是否是字符串 path2 的上级目录。如果是返回 true, 否则返回 false。例子包括 is_path_parent("E:\\temp", "E:\\temp\\..\\temp\\test") (Windows)和 is_path_parent(".", "Documents/cv.pdf") (Android)。</p>
is_path_same	<p>is_path_same(2) :</p> <p>is_path_same(path1, path2)用于判断位于字符串 path1 是否和字符串 path2 指向同一条路径。如果是返回 true, 否则返回 false。例子包括 is_path_same("E:\\temp", "E:\\temp\\..\\temp\\") (Windows)和 is_path_parent("/home/tony/Documents", "Documents/") (Android)。</p>
is_symbol_link	<p>is_symbol_link(1) :</p> <p>is_symbol_link(path)用于判断位于字符串 path 的文件（或者目录）是否是一个符号链接。如果该文件或目录存在并且是一个符号链接返回 true, 否则返回 false。例子包括 is_symbol_link("E:\\") (Windows)和 is_symbol_link("/home/tony/Documents/cv.pdf") (Android)。</p>

`set_file_last_modified_time`

`set_file_last_modified_time(2) :`

`set_file_last_modified_time(path, time)`设置基于字符串路径的 path 的文件或目录的上一次更改时间为 time。该时间等于从 1970 年 1 月 1 日午夜开始到上一次更改时刻所经历的毫秒数。如果 path 不存在或者没有访问权限，返回 false，否则返回 true。例子包括("C:\\Temp\\Hello\\", 99999999) (Windows) 和 `set_file_last_modified_time("./hello.txt", 1111111111)` (Android)。

上述函数均只使用最多 2 个参数，函数用法非常简单，以 `get_` 打头的函数，均只有一个参数，该参数要么是 `fopen` 函数返回的整数类型文件号，要么是基于字符串的文件路径，而这些函数的返回值，要么是一个一个基于字符串的文件路径或路径分隔符，要么是一个整数代表文件的访问时间或者文件的大小。

以 `is_` 打头的函数，均是对文件或者路径属性的判断，返回值均为布尔值 True 或者 False。和 `get_` 打头的函数一样，这些函数的参数要么是 `fopen` 函数返回的整数类型文件号，要么是基于字符串的文件路径。

以 `set_` 打头的函数，这里仅有 `set_file_last_modified_time`。该函数是唯一用于修改文件属性的函数，该函数用于设置文件的最近修改时间。该函数的第一个参数为文件路径，第二个参数为一个整数，表示需要设置的时间。

以下给出了上述函数的使用的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 io and file libs 子目录中的 examples.mfps 文件中找到。

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()
```

```
//假设当前的工作目录为 AnMath 目录（安卓下）或者
```

```
//JCcmdLine.jar 所在目录（基于 JAVA 的可编程科学计算器）
```

```
// Assume current working directory is AnMath in Android
```

```
// or the folder where JCcmdline.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
```

```
// 打开本函数所在的源代码文件
```

```
variable strPath = "scripts/manual/io_and_file_libs/examples.mfps"
```

```
variable fd = fopen(strPath, "r")
```

```
[
```

```
// get source code file's absolute path
```

```
// 获得源代码文件的绝对路径
```

```
retval = get_absolute_path(fd)
```

```
print("Current source file's absolute path is " + retval + "\n")
```

```
[
```

```
fclose(fd)
```

```
[
```

```
// get source code file's canonical path  
  
// 获得源代码文件不包括符号链接的绝对路径  
  
retval = get_canonical_path(strPath)  
  
print("Current source file's canonical path is " + retval + "\n")  
  
//  
  
// get source code file's last modified time  
  
// 获得源代码文件上一次修改时间  
  
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  
  
// 将源代码文件上一次修改时间设置为 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  
  
// 获得源代码文件尺寸  
  
retval = get_file_size(strPath)  
  
print("Current source file's size is " + retval + "\n")  
  
//
```

```
// 源代码文件是一个目录吗?  
  
retval = is_directory(strPath)  
  
print("Is current source file a directory: " + retval + "\n")  
  
//  
  
// is source code file executable?  
  
// 源代码文件可执行吗?  
  
retval = is_file_executable(strPath)  
  
print("Is current source file executable: " + retval + "\n")  
  
//  
  
// is source code file existing?  
  
// 源代码文件存在吗?  
  
retval = is_file_existing(strPath)  
  
print("Is current source file existing: " + retval + "\n")  
  
//  
  
// is source code file hidden?  
  
// 源代码文件是隐藏的吗?  
  
retval = is_file_hidden(strPath)  
  
print("Is current source file hidden: " + retval + "\n")  
  
//  
  
// is source code file normal?  
  
// 源代码文件是常规文件吗?  
  
retval = is_file_normal(strPath)  
  
print("Is current source file normal: " + retval + "\n")
```

```
// is source code file readable?  
// 源代码文件可读吗?  
  
retval = is_file_readable(strPath)  
  
print("Is current source file readable: " + retval + "\n")  
  
//  
  
// is source code file writable?  
// 源代码文件可写吗?  
  
retval = is_file_writable(strPath)  
  
print("Is current source file writable: " + retval + "\n")  
  
//  
  
// is source code file path absolute?  
// 源代码文件路径是绝对路径吗?  
  
retval = is_path_absolute(strPath)  
  
print("Is current source file path absolute: " _  
     + retval + "\n")  
  
//  
  
// is source code file path a symbol link?  
// 源代码文件路径是符号链接吗?  
  
retval = is_symbol_link(strPath)  
  
print("Is current source file path symbol link: " _  
     + retval + "\n")
```

```

    // is path1 the parent of source code file path?

    // Path1 是源代码路径的上层路径吗?

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?

    // Path2 和源代码路径是同一路径吗?

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")

endif

```

上述函数的运行结果如下。需要注意的是，运行上述函数时如果用户的当前目录不是 AnMath 目录（如果是运行于安卓系统）或者 JCcmdLine.jar/mfplang.cmd/mfplang.sh 所在目录（如果运行基于 JAVA 的可编程科学计算器），将会得到完全不同的结果，程序可能无法正常运行。

```

Current working directory is
E:\Development\workspace\AnMath\misc\marketing\JCcmdLine_runtime

```

```

Current source file's absolute path is
E:\Development\workspace\AnMath\misc\marketing\JCcmdLine_runtime\sc
ripts\manual\io and file libs\examples.mfps

```

Current source file's canonical path is
E:\Development\workspace\AnMath\misc\marketing\JCmdLine_runtime\scripts\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

第 5 节 类似 Dos 和 Unix 命令的文件整体操作函数

以上对文件的内容读些和对文件的属性的操作函数，只是对单一的文件进行操作，并且不能够删除文件。如果用户想查看文件夹中所有的文件列表，或者想移动一个文件夹。很显然，上述函数无法满足要求。

如果用户使用 Dos 或者 Unix 的命令提示符，可以通过调用操作系统提供的指令对文件的整体进行操作，比如 pwd 指令获取当前目录，cd 指令更改当前目录，xcopy 或者 cp 指令拷贝文件等。为了方便用户，MFP 编程语言也提供了类似的函数进行文件的整体操作。函数列表如下：

函数名	函数帮助信息
cd	cd(1) : change_dir(path) (别名 cd(path)) 将当前路径变为字符串路径 path。如果成功，返回 true，否则返回 false。例子包括 change_dir("D:\\Windows") (Windows) 和 cd("/") (Android)。
change_dir	change_dir(1) : change_dir(path) (别名 cd(path)) 将当前路径变为字符串路径 path。如果成功，返回 true，否则返回 false。例子包括 change_dir("D:\\Windows") (Windows) 和 cd("/") (Android)。
copy_file	copy_file(3) : copy_file(source, destination, replace_exist) 函数拷贝位于字符串 source 路径的文件或文件夹到位于字符串 destination 路径的文件或文件夹。如果第三个参数，replace_exist，是 true，那么如果目标文件已经存在，它将会被源文件（或者源文件夹中的对应文件）替换。注意第三个参数可以省略，它的缺省值为 false。例子包括 copy_file("c:\\temp\\try1", "D:\\", true) (Windows) 和 copy_file("/mnt/sdcard/testfile.txt", "./testfile_copy.txt") (Android)。

create_file	<p><code>create_file(2) :</code></p> <p><code>create_file(path, is_folder)</code> 创建一个文件（如果 <code>is_folder</code> 是 <code>false</code> 或者不存在）或者目录（如果 <code>if_folder</code> 是 <code>true</code>）。如果这个基于字符串的路径 <code>path</code> 的上级目录不存在，不存在的上级目录将会被创建。如果文件能够被创建，这个函数返回 <code>true</code>，否则返回 <code>false</code>。例子包括 <code>create_file("c:\\temp\\try1", true)</code> (Windows) 和 <code>create_file("testfile_copy.txt")</code> (Android)。</p>
delete_file	<p><code>delete_file(2) :</code></p> <p><code>delete_file(path, delete_children_in_folder)</code> 删除一个位于字符串 <code>path</code> 的文件或者目录。如果是一个目录且第二个参数 <code>delete_children_in_folder</code> 是 <code>true</code>，目录中的所以文件和子目录将会被删除。注意第二个参数可以省略，它的缺省值是 <code>false</code>。如果删除成功，本函数返回 <code>true</code>，否则返回 <code>false</code>。例子包括 <code>delete_file("c:\\temp\\try1", true)</code> (Windows) 和 <code>delete_file("testfile_copy.txt")</code> (Android)。</p>
dir	<p><code>dir(1) :</code></p> <p><code>print_file_list(path)</code> (别名 <code>ls(path)</code> 或者 <code>dir(path)</code>) 函数和 Windows 平台上的 <code>dir</code> 命令以及 Linux 平台上的 <code>ls</code> 命令类似。它打印出位于字符串 <code>path</code> 路径的文件或者目录中的所有子文件和子目录的信息。它返回打印的条目的个数。如果不存在一个文件或者目录对应于 <code>path</code> 路径，它返回 1。注意参数 <code>path</code> 是可以省略的。它的缺省值是当前目录。例子包括 <code>dir()</code> (Windows) 和 <code>ls("../testfile_copy.txt")</code> (Android)。</p>
get_working_dir	<p><code>get_working_dir(0) :</code></p> <p><code>get_working_dir()</code> (别名 <code>pwd()</code>) 返回基于字符串的当前路径。</p>
list_files	<p><code>list_files(1) :</code></p> <p><code>list_files(path)</code> 返回位于字符串 <code>path</code> 路径的目录中的所有</p>

	子文件或者子目录的名字，或者如果 path 路径对应的是一个文件，它返回该文件的文件名。如果不存在一个文件或者目录对应于 path 路径，它返回 NULL。注意参数 path 是可以省略的。它的缺省值是当前目录。例子包括 list_files("c:\\temp\\try1") (Windows) 和 list_files("../testfile_copy.txt") (Android)。
ls	<p>ls(1) :</p> <p>print_file_list(path) (别名 ls(path) 或者 dir(path)) 函数和 Windows 平台上的 dir 命令以及 Linux 平台上的 ls 命令类似。它打印出位于字符串 path 路径的文件或者目录中的所有子文件和子目录的信息。它返回打印的条目的个数。如果不存在一个文件或者目录对应于 path 路径，它返回 1。注意参数 path 是可以省略的。它的缺省值是当前目录。例子包括 dir() (Windows) 和 ls("../testfile_copy.txt") (Android)。</p>
move_file	<p>move_file(3) :</p> <p>move_file(source, destination, replace_exist) 函数移动位于字符串 source 路径的文件或文件夹到位于字符串 destination 路径的文件或位于 destination 路径的文件夹内（而不是位于 destination 路径的文件夹本身）。如果第三个参数，replace_exist，是 true，那么如果目标文件已经存在，它将会被源文件（或者源文件夹中的对应文件）替换。注意第三个参数可以省略，它的缺省值为 false。例子包括 move_file("c:\\temp\\try1", "D:\\", true) (Windows) 和 copy_file("/mnt/sdcard/testfile.txt", "./testfile_copy.txt") (Android)。</p>
print_file_list	<p>print_file_list(1) :</p> <p>print_file_list(path) (别名 ls(path) 或者 dir(path)) 函数和 Windows 平台上的 dir 命令以及 Linux 平台上的 ls 命令类似。它打印出位于字符串 path 路径的文件或者目录中的所有子文件和子目录的信息。它返回打印的条目的个数。如果不存在一个文件或者目录对应于 path 路径，它返回 -</p>

	1。注意参数 path 是可以省略的。它的缺省值是当前目录。例子包括 dir() (Windows) 和 ls("../testfile_copy.txt") (Android)。
pwd	<p>pwd(0) :</p> <p>get_working_dir() (别名 pwd()) 返回基于字符串的当前路径。</p>

在这些函数中，cd 和 change_dir，pwd 和 get_working_dir，以及 ls，dir 和 print_file_list 是三组名字不同但是功能和用法完全相同的函数。以上的所有函数和 Dos 以及 Unix 下的文件操作指令对应关系如下（函数的用法和功能不见得和 Dos 以及 Unix 下的文件操作指令完全一样，但是很接近）：

1. cd 和 change_dir: 对应 Dos 和 Unix 中的 cd 指令；
2. pwd 和 get_working_dir: 对应 Dos 和 Unix 中的 pwd 指令；
3. ls, dir 和 print_file_list: 对应 Dos 下的 dir 和 Unix 下的 ls 指令；
4. copy_file: 对应 Dos 下的 xcopy 和 Unix 下的 cp 指令；
5. move_file: 对应 Dos 下的 move 和 Unix 下的 mv 指令；
6. delete_file: 对应 Dos 下的 del 和 Unix 下的 rm 指令；

此外，还有一个 list_files 函数，该函数也是返回一个目录中的所有文件的文件名。和 dir (或者 ls 或者 print_file_list) 函数不同之处在于，list_files 函数返回的是一个数组，数组中的每一个元素是一个代表目录中的一个文件的文件名的字符串，dir 函数，则是将目录中所有的文件的名字，修改时间，以及其他属性都打印输出到命令提示符上。Dir 函数在用户把命令提示符当作 Unix 终端或者 Dos 窗口使用时能够提供更加详细的信息，但是 list_files 在编写程序时更有用。

以下给出了上述函数的使用的例子。本示例的代码可以在本手册自带的示例代码所在目录中的 io and file libs 子目录中的 examples.mfps 文件中找到。

Help

@language:

test file operation commands

@end

@language:simplified_chinese

测试文件整体操作函数

@end

endh

function fileopr()

//假设当前的工作目录为 AnMath 目录（安卓下）或者

//JCcmdLine. jar 所在目录（基于 JAVA 的可编程科学计算器）

// Assume current working directory is AnMath in Android

// or the folder where JCcmdline. 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/

//改变目录至 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

```
// 现在显示当前目录下的内容  
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  
// 在调用 create_file 之后显示当前目录下的内容  
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  
// 在调用 move_file 之后显示当前目录下的内容  
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

// 在调用 delete_file 之后显示当前目录下的内容

print("After delete folder test_folder1, run dir:\n")

dir(pwd())



// return to original working directory

// 返回原来的工作目录。

cd(strOriginalPWD)

endif

```

上述函数的运行结果如下，需要注意的是，运行上述函数时如果用户用户的当前目录不是 AnMath 目录（如果是运行于安卓系统）或者 JCcmdLine. jar/mfplang. cmd/mfplang. sh 所在目录（如果运行基于 JAVA 的可编程科学计算器），将会得到完全不同的结果，程序可能无法正常运行：

Current directory is
E:\Development\workspace\AnMath\misc\marketing\JCcmdLine_runtime

After cd, current directory is
E:\Development\workspace\AnMath\misc\marketing\JCcmdLine_runtime\scripts\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			

第6节 进行复杂文件操作示例

上面的章节中介绍了 MFP 编程语言所提供的全套文件操作函数。事实上，上述函数的功能非常全面和强大，用户完全可以通过它们实现任何文件操作。以下就是调用上述函数进行复杂文件操作的一个实例。

在可编程科学计算器中，`set_array_elem` 函数（具体用法参见第 3 章第 3 节的第 3 部分）会对参数数组的某一个元素或者子元素（如果该元素也是一个矩阵的话）进行赋值，并返回新的数组，原来的参数数组，可能会发生改变，也可能不会发生改变，但即便发生改变，改编后的参数数组的

值，也不见得就和返回值相同，所以，用户必须通过 `set_array_elem` 函数的返回值来得到新的数组。换句话说，`set_array_elem` 函数的正确调用方式为：

```
array_to_change = set_array_elem(array_to_change, index, value)
```

。但是，在老版本（1.6.6 之前的版本）的可编程科学计算器中，`set_array_elem` 函数会将参数矩阵的值直接改变为新值，换句话说，用户不必将 `set_array_elem` 的返回值赋给 `array_to_change` 一样可以得到改编后的矩阵值，所以，很多用户调用 `set_array_elem` 函数的方式为

```
set_array_elem(array_to_change, index, value)
```

，换句话说，他们根本就不使用 `set_array_elem` 的返回值。而这种调用办法在可编程科学计算器 1.7 及其以后的版本中将不再被支持。为了保证所有的代码可以正确运行，用户必须对老的调用方法进行修改。第一种修改办法是手动修改，也就是搜索所有代码源文件中的 `set_array_elem` 调用，然后一行一行地把旧的调用办法改为新的调用办法，显然，这样做费时费力，而且还容易出错。

第二种修改办法是使用 MFP 编程，直接在源文件中自动搜索 `set_array_elem` 调用并对其自动修改，这样做速度快，并且不会出错。

用户可能会问，在 MFP 函数运行的过程中对源文件进行修改，会不会对程序的运行产生影响？答案是不会，因为，在现阶段的可编程科学计算器启动的时候，所有的代码已经被读入，除非用户重启可编程科学计算器，或者在安卓上使用可编程科学计算器所提供的代码编辑器修改并保存代码，或者在个人电脑上把输入焦点切换回基于 JAVA 的图形界面可编程科学计算器，否则代码不会被重新读入，所以，调用 MFP 程序修改自己的代码是没有关系的。当然，现在没有关系并不代表将来没有问题，未来可编程科学计算器可能会动态地装载代码，届时，修改运行中的代码就可能会有影响了。

使用 MFP 编程修改文件，关键要实现几点：

1. 代码需要能够对 `scripts` 目录下（包括其子目录下）的所有 `.mfps` 文件进行遍历搜索；

2. 代码能通过对文本文件的每一行进行匹配，找到旧的 set_array_elem 调用位置；
3. 代码能够将文本文件的某些行进行修改并保存。

如果要对某个目录下的所有文件（包括子目录下的所有文件）进行遍历搜索，就需要用到上面提到的 list_files 函数和 is_directory 函数以及 is_file_normal 函数。List_files 函数列出指定目录下的所有文件和目录，然后一遍历这些文件和目录，用 is_directory 函数判断是不是一个子目录，如果是，再回到上一步用 list_files 进行操作，如果不是，用 is_file_normal 函数判断是不是普通的文件，同时还要确定文件名最后 5 个字符是不是.mfps，如果所有的条件都满足，则是一个.mfps 代码源文件。

实现上述流程，一个办法是用迭代函数，每一次发现一个新的目录，就调用迭代函数对该目录下的所有内容进行遍历搜索；另外一个办法是，创建一个包括所有目录的数组，每发现一个新的目录，就把该目录的路径添加的数组的尾部，直到该数组不再增长，并且数组中所有的目录都已经被搜索。下面的代码片断采用的是第二种办法：

```
// all_mfps_files 是一个 1 维数组，每个元素是一个.mfps 源文件路径  
Variable all_mfps_files = []  
  
// all_folders 是一个 1 维数组，每个元素是初始目录本身  
// （也就是 strScriptsPath）或者其下一个子目录的路径。  
Variable all_folders = [strScriptsPath]  
  
Variable folder_idx = 0  
  
// 遍历 all_folders 数组，注意在遍历过程中，all_folders 还在增长  
While(folder_idx < size(all_folders)[0])  
  
    // 列出一个 folder 内的所有文件  
  
    Variable these_files = list_files(all_folders[folder_idx])
```

```

// 遍历这些文件。注意 these_files 是一个一维数组，所以

// size(these_files)[0]必然是一个等于该数组长度的正整数，

// 此外还要注意数组的索引是从 0 开始到数组长度-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))

        //如果这个文件名对应的文件实际上是一个目录，将其

        //添加到 all_folders 数组中。

        All_folders = set_array_elem(all_folders, _

            size(all_folders), _

            this_file_name)

    Elseif and(strcmp(strsub(this_file_name, _

                strlen(this_file_name)-5), ".mfp") ==0, _

                is_file_normal(this_file_name))

        //如果这个名称对应的文件确实为以.mfp 为后缀的源文件，

        //将其添加到 all_mfp_files 数组中。

        all_mfps_files = set_array_elem(all_mfps_files, _

            size(all_mfps_files), _

```

```
        this_file_name)

Endif

Next

folder_idx = folder_idx + 1

Loop
```

找到了每一个 mfps 源文件后，就需要读取文件的每一行，看该行的内容是否符合 set_array_elem 函数的旧的调用方式，注意到 set_array_elem 函数的旧的调用方式意味着去除该行的头尾空格之后，set_array_elem 永远打头，然后是括号（括号和 set_array_elem 之间可能会有空格），然后是待修改的数组变量名称（和括号之间可能会有空格），然后是逗号（和待修改的数组变量名称之间可能会有空格）。对于这种固定的模式，可以使用 split 函数，对代码行从括号和逗号处分割，然后判断分割出来的第一个子字符串去除头尾空格之后是否是 set_array_elem。具体代码片断如下：

```
//假设文件已经打开，现在正在读取

Variable strLine = freadline(fd)

//注意这里用括号分割字符串是使用"\\("而不是"()", 原因在于括号

//对于 split 函数来说是一个特殊的控制字符，所以要用两个反斜杆

//规避。

Variable strarray1 = split(strline, "\\(")

//如果分割后至少有两个子字符串，并且分割后的第一个子字符串是

//set_array_elem，注意这里要调用 trim 函数去除头尾空格。

If and(size(strarray1)[0] >= 2, _

        Stricmp(trim(strarray1[0]), "set_array_elem") == 0)

//和括号不同，逗号不是 split 函数的特殊控制字符，所以不用规避
```

```
Variable strarray2 = split(strarray1[1], ",")  
  
If size(strarray2)[0] >= 2  
  
    //必须至少有两个子字符串，否则可能意味着没有逗号  
  
    //是 set_array_elem 的旧调用方式，现在进行修改  
  
    .....  
  
Endif  
  
EndIf
```

将旧的 set_array_elem 调用方式字符串改为新的 set_array_elem 调用方式字符串则非常简单，在前面加入被修改的数组变量的名字和等号即可：

```
StrLine = strarray2[0] + " = " + trim(strLine)
```

这里，strarray2[0] 是待修改的数组变量的名字（参见搜索 set_array_elem 的旧的调用方式的代码片断）。

然后把修改后的代码写入一个新的文件：

```
Fprintf(fd1, "%s\n", strLine)
```

最后调用 move_file 函数用新的 mfps 代码文件覆盖旧的 mfps 代码文件即可。完整的代码如下，本示例的代码可以在本手册自带的示例代码所在目录中的 io and file libs 子目录中的 examples.mfps 文件中找到。

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 file0pr2()

    Variable strConfirm

        // first get current working directory, should be AnMath

        // in Android or the folder where JCcmdline.jar is located

        // (for Scientific Calculator for JAVA)

        // 首先获取当前工作目录，必须位于 AnMath 目录（安卓下）或者

        // JCcmdLine.jar 所在目录（基于 JAVA 的可编程科学计算器）

    Variable strOriginalPWD = pwd()

    printf("Current directory is " + strOriginalPWD + "\n")

    // confirm it is the right working folder

    // 请用户确认是正确的工作目录

    printf("Is this AnMath folder in Android "
        + "or if you are working on Scientific Calculator for JAVA, "
        + "is the folder where JCcmdline.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

        //不是在正确的工作目录，退出

        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

// 请先备份您的源代码

Print("Please back up your source codes before run the program!!!\n")

// have you backed up your source codes, if no I cannot continue.

//您备份了您的源代码吗？如果不是 Y，我没法进行下一步。

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.

// 如果还没有备份您的源代码，我来帮您

print("If haven't been backed up, I will do it for you!\n")

// 按回车键开始备份

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.
```

```
// 现在我确信，工作目录是对的了并且代码已经被备份。准备工作完成，按回车继续

pause("Now preparation work has finished, press Enter to continue")

// all_mfps_files 是一个一维数组，每一个元素是一个 mfps 源文件路径

Variable all_mfps_files = []

// all_folders 是 also a 1D array, each element is path of a folder

// including source codes

// all_folders 是一个一维数组，每一个元素是自定目录本身或者其下的

// 一个子目录的路径。

Variable all_folders = [strScriptsPath]

Variable folder_idx = 0

// Go through all_folders, note that in the procedure all_folders is

// increasing

// 遍历 all_folders 数组，注意在遍历过程中，all_folders 还在增长

While(folder_idx < size(all_folders)[0])

    // list all the files in a folder

    // 列出一个 folder 内的所有文件

    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.

    // 遍历这些文件。注意 these_files 是一个一维数组，所以
```

```

    // size(these_files)[0]必然是一个等于该数组长度的正整数,
    // 此外还要注意数组的索引是从 0 开始到数组长度-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
            //如果这个文件名对应的文件实际上是一个目录，将其
            //添加到 all_folders 数组中。
            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
            //如果这个名称对应的文件确实为以.mfps 为后缀的源文件,
            //将其添加到 all_mfp_files 数组中。
            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 .mpfs files
```

```
// 现在 all_mfps_files 包含了所有的.mpfs 文件
```

```
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 is UTF-8 to ensure that unicode (e.g. Chinese and
```

```
// Japanese characters) is supported
```

```
// 创建一个临时源文件以写入修改后的代码。设置编码模式为 UTF-8 以确保
```

```
// 文件中的 Unicode 字符（也就是中文汉字）能够被支持。
```

```
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.
```

```

//注意这里用括号分割字符串是使用"\\"而不是"()", 原因在于括号

//对于 split 函数来说是一个特殊的控制字符, 所以要用两个反斜杆

//规避。

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

//如果分割后至少有两个子字符串, 并且分割后的第一个子字符串是

//set_array_elem, 注意这里要调用 trim 函数去除头尾空格。

If and(size(strarray1)[0] >= 2, _

Stricmp(trim(strarray1[0]), "set_array_elem") == 0)

// need not to escape "," for regex, different from "("

//和括号不同, 逗号不是 split 函数的特殊控制字符, 所以不用规避

Variable strarray2 = split(strarray1[1], ",") 

If size(strarray2)[0] >= 2

// Should have at least two sub strings, otherwise may mean

// no comma.

//必须至少有两个子字符串, 否则可能意味着没有逗号

// Now we can confirm that this line needs change.

//是 set_array_elem 的旧调用方式, 现在进行修改

print("\tset_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

    // 将 strLine 写入临时文件中

    fprintf(fd1, "%s\n", strLine)

Loop

fclose(fd1)

fclose(fd)

//move temporary file to replace all_mfps_files[idx]

//用临时源代码文件替换 all_mfps_files[idx]

move_file("temporary_src_file", all_mfps_files[idx], true)

Next

//Done! (搞定了! )

printf("Done!\n")

endif

```

程序的大致流程解释如下：

第一步是判断运行上述代码是用户的工作目录是否正确，当前工作目录，必须位于 AnMath 目录（安卓下）或者 JCmdLine.jar 所在目录（基于 JAVA 的可编程科学计算器）。如果不正确，可能根本就找不到文本文件；

第二步是提示用户需要做备份，如果还没有做备份，将会自动把用户源文件存储文件夹，也就是 scripts 目录，拷贝为 scripts.bakup 目录；

第三步是寻找所有的源代码文件；

第四步是寻找 set_array_elem 函数的旧的调用办法；

第五步是修改并存储临时的修改文件；

第六部是将临时提供的修改文件拷贝覆盖可编程科学计算器中旧的源文件。

程序运行的结果如下。注意，上述函数的最终输出和用户运行时间以及代码文件的不同而变化，每一个用户看到的可能都是不同的输出，但最终，程序运行完毕之后用户将会发现所有的 set_array_elem 都是采用新型的调用方法了。

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\JCmdLine_runtime\scripts\userdef_lib\cv100plus 测试.mfps

Now analyse

C:\Users\tonyc\Development\workspace\AnMath\misc\marketing\JCmdLine_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)
```

.....

小结

输入输出和文件操作是 MFP 高级编程的一个重要部分。MFP 编程语言提供了一整套类似 C 语言的命令提示符终端，字符串和文件的输入输出函数。此外，MFP 编程语言还提供了类似 Dos 和 Unix 命令的文件操作函数用于改变当前目录，察看目录内容，复制、移动和删除文件。通过这一系列的函数，只要用户有权限，完全可以访问到操作系统的任何一个角落，完成很多其它应用无法实现的功能。

当然，在安卓系统中进行文件操作也是一件很危险的事情，由于文件对于用户已经被隐藏，程序运行完成之后，无法很方便地察看结果，也无法快速确认哪些文件被修改，修改是否符合预期。在这种情况下，建议用户在可编程科学计算器的工作目录 AnMath 下专门建立一个用于文件操作的目录，把所操作的文件限定在此目录中。

第7章 日期和时间以及系统函数

对日期和时间的读写是任何编程语言不可缺少的部分，此外，MFP 编程语言还提供了一些系统函数以方便用户直接使用操作系统的一些功能。

第1节 日期和时间的函数

MFP 编程语言所提供的日期和时间的函数参见下表：

函数名	函数帮助信息
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) 和 get_millisecond(timestamp) 分别返回时标 timestamp 所对应的年，月，本年中的第几天，本月中的第几天，本星期中的第几天（礼拜天是第 0 天，礼拜一是第一天，...），小时，分钟，秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差。例如，get_day_of_week(get_time_stamp(2014, 12, 21)) 返回 0，表示 2014 年 12 月 21 日是礼拜天。
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) 和 get_millisecond(timestamp) 分别返回时标 timestamp 所对应的年，月，本年中的第几天，本月中的第几天，本星期中的第几天（礼拜天是第 0 天，礼拜一是第一天，...），小时，分钟，秒钟和毫秒。时标是时标所表

	示的时刻和 1970 年 1 月 1 日午夜 (UTC) 的毫秒数时间差。例如, <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0, 表示 2014 年 12 月 21 日是礼拜天。
<code>get_day_of_year</code>	<p><code>get_day_of_year(1)</code>:</p> <p><code>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)</code> 和 <code>get_millisecond(timestamp)</code> 分别返回时标 <code>timestamp</code> 所对应的年, 月, 本年中的第几天, 本月中的第几天, 本星期中的第几天 (礼拜天是第 0 天, 礼拜一是第一天, ...), 小时, 分钟, 秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜 (UTC) 的毫秒数时间差。例如, <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0, 表示 2014 年 12 月 21 日是礼拜天。</p>
<code>get_hour</code>	<p><code>get_hour(1)</code>:</p> <p><code>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)</code> 和 <code>get_millisecond(timestamp)</code> 分别返回时标 <code>timestamp</code> 所对应的年, 月, 本年中的第几天, 本月中的第几天, 本星期中的第几天 (礼拜天是第 0 天, 礼拜一是第一天, ...), 小时, 分钟, 秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜 (UTC) 的毫秒数时间差。例如, <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0, 表示 2014 年 12 月 21 日是礼拜天。</p>
<code>get_millisecond</code>	<p><code>get_millisecond(1)</code>:</p> <p><code>get_year(timestamp), get_month(timestamp), get_day_of_year(timestamp),</code></p>

	<code>get_day_of_month(timestamp),</code> <code>get_day_of_week(timestamp), get_hour(timestamp),</code> <code>get_minute(timestamp), get_second(timestamp)和</code> <code>get_millisecond(timestamp)</code> 分别返回时标 timestamp 所对应的年，月，本年中的第几天，本月中的第几天，本星期中的第几天（礼拜天是第 0 天，礼拜一是第一天，...），小时，分钟，秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差。例如， <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0，表示 2014 年 12 月 21 日是礼拜天。
<code>get_minute</code>	<code>get_minute(1) :</code> <code>get_year(timestamp), get_month(timestamp),</code> <code>get_day_of_year(timestamp),</code> <code>get_day_of_month(timestamp),</code> <code>get_day_of_week(timestamp), get_hour(timestamp),</code> <code>get_minute(timestamp), get_second(timestamp)和</code> <code>get_millisecond(timestamp)</code> 分别返回时标 timestamp 所对应的年，月，本年中的第几天，本月中的第几天，本星期中的第几天（礼拜天是第 0 天，礼拜一是第一天，...），小时，分钟，秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差。例如， <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0，表示 2014 年 12 月 21 日是礼拜天。
<code>get_month</code>	<code>get_month(1) :</code> <code>get_year(timestamp), get_month(timestamp),</code> <code>get_day_of_year(timestamp),</code> <code>get_day_of_month(timestamp),</code> <code>get_day_of_week(timestamp), get_hour(timestamp),</code> <code>get_minute(timestamp), get_second(timestamp)和</code> <code>get_millisecond(timestamp)</code> 分别返回时标 timestamp 所对应的年，月，本年中的第几天，本月中的第几天，本星期中的第几天（礼拜天是第 0 天，礼拜一是第一天，...），小时，分钟，秒钟和毫秒。时标是时标所表

	示的时刻和 1970 年 1 月 1 日午夜 (UTC) 的毫秒数时间差。例如, <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0, 表示 2014 年 12 月 21 日是礼拜天。
<code>get_second</code>	<p><code>get_second(1)</code>:</p> <pre>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) 和 get_millisecond(timestamp) 分别返回时标 timestamp 所对应的年, 月, 本年中的第几天, 本月中的第几天, 本星期中的第几天 (礼拜天是第 0 天, 礼拜一是第一天, ...), 小时, 分钟, 秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜 (UTC) 的毫秒数时间差。例如, <code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0, 表示 2014 年 12 月 21 日是礼拜天。</pre>
<code>get_time_stamp</code>	<p><code>get_time_stamp(1...)</code>:</p> <p><code>get_time_stamp(string_or_year, ...)</code> 返回由其参数所决定的时标。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜 (UTC) 的毫秒数时间差。这个函数有两种工作模式。第一种模式是 <code>get_time_stamp(string_time_stamp)</code>。这种模式仅仅接受一个字符串参数, 该参数必须基于 <code>yyyy-mm-dd hh:mm:ss[.f...]</code> 的格式。其中, 秒的分数部分可以忽略。第二种模式是 <code>get_time_stamp(year, month, day, hour, minute, second, millisecond)</code>。这些参数中, 除了第一个参数 <code>year</code> (年), 所有的其他参数都可以省略。如果省略, <code>month</code> (月) 和 <code>day</code> (日) 的缺省值是 1, <code>hour</code> (小时), <code>minute</code> (分钟), <code>second</code> (秒) 和 <code>millisecond</code> (毫秒) 的缺省值是 0。比如 <code>get_time_stamp("1981-05-30 17:05:06")</code> 返回 1981 年 5 月 30 日 17 点 5 分 6 秒 0 毫秒的时标, 用户也可以调用 <code>get_time_stamp(1981, 5, 30, 17, 5, 6, 0)</code> 获得同样的结果。</p>

get_year	<p><code>get_year(1) :</code></p> <p><code>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)和 get_millisecond(timestamp)</code> 分别返回时标 timestamp 所对应的年，月，本年中的第几天，本月中的第几天，本星期中的第几天（礼拜天是第 0 天，礼拜一是第一天，...），小时，分钟，秒钟和毫秒。时标是时标所表示的时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差。例如，<code>get_day_of_week(get_time_stamp(2014, 12, 21))</code> 返回 0，表示 2014 年 12 月 21 日是礼拜天。</p>
now	<p><code>now(0) :</code></p> <p><code>now()</code> 返回当前时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差。</p>

上述函数的用法均比较简单，其中，`get_day_of_month`，`get_day_of_week`，`get_day_of_year`，`get_hour`，`get_millisecond`，`get_minute`，`get_month`，`get_second` 和 `get_year` 均为将一个时标（时标是时标所表示的时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差）转换为人能够看得懂的时间，而 `get_time_stamp` 是将一个供人读取的时间转换为时标，`now` 函数则是返回当前时间所对应的时标。

以下是上述函数的一个例子。本示例的代码可以在本手册自带的示例代码所在目录中的 `time date and sys libs` 子目录中的 `examples.mfps` 文件中找到。

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
```

```
//测试将一个不合法的时间表达式转换为时标，
```

```
//在不同的操作系统上会有不同的结果。
```

```
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
```

```
//测试 now 函数
```

```
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
//测试时标转换函数
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

//测试转化一个非法的时间，在不同的操作系统上会有不同的结果。

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

endif

```

上述代码的运行结果如下：

```

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
```

这里需要注意一点，如果 `get_time_stamp` 接受的参数不是一个合法的时间表达式，比如 "1980-12-71 00:00:00.0"（日期不合法），那么 `get_time_stamp` 的行为是不确定的，在有的操作系统上会出错，在有的操作系统上却能够返回一个时标。所以，用户在调用这些函数时，要尽可能地保证参数的正确性。

第 2 节 系统相关函数

MFP 编程语言提供的和操作系统调用相关的函数包括两个，一个是 `sleep`，另一个是 `system`。

`Sleep` 函数用于暂停当前正在运行中的函数一段固定的时间然后再继续。`Sleep` 函数的调用方法为 `sleep(x)`，这里 `x` 是一个正实数，表示程序暂停的毫秒数。此函数不返回任何值。

`System` 函数则用于运行一个系统指令并且返回该系统指令的返回值。它有两种用法，第一种用法，也是 1.6.6 版和更老的版本的唯一的用法，参数是一个字符串，该字符串就是所要执行的系统指令，注意系统指令必须是一个可执行的文件以及它的命令参数。由于这个原因，在 Windows 平台上运行基于 JAVA 的可编程科学计算器，诸如 `system("dir")` 是无法正确执行的，因为 `dir` 并非是一个单独的可执行文件，而是 `cmd.exe` 的一个内部功能。要执行 `dir` 指令，用户需要运行 `system("cmd /c dir")`；类似地，如果在 Linux 平台上运行基于 JAVA 的可编程科学计算器，要执行 `ls` 指令，用户需要运行 `system("sh -c ls")`。

`system` 函数的第一种用法可以在 JAVA 平台上运行，但是在安卓平台上，如果执行比如 `system("sh -c ls")`，就会出错，从 1.6.7 版开始，`system` 函数提供另外一种用法，它还是接受一个参数，但这个参数必须是一个字符串数组，数组的每一个元素是系统指令的一部分。比如，要执行 "`sh -c ls`"，可以调用 `system(["sh", "-c", "ls"])`。而要执行将文件 `file1` 改名为 `file2`，需要调用 `system(["sh", "-c", "mv file1 file2"])`，原因是 "`mv file1 file2`" 是 `sh` 的一个内部指令，它不能被人为地分割为几个部分。

还要注意，在现阶段，这个函数只能将系统指令的输出打印出来，还无法接受在运行中用户对于该系统指令的输入。此外，如果该系统指令不存在，则会抛出异常。

需要指出的是，`system` 函数在基于 JAVA 的可编程科学计算器上工作是没有问题的，但是在安卓系统上，功能受到很多限制，原因在于，安卓系统并不保证提供完整的 `sh (shell)` 调用，所以，有些命令比如 `system(["sh", "-c", "echo hello"])` 可能在一些手机上可以执行，在另外一些手机上会运行出错。要想在安卓上执行文件复制移动删除以及目录的转换等操作，建议用户调用第 6 章第 5 节中提供的对文件整体操作的函数而最好不要去运行比如 `system(["sh", "-c", "cp file1 file2"])`。

那么，是不是 `system` 函数在安卓上就没有什么很大的作用了呢？也不是，`system` 函数可以调用安卓内部的应用管理器以打开某个应用，前提条件是用户知道该应用的包 id（package id）和主界面（main activity）的名字，具体的用法为：

```
system("am start -n 包 ID/启动主界面的名字")
```

比如，如果用户的手机上安装有智慧拍照计算器（可编程科学计算器的姊妹产品），启动智慧拍照计算器的命令为：

```
system("am start -n  
com.cyzapps.SmartMath/com.cyzapps.SmartMath.ActivitySmartCalc")
```

这里 `com.cyzapps.SmartMath` 是智慧拍照计算器的包 ID，`com.cyzapps.SmartMath.ActivitySmartCalc` 是启动主界面的名字。

可编程科学计算器具有生成 APK 应用的功能，这些应用的包 ID 是由用户指定的，但是启动主界面的名字是固定的（但要注意该名字可能会在以后的版本中更改，但无论如何如果用户的 APP 已经生成，启动主界面的名字就不会变了），总是 `com.cyzapps.AnMFPApp.ActivityAnMFPMain`。所以，用户可以在 MFP 程序中启动自己的 MFP 应用。

安卓内部的应用管理器也具有终止某个进程的功能，但是，这需要具有相应的权限。为了安全起见，可编程科学计算器没有添加相应的权限，所以，`System` 函数不能用于终止某个应用。

以下是上述函数的一个例子。本示例的代码可以在本手册自带的示例代码所在目录中的 time date and sys libs 子目录中的 examples.mfps 文件中找到。

```
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

    //如果您已经安装了智慧拍照计算器软件，下述语句将启动它

    pause("Now try to start Smart Photographic Calculator. Press Enter to
continue")

    system("am start -n
com.cyzapps.SmartMath/com.cyzapps.SmartMath.ActivitySmartCalc")

endf
```

运行上述代码，用户先会看到一条打印输出为 Now sleep 3 seconds，然后程序停顿 3 秒，又打印输出 Now wake up!。最后，程序尝试启动智慧拍照计算器，如果不成功，则打印出错误信息。

小结

日期时间函数是进行编程的不可缺少的部分。在所有的操作系统中，对时间的描述是基于时标，也就是所表示的时刻和 1970 年 1 月 1 日午夜（UTC）的毫秒数时间差，MFP 编程语言提供了一组函数用于时标和人可读的时间信息之间的相互转换，大大方便了程序和用户在时间上进行交互。

MFP 编程语言还提供了一些系统相关的函数，比如 sleep 函数用于暂停正在运行的程序，system 函数用于调用一个系统指令。需要注意的是，在安卓系统上，不推荐直接调用操作系统的 shell 指令，原因在于由于对 shell 指令的实现不同，在不同的安卓上，同样的 system 命令可能得到不同的结果。System 函数在安卓系统上更多地用来启动某一个应用，如果用户知道该应用的主界面的名称的话。

第8章 用MFP编程语言进行游戏开发

从1.7.2版开始，可编程科学计算器引入了二维游戏开发引擎。使用者可以通过MFP编程语言调用该引擎所提供的函数开发二维游戏。由于MFP编程语言的跨平台性，开发出来的`mfps`游戏脚本，既可以在装有JAVA的PC上在基于JAVA的可编程科学计算器内运行，也可以在安卓上的可编程科学计算器内运行，还可以打包成为安卓系统APK安装包，作为独立的应用安装在安卓系统上，更可以直接把`mfps`游戏脚本和附件直接拷贝到安装有MFP语言解释器的PC的一个文件夹中，然后用鼠标双击直接执行，就像任何一个Windows程序一样。由于MFP编程的跨平台性，开发者可以在电脑上编写调试程序，在手机上运行，大大方便了脚本的开发。

第1节 创建，调整和关闭游戏显示窗口

用MFP语言进行游戏开发，第一件事情就是创建游戏显示窗口。在安卓系统中，游戏显示窗口就是整个屏幕，它的大小无法调整，也没有标题显示，但是，横竖屏的状态可以通过编程来控制。在PC上，窗口的大小可以调整，标题也会显示，但是没有横竖屏的概念。另外，不论在安卓上还是PC上，窗口的背景颜色，以及用户关闭窗口后窗口的行为，也就是是直接退出还是等待用户再次确认后退出，都是由程序控制。

创建游戏显示窗口的函数为`open_screen_display`。它返回一个基于屏幕的显示窗口的句柄。这里，基于屏幕的显示窗口是相对于把图像当作显示窗口而言的（参见后面的章节）。这个函数有6个参数。第一个参数是标题（基于字符串），该参数在安卓平台上不起作用。第二个参数是背景颜色，它是一个4个或3个元素的数组，如果是4个元素，就是`[Alpha, R, G, B]`，如果是3个元素，就是`[R, G, B]`，在该数组中，每一个元素的值都是从0到255。第三个参数是退出时是否需要用户确认。第四个参数是显示窗口的尺寸，它是一个包含两个元素的数组，第一个元素是宽度，第二个元素是高度，由于在安卓系统中显示窗口总是占据整个屏幕，所以在安卓系统中这个参数不起作用。第五个参数是显示窗口大小是否可以调整，同样的，这个参数仅仅在PC上才能有作用。第六个参数是显示窗口的方位，它是一个整数，也就是横屏（0）还是竖屏（1）还是任意（-1，根据安卓设备的姿势而定），这个参数仅仅在安卓上才能起作用。需要注意的是，这六个参数都是可以省略的。比如，以下语句，

```
variable display = open_screen_display("Hello world", [255, 238, 17], true, [640, 480], true, 0)
```

如果在 PC 上运行，得到的显示窗口为：

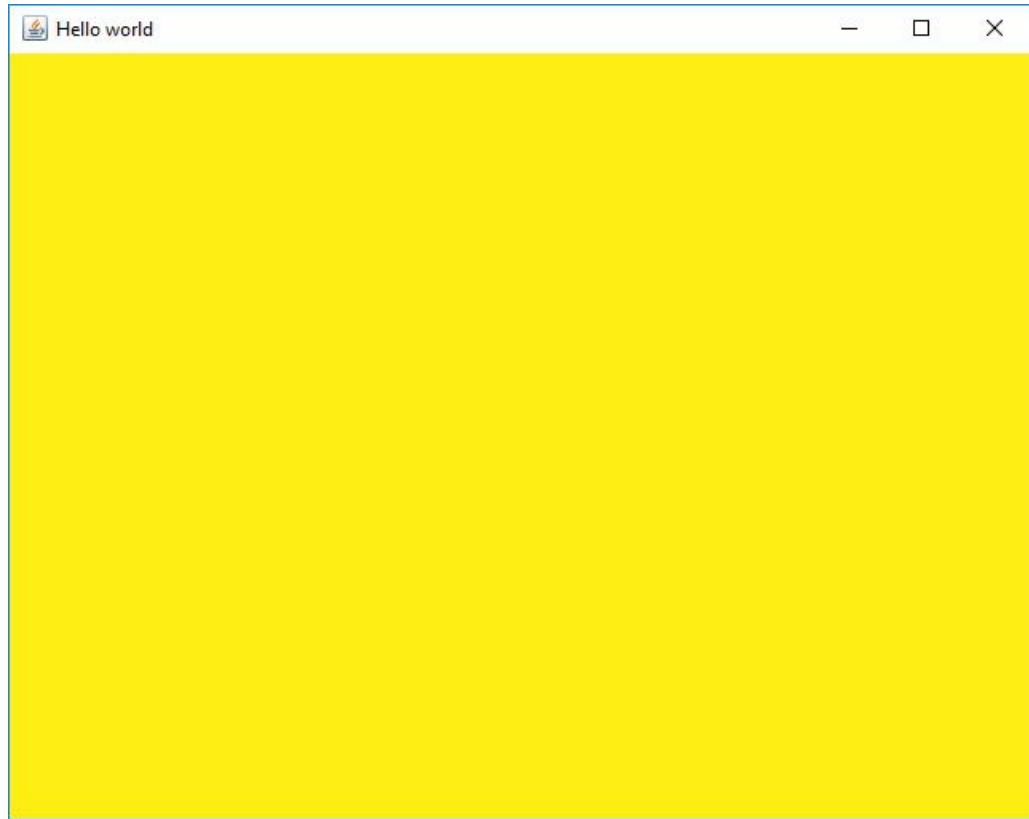


图 8.1： PC 上的显示窗口。

。而如果在安卓上运行，得到的则是一个黄色的屏幕。

如果用户想要关闭显示窗口，则需要调用 `shutdown_display` 函数。该函数有两个参数。第一个参数为显示窗口的句柄，第二个参数是一个可选参数，它是一个布尔值，`true` 表示关闭 `screen display` 时不需要用户确认，而不管在该 `display` 创建时，有无设置需要确认退出。它的缺省值是 `false`。如果想要关闭上述例子生成的 `display`，只需要执行 `shutdown_display(display, true)`，在无需用户干预的情况下，该窗口即被关闭。下图显示了，在需要用户确认的情况下显示窗口如何关闭。

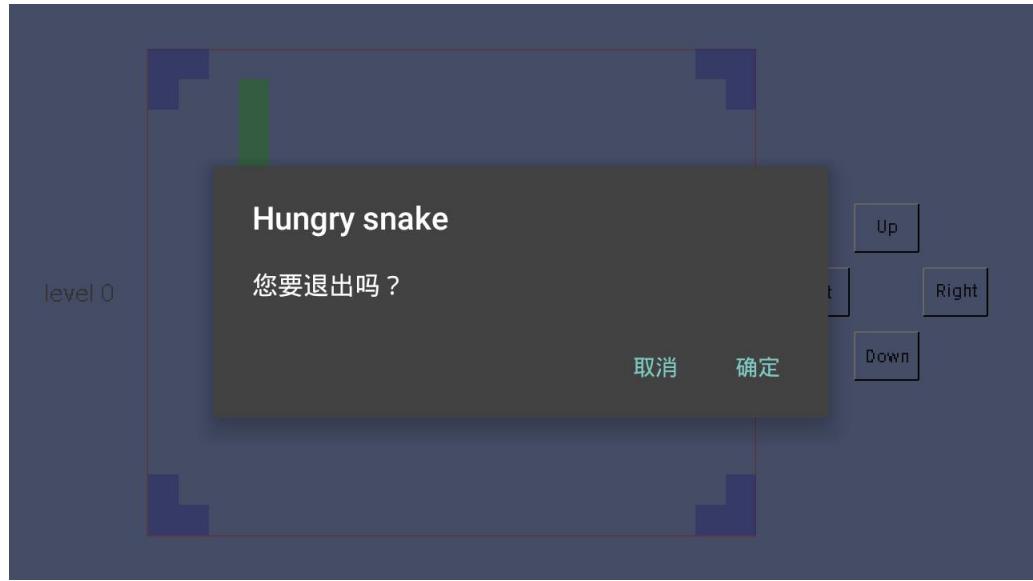


图 8.2： 用户确认关闭安卓上的显示窗口。

由于一个显示窗口可能具有标题，背景色，关闭是否需要确认，大小，大小是否可调，以及横竖屏这六大属性，MFP 提供了相应的读（get）和写（set）函数，这些函数列表如下：

函数名	函数帮助信息
get_display_caption	::mfp::graph_lib::display::get_display_caption(1) : get_display_caption(display) 在 JAVA 平台上返回一个 screen display 的标题。对于 image display，或者是对于安卓平台，get_display_caption 总是返回空字符串。
set_display_caption	::mfp::graph_lib::display::set_display_caption(2) : set_display_caption(display, caption) 在 JAVA 平台上设置一个 screen display 的标题。 set_display_resizable 对 image display 不起作用，在安卓平台上也不起作用。

get_display_bgrnd_color	::mfp::graph_lib::display::get_display_bgrnd_color(1) : get_display_bgrnd_color(display) 返回一个 display 的背景色。该 display 既可以是 screen display，也可以是 image display。背景色是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是 [Alpha, R, G, B]，如果是 3 个元素，就是 [R, G, B]，在该数组中，每一个元素的值都是从 0 到 255。
set_display_bgrnd_color	::mfp::graph_lib::display::set_display_bgrnd_color(2) : set_display_bgrnd_color(display, color) 为一个 display 设置背景色。该 display 既可以是 screen display，也可以是 image display。背景色是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是 [Alpha, R, G, B]，如果是 3 个元素，就是 [R, G, B]，在该数组中，每一个元素的值都是从 0 到 255。
get_display_confirm_close	::mfp::graph_lib::display::get_display_confirm_close(1) : get_display_confirm_close(display) 返回关闭一个 screen display 是否需要确认。对于 image display，get_display_confirm_close 总是返回 FALSE。
set_display_confirm_close	::mfp::graph_lib::display::set_display_confirm_close(2) : set_display_confirm_close(display, confirm_close_or_not) 设置关闭一个 screen display 是否需要确认。set_display_confirm_close 对 image display 不起作用。
get_display_size	::mfp::graph_lib::display::get_display_size(1)

	<pre>: get_display_size(display)返回一个 display (既可以是 screen display, 也可以是 image display) 的长度和高度。返回值是一个包含两个元素的矩阵, 第一个元素是长度, 第二个元素是高度。</pre>
set_display_size	<pre>::mfp::graph_lib::display::set_display_size(3) : set_display_size(display, width, height)设置一个 display (既可以是 screen display, 也可以是 image display) 的长度和高度分别为 width 和 height。</pre>
get_display_resizable	<pre>::mfp::graph_lib::display::get_display_resizable(1) : get_display_resizable(display)用于返回一个布尔量, 表示一个 screen display 是否可以改变大小。对于 image display, get_display_resizable 总是返回 false。</pre>
set_display_resizable	<pre>::mfp::graph_lib::display::set_display_resizable(2) : set_display_resizable(display, resizable_or_not)设置一个 screen display 是否可以改变大小。set_display_resizable 对 image display 不起作用。</pre>
get_display_orientation	<pre>::mfp::graph_lib::display::get_display_orientation(1) : get_display_orientation(display)返回一个 screen display 是横屏 (0) 还是竖屏 (1) 还是横竖均可 (-1)。对于 image display, 或者在 JAVA 平台上, get_display_orientation 总是返回 -1。</pre>

set_display_orientation	<pre>::mfp::graph_lib::display::set_display_orientation(2) :</pre> <p>set_display_orientation(display, orientation) 设置安卓平台上一个 screen display 是横屏 (orientation 等于 0) 还是竖屏 (orientation 等于 1) 还是横竖均可 (orientation 等于 -1)。 set_display_orientation 对 image display 不起作用，在 JAVA 平台上也不起作用。</p>
-------------------------	---

显示窗口还有一个重要的特性，就是它的背景图像。设置显示窗口的背景图像和布置模式的函数是 `set_display_bgrnd_image(display, image, mode)`，它有三个参数。第一个参数是显示窗口的句柄，第二个参数是背景图像，第三个参数是背景图像的布置模式（是一个整数）。注意背景图像和背景颜色不同。背景颜色弥漫在整个窗口，而背景图像是在为显示窗口绘制完背景颜色之后，根据背景图像的布置模式在窗口的某一个或多个位置绘制出一副图像。背景图像有 4 种不同的布置模式，模式 0，表示背景图像被置于显示窗口的左上角；模式 1，表示背景图像被放大或缩小到窗口大小，然后覆盖住整个窗口；模式 2，表示背景图像像贴瓷砖一样在显示窗口上重复整齐地排列，但是图像大小不变；模式 3，表示单一的背景图像置于窗口中央。

如果要读取显示窗口背景图像和布置模式，则需要分别调用 `get_display_bgrnd_image` 和 `get_display_bgrnd_image_mode` 函数。这两个函数均只接受一个参数，也就是显示窗口的句柄 `display`。第一个函数返回窗口背景图像的句柄，第二个函数返回显示窗口背景图像的布置模式。

第 2 节 在游戏显示窗口上作图

显示窗口启动之后，开发者需要在窗口上绘制和擦去图像来实现游戏的场景。MFP 二维游戏引擎提供了一系列函数以绘制/擦去不同的几何图形，绘制图像以及绘制文字

1. 游戏动画原理

在用户调用 MFP 绘制图形图像函数时，并非马上在显示窗口上绘制图像，而是将绘制事件保存在一个游戏窗口绘图调度器中，等游戏窗口需要重绘

(无论是全部重绘还是部分重绘)时, MFP 会首先将需要重新绘制的部分清除, 然后先绘制背景颜色, 再绘制背景图案, 最后将绘图调度器中的绘图事件按顺序取出, 一一绘制在窗口上面。绘图虽然包括若干步骤, 但是由于时间很短暂, 用户无法察觉具体每一步的变化, 只是看到重绘之前和之后的图像不一样了, 就好像动起来一样。

需要注意的是, 游戏窗口自己何时重绘用户是无法知道的, 如果用户需要强制游戏窗口重绘, 需要调用 `update_display` 函数, 这个函数只有一个参数, 就是游戏窗口的句柄。如上所述, 该函数会把绘图调度器中的所有绘图事件重新执行一遍。如果绘图调度器中包含太多绘图事件, 就可能花费很长时间, 用户就会感觉迟滞或者闪烁。在这种情况下, 用户可以调用 `drop_old_painting_requests` 函数, 把一些不再需要的绘图事件从绘图调度器中清除。这个函数有两个参数, 第一个参数是 `owner_info`, 用于定义移除的标准, 通过和每一个绘图事件自己的 `owner_info` 进行比较, 来决定是否需要将该绘图事件移除; 第二个参数是游戏窗口的句柄。在这个函数调用时, 首先检查绘图事件的 `owner_info` 参数(也就是该事件的拥有者和创建时标), 如果该绘图事件的拥有者和 `drop_old_painting_requests` 的拥有者完全一样, 并且创建时标早于(小于) `drop_old_painting_requests` 的创建时标, 则该事件被移除。注意, 有很多 `owner_info` 参数不包括创建时标, 这种情况下创建时标为创建时的系统时间。比如, 调用 `drop_old_painting_requests("my owner", display)`, 等价于调用 `drop_old_painting_requests(["my owner", now()], display)`。在该例子中, `drop_old_painting_requests` 从头开始检查绘图调度器中的每一个事件, 如果该事件的拥有者有名字(有的事件拥有者没有名字, 只有 `id`)并且名字是“`my owner`”, 并且该事件的创建时标早于当前时间, 则该事件被清除, 否则该事件会被保留。

2. 绘制几何图形

几何图形的绘制比较简单, 包含以下函数:

函数名	函数帮助信息
<code>draw_point</code>	<code>::mfp::graph_lib::draw::draw_point(6) :</code> <code>draw_point(owner_info, display, point_place,</code>

`color, point_style, painting_extra_info`) 为绘图事件调度器添加一个绘制点的事件。在绘图事件调度器调用这个绘制点的事件时，该事件将在 `display` 上绘制一个点。`Draw_point` 有六个参数。第一个参数是 `owner_info`。`Owner_info` 告诉绘图事件调度器谁拥有这个绘图事件。`Owner_info` 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 `id`，还可以是 `NULL`，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 `id` 的整数，或者代表系统的 `NULL`，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是任意值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 `display`，它既可以是 `screen display`，也可以是 `image display`。第三个参数是 `point_place`，也就是点的位置，他是一个包含两个元素的数组，也就是 `[x, y]`。第四个参数 `color`，代表点的颜色，它是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是 `[Alpha, R, G, B]`，如果是 3 个元素，就是 `[R, G, B]`，在该数组中，每一个元素的值都是从 0 到 255。第五个参数是 `point_style`。在现阶段它的格式是 `[point_size, point_shape]`。`Point_size` 是点的大小。它是一个正整数。`Point_shape` 是一个字符串，代表点的形状，可以取以下值：“dot”（点，注意它的大小只能是 1，`point_size` 对它不起作用），“circle”（圆圈），“square”（方块），“diamond”（正菱形），“up_triangle”（尖朝上的三角形），“down_triangle”（尖朝下的三角形），“cross”（十字形）和“x”（叉号）。这个参数是可省略的，它的缺省值是 `[1, "dot"]`。最后一个参数是 `painting_extra_info`，它告诉绘图事件调度器采用什么样的 `porterduff` 模式来绘制目标图像。这个参数也是可选参数。`porterduff` 模式内部机制比较复

	<p>杂，建议开发者省略这个参数（也就是使用参数的缺省值）。如果开发者想要详细了解 painting extra info，可以参考 set_porterduff_mode 以及 get_porterduff_mode 的函数帮助信息。如果开发者想要详细了解 porterduff 模式，建议阅读相关的 JAVA 文档。</p> <p>Draw_point 的例子包括: draw_point(["my draw", 0.381], d, [128, 45], [79, 255, 0, 142]) 以及 draw_point(NULL, d, [23, 111], [23, 178, 222], [78, "square"])。</p>
draw_line	<pre>::mfp::graph_lib::draw::draw_line(7) :</pre> <p>draw_line(owner_info, display, start_point_place, end_point_place, color, line_style, painting_extra_info) 为绘图事件调度器添加一个绘制线段的事件。在绘图事件调度器调用这个绘制线段的事件时，该事件将在 display 上绘制一条线。Draw_line 有 7 个参数。第一个参数是 owner_info。Owner_info 告诉绘图事件调度器谁拥有这个绘图事件。Owner_info 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 id，还可以是 NULL，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 id 的整数，或者代表系统的 NULL，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是任意值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 display，它既可以是 screen display，也可以是 image display。第三个参数和第四个参数是线段的起始位置 ([x1, y1]) 和终止位置 ([x2, y2])，它们均为包含两个元素的数组。第五个参数是 color，代表绘制使用的颜色，它是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是 [Alpha, R, G, B]，如果是 3 个元</p>

	<p>素，就是 [R, G, B]，在该数组中，每一个元素的值都是从 0 到 255。第六个参数是 <code>line_style</code>。在现阶段它是一个包含一个元素的数组，该元素是一个正整数，代表线的粗细。这个参数是可省略的，它的缺省值是 [1]。最后一个参数是 <code>painting_extra_info</code>，它告诉绘图事件调度器采用什么样的 porterduff 模式来绘制目标图像。这个参数也是可选参数。</p> <p>porterduff 模式内部机制比较复杂，建议开发者省略这个参数（也就是使用参数的缺省值）。如果开发者想要详细了解 <code>painting extra info</code>，可以参考 <code>set_porterduff_mode</code> 以及 <code>get_porterduff_mode</code> 的函数帮助信息。如果开发者想要详细了解 porterduff 模式，建议阅读相关的 JAVA 文档。</p> <p><code>Draw_line</code> 的例子包括： <code>draw_line(["my draw", 0.381], d, [128, 45], [250, -72], [79, 255, 0, 142])</code> 以及 <code>draw_line(NULL, d, [23, 111], [70, 333], [23, 178, 222], [7])</code>。</p>
<code>draw_rect</code>	<pre>::mfp::graph_lib::draw::draw_rect(8) :</pre> <p><code>draw_rect(owner_info, display, left_top, width, height, color, frame_or_fill, painting_extra_info)</code> 为绘图事件调度器添加一个绘制长方形的事件。在绘图事件调度器调用这个绘制事件时，该事件将在 <code>display</code> 上绘制一个长方形。</p> <p><code>Draw_rect</code> 有至少 7 个参数。第一个参数是 <code>owner_info</code>。<code>Owner_info</code> 告诉绘图事件调度器谁拥有这个绘图事件。<code>Owner_info</code> 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 <code>id</code>，还可以是 <code>NULL</code>，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 <code>id</code> 的整数，或者代表系统的 <code>NULL</code>，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是任意值。该浮点数的值在清除本绘图</p>

	<p>事件时会发挥作用。第二个参数是 display，它既可以是 screen display，也可以是 image display。第三个参数是包含两个元素的数组 ([x, y])，表示该长方形左上顶点的位置。第四个和第五个参数为长方形的长度和高度。紧接着的参数是 color，代表绘制使用的颜色，它是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是 [Alpha, R, G, B]，如果是 3 个元素，就是 [R, G, B]，在该数组中，每一个元素的值都是从 0 到 255。倒数第二个参数是一个整数。它等于或小于零表示填充长方形，大于零表示长方形的边的宽度。最后一个参数是 painting_extra_info，它告诉绘图事件调度器采用什么样的 porterduff 模式来绘制目标图像。这个参数是可选参数。</p> <p>porterduff 模式内部机制比较复杂，建议开发者省略这个参数（也就是使用参数的缺省值）。如果开发者想要详细了解 painting extra info，可以参考 set_porterduff_mode 以及 get_porterduff_mode 的函数帮助信息。如果开发者想要详细了解 porterduff 模式，建议阅读相关的 JAVA 文档。</p> <p>Draw_rect 的例子包括：draw_rect(["my draw", 0.381], d, [128, 45], 18, 30, [79, 255, 0, 142], 0) 以及 draw_rect(NULL, d, [23, 111], 70, 19, [23, 178, 222], 3)。</p>
draw_oval	<pre>::mfp::graph_lib::draw::draw_oval(8) :</pre> <p>draw_oval(owner_info, display, left_top, width, height, color, frame_or_fill, painting_extra_info) 为绘图事件调度器添加一个绘制椭圆形的事件。在绘图事件调度器调用这个绘制事件时，该事件将在 display 上绘制一个椭圆形。</p> <p>Draw_oval 有至少 7 个参数。第一个参数是 owner_info。Owner_info 告诉绘图事件调度器谁拥有这个绘图事件。Owner_info 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的</p>

	<p><code>id</code>, 还可以是 <code>NULL</code>, 代表系统拥有该事件, 更可以是一个包含两个元素的数组, 其中第一个元素是一个代表拥有者名字的字符串, 或者代表拥有者 <code>id</code> 的整数, 或者代表系统的 <code>NULL</code>, 第二个元素是一个代表时标的浮点数, 但要注意这里的时标不是真正的时标, 该浮点数可以是任意值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 <code>display</code>, 它既可以是 <code>screen display</code>, 也可以是 <code>image display</code>。第三个参数是包含两个元素的数组 (<code>[x, y]</code>) , 表示该椭圆的包络长方形左上顶点的位置。第四个和第五个参数为该椭圆的包络长方形的长度和高度。紧接着的参数是 <code>color</code>, 代表绘制使用的颜色, 它是一个 4 个或 3 个元素的数组, 如果是 4 个元素, 就是 [<code>Alpha, R, G, B</code>], 如果是 3 个元素, 就是 [<code>R, G, B</code>], 在该数组中, 每一个元素的值都是从 0 到 255。倒数第二个参数是一个整数。它等于或小于零表示填充椭圆形, 大于零表示椭圆形的边的宽度。最后一个参数是 <code>painting_extra_info</code>, 它告诉绘图事件调度器采用什么样的 <code>porterduff</code> 模式来绘制目标图像。这个参数是可选参数。<code>porterduff</code> 模式内部机制比较复杂, 建议开发者省略这个参数(也就是使用参数的缺省值)。如果开发者想要详细了解 <code>painting extra info</code>, 可以参考 <code>set_porterduff_mode</code> 以及 <code>get_porterduff_mode</code> 的函数帮助信息。如果开发者想要详细了解 <code>porterduff</code> 模式, 建议阅读相关的 JAVA 文档。</p> <p><code>Draw_oval</code> 的例子包括: <code>draw_oval(["my draw", 0.381], d, [128, 45], 18, 30, [79, 255, 0, 142], 0)</code> 以及 <code>draw_oval(NULL, d, [23, 111], 70, 19, [23, 178, 222], 3)</code>。</p>
<code>clear_rect</code>	<pre>::mfp::graph_lib::draw::clear_rect(5) : clear_rect(owner_info, display, left_top, width, height) 为绘图事件调度器添加一个清除长方</pre>

	<p>形的事件。在绘图事件调度器调用这个绘制事件时，该事件将在 display 上清除一个长方形。Clear_rect 有 5 个参数。第一个参数是 owner_info。Owner_info 告诉绘图事件调度器谁拥有这个绘图事件。Owner_info 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 id，还可以是 NULL，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 id 的整数，或者代表系统的 NULL，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是任意值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 display，它既可以是 screen display，也可以是 image display。第三个参数是包含两个元素的数组 ([x, y])，表示该长方形左上顶点的位置。第四个和第五个参数为长方形的长度和高度。Clear_rect 的例子包括：clear_rect(["my draw", 0.381], d, [128, 45], 18, 30) 以及 clear_rect(NULL, d, [23, 111], 70, 19)。</p>
clear_oval	<pre>::mfp::graph_lib::draw::clear_oval(5) : clear_oval(owner_info, display, left_top, width, height) 为绘图事件调度器添加一个清除椭圆形的事件。在绘图事件调度器调用这个绘制事件时，该事件将在 display 上清除一个椭圆形。Clear_oval 有 5 个参数。第一个参数是 owner_info。Owner_info 告诉绘图事件调度器谁拥有这个绘图事件。Owner_info 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 id，还可以是 NULL，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 id 的整数，或者代表系统的 NULL，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是</pre>

	任意值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 display，它既可以是 screen display，也可以是 image display。第三个参数是包含两个元素的数组 ([x, y])，表示该椭圆的包络长方形左上顶点的位置。第四个和第五个参数为该椭圆的包络长方形的长度和高度。Clear_oval 的例子包括：clear_oval(["my draw", 0.381], d, [128, 45], 18, 30) 以及 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) 为绘 图事件调度器添加一个绘制多边形的事件。在绘图事 件调度器调用这个绘制事件时，该事件将在 display 上绘制一个多边形。Draw_polygon 有至少 7 个参数。 第一个参数是 owner_info。Owner_info 告诉绘图事 件调度器谁拥有这个绘图事件。Owner_info 可以是一 个字符串，代表拥有者的名字，也可以是一个整数， 代表拥有者的 id，还可以是 NULL，代表系统拥有该 事件，更可以是一个包含两个元素的数组，其中第一 个元素是一个代表拥有者名字的字符串，或者代表拥 有者 id 的整数，或者代表系统的 NULL，第二个元素 是一个代表时标的浮点数，但要注意这里的时标不是 真正的时标，该浮点数可以是任意值。该浮点数的值 在清除本绘图事件时会发挥作用。第二个参数是 display，它既可以是 screen display，也可以是 image display。从第三个参数开始是多边形顶点的 位置，它们均为包含两个元素的数组。紧接着的参数 是 color，代表绘制使用的颜色，它是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是[Alpha, R, G, B]，如果是 3 个元素，就是[R, G, B]，在该数组 中，每一个元素的值都是从 0 到 255。倒数第二个参

数是是一个整数。它等于或小于零表示填充多边形，大于零表示多边形的边的宽度。最后一个参数是 painting_extra_info，它告诉绘图事件调度器采用什么样的 porterduff 模式来绘制目标图像。这个参数是可选参数。porterduff 模式内部机制比较复杂，建议开发者省略这个参数（也就是使用参数的缺省值）。如果开发者想要详细了解 painting extra info，可以参考 set_porterduff_mode 以及 get_porterduff_mode 的函数帮助信息。如果开发者想要详细了解 porterduff 模式，建议阅读相关的 JAVA 文档。

Draw_polygon 的例子包括：draw_polygon(["my draw", 0.381], d, [128, 45], [250, -72], [338, 29], [79, 255, 0, 142], 0) 以及 draw_polygon(NULL, d, [23, 111], [70, 333], [-239, 89], [66, 183], [23, 178, 222], 3)。

需要注意一点，clear_rect 和 clear_oval 将一个矩形或者一个椭圆形从显示窗口中“挖”掉，这两个函数调用之后，无论显示窗口的背景颜色和背景图像是什么，被“挖”去的部分都是程黑色。

3. 绘制图像

为显示窗口的绘图事件调度器添加一个绘制图像的函数是 draw_image。该函数有两种不同的调用方法：第一个是 draw_image(owner_info, display, image_or_path, left, top, width_ratio, height_ratio, painting_extra_info)。第二个是 draw_image(owner_info, display, image_or_path, srcx1, srcy1, srcx2, srcy2, destx1, desty1, destx2, desty2, painting_extra_info)。在这两种不同的调用方式中，第一个参数是 owner_info。Owner_info 告诉绘图事件调度器谁拥有这个绘图事件。Owner_info 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 id，还可以是 NULL，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 id 的整数，或者代表系统的 NULL，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是任意

值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 display，也就是显示屏幕的句柄。第三个参数是图像的句柄或者是一个指向图像文件的地址字符串。最后一个参数是 painting_extra_info，它告诉绘图事件调度器采用什么样的 porterduff 模式来绘制目标图像。这个参数是可选参数。porterduff 模式内部机制比较复杂，建议开发者省略这个参数（也就是使用参数的缺省值）。如果开发者想要详细了解 painting extra info，可以参考 set_porterduff_mode 以及 get_porterduff_mode 的函数帮助信息。如果开发者想要详细了解 porterduff 模式，建议阅读相关的 JAVA 文档。在第一种调用方式中，从第四个到第七个参数分别是图像将被绘制的位置的左边界坐标，图像将被绘制的位置的上边界坐标，图像绘制时沿长度方向的缩放比例（是一个可以省略的参数，它的缺省值为 1），以及图像绘制时沿高度方向的缩放比例（是一个可以省略的参数，它的缺省值为 1）。在第二种调用方式中，从第四个到第十一个参数分别是选取图像将被绘制的部分的源长方形的左边界，选取图像将被绘制的部分的源长方形的上边界，选取图像将被绘制的部分的源长方形的右边界，选取图像将被绘制的部分的源长方形的下边界，目标位置的左边界，目标位置的上边界，目标位置的右边界以及目标位置的下边界。Draw_image 的例子 包括：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) 以及 draw_image("imagesrc", display, gem3Img, 0, 0, 32, 32, 210, 540, 300, 580, a_painting_extra_info)。

4. 绘制文字

绘制文字和绘制图像或者几何图形略有不同。绘制文字时，必须考虑文字书写的起始点和文字块的对齐方式。起始点不见得是文字块的左上角，这取决于是在 JAVA 平台还是在安卓平台。文字块的对齐方式和在 Word 中的对齐方式略有不同。文字块的对齐方式是，给定一个长方形，文字块水平左对齐是指整个文字块的最左边和长方形的左边界对齐，文字块水平右对齐是指整个文字块的最右边和长方形的右边界对齐，文字块水平中对齐是指整个文字块的最左边和最右边之间的中点和长方形的左边界和右界的中间线重合。长方形的宽度不见得要比文字块宽，同时，无论水平方向如何对齐，在现阶段，文字块内部每一行都是左对齐的。具体横对齐方式参见下图，其中灰色的表示文字，红色表示边界长方形。

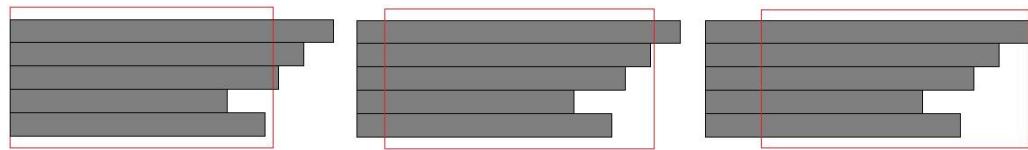


图 8.3： 文字的左中右横对齐方式。

文字块的纵向对齐方式则比较简单，文字块垂直上对齐是指整个文字块的最上边和边界长方形的上边界对齐，文字块垂直下对齐是指整个文字块的最下边和边界长方形的下边界对齐，文字块垂直中对齐是指整个文字块的最上边和最下边之间的中点和长方形的上边界和下边界的中间线重合。

MFP 提供了两个函数分别用于根据文字块的起始点计算文字块的上下左右边界，以及根据设定的边界长方形（不见得一定要比文字块要大）计算文字块起始点的位置。这两个个函数为

函数名	函数帮助信息
calculate_text_boundary	<pre>::mfp::graph_lib::draw::calculate_text_boundary(4) :</pre> <p>calculate_text_boundary(display, string, text_origin, text_style) 返回一个 文本块的边界长方形。边界长方形是一个 4 元素数组，第一个元素是左边界，第二个元素是上边界，第三个元素是宽度，第四个元素是高度。Calculate_text_boundary 的第一个参数是 display，它既可以是 screen display，也可以是 image display。第二个参数是文本块的文本，可以不止一行。第三个参数是文本块起始点的位置。这是一个两元素数组，第一个元素是起始点横坐标，第二个元素是起始点纵坐标。最后一个参数是可省略参数，用于定义文本的字体和大小。如果它被省略，则字体为系统缺省字体，大小为 16。如果它不被省略，那么它必须是一个包含一个或者两个元素的数组。如果是一个元素的数组，那么该元素必须是一个正整数，代表字体的大小，而字体则为系统缺省字体。如果是包含两个元素的数组，那么第一个</p>

	<p>元素为字体的大小，第二个元素是基于字符串的字体的名字。一个本函数的例子为：</p> <pre>calculate_text_boundary(display, txtStr, [108, 190], [27, "SimSun"]).</pre>
calculate_text_origin	<pre>::mfp::graph_lib::draw::calculate_text_origin(8) :</pre> <p>calculate_text_origin(display, string, boundary_rect_left_top, width, height, horAlign, verAlign, text_style)返回指定边界长方形和对齐方式的文本块的起始点。起始点是一个两元素 ([x, y]) 数组，将会被 draw_text 作为参数使用。Calculate_text_origin 的第一个参数是 display，它既可以是 screen display，也可以是 image display。第二个参数是文本块的文本，可以不止一行。第三个参数是边界长方形的左上角位置。这是一个两元素数组，第一个元素是左边界坐标，第二个元素是上边界坐标。第四个和第五个参数是边界长方形的宽度和高度。第六个参数是文本块的横向对齐方式。-1 表示左对齐，0 表示中对齐，1 表示右对齐。第七个参数是文本块的纵向对齐方式。-1 表示上对齐，0 表示中对齐，1 表示下对齐。最后一个参数是可省略参数，用于定义文本的字体和大小。如果它被省略，则字体为系统缺省字体，大小为 16。如果不被省略，那么它必须是一个包含一个或者两个元素的数组。如果是一个元素的数组，那么该元素必须是一个正整数，代表字体的大小，而字体则为系统缺省字体。如果是包含两个元素的数组，那么第一个元素为字体的大小，第二个元素是基于字符串的字体的名字。一个本函数的例子为：</p> <pre>calculate_text_origin(display, "pei is " + peichoices[idx], [256, 72], peiBndrySize[0], peiBndrySize[1], horAlign, verAlign, [22]).</pre>

使用上述函数，用户可以自由地在屏幕任何地方绘制一个长方形，然后在其中填入文字，形成一个按钮。绘制文字的函数为 `draw_text(owner_info, display, string, origin_place, color, text_style, painting_extra_info)`。这个函数为绘图事件调度器添加一个绘制文本块的事件。在绘图事件调度器调用这个绘制事件时，该事件将在 `display` 上绘制一个文本块。`Draw_text` 的第一个参数是 `owner_info`。`Owner_info` 告诉绘图事件调度器谁拥有这个绘图事件。`Owner_info` 可以是一个字符串，代表拥有者的名字，也可以是一个整数，代表拥有者的 id，还可以是 `NULL`，代表系统拥有该事件，更可以是一个包含两个元素的数组，其中第一个元素是一个代表拥有者名字的字符串，或者代表拥有者 id 的整数，或者代表系统的 `NULL`，第二个元素是一个代表时标的浮点数，但要注意这里的时标不是真正的时标，该浮点数可以是任意值。该浮点数的值在清除本绘图事件时会发挥作用。第二个参数是 `display`，也就是显示屏幕的句柄。第三个参数是基于字符串的文本块，它可以多于一行。第四个参数是文本块起始点的位置。这是一个两元素数组，第一个元素是起始点横坐标，第二个元素是起始点纵坐标。第五个参数是 `color`，代表绘制使用的颜色，它是一个 4 个或 3 个元素的数组，如果是 4 个元素，就是 `[Alpha, R, G, B]`，如果是 3 个元素，就是 `[R, G, B]`，在该数组中，每一个元素的值都是从 0 到 255。第六个参数是可省略参数，用于定义文本的字体和大小。如果它被省略，则字体为系统缺省字体，大小为 16。如果它不被省略，那么它必须是一个包含一个或者两个元素的数组。如果是一个元素的数组，那么该元素必须是一个正整数，代表字体的大小，而字体则为系统缺省字体。如果是包含两个元素的数组，那么第一个元素为字体的大小，第二个元素是基于字符串的字体的名字。用户需要注意的是，这个参数必须和在调用 `draw_text` 函数之前调用 `calculate_text_origin` 时和在调用 `draw_text` 函数之后调用 `calculate_text_boundary` 时所使用的 `text_style` 一致，否则文本块的位置会出现偏差。最后一个参数是 `painting_extra_info`，它告诉绘图事件调度器采用什么样的 `porterduff` 模式来绘制目标图像。这个参数是可选参数。`porterduff` 模式内部机制比较复杂，建议开发者省略这个参数（也就是使用参数的缺省值）。如果用户想要详细了解 `painting extra info`，可以参考 `set_porterduff_mode` 以及 `get_porterduff_mode` 的函数帮助信息。如果开发者想要详细了解 `porterduff` 模式，建议阅读相关的 JAVA 文档。`Draw_text` 的例子包括 `draw_text("image", display, txtStr, [108, 190], [255, 255, 255], [10 + idx, font])` 以及 `draw_text("image", display, txtStr, [108, 190], [255, 255, 255], [idx * 2])`。

5. 动画例子

以下给出了一个用 MFP 语言实现动画的例子。“贪吃蛇”是一个风靡全球的小游戏。它的逻辑非常简单，一个在屏幕上快速移动的蛇，玩家可以操控蛇上下左右转向，屏幕上还有一块食物供蛇吃，蛇吃下食物后长长一截，如果蛇头在移动时撞墙了或撞到自己，游戏就输了。

由于操控蛇的移动方向需要用到后面章节方才涉及的用户输入事件处理，在本节中主要展示如何绘图实现蛇的运动以及积累的分数。

在实现贪吃蛇时，通常采用的办法是将蛇的运动空间，通常是一个长方形，在逻辑上横向纵向切割成正方形的小格，每一个小格的宽度（高度）和蛇的身体的宽度正好一样，这样实现蛇的移动，就等价于把蛇头和蛇尾的小格重新绘制，大大减少了绘制图形的工作量。

为了能够尽可能用到以上介绍的所有函数，蛇的食物我们采用贴图片的方式绘制，也就是调用 `draw_image`，蛇的身体则简单地采用 `draw_rect` 绘制矩形，墙体由于比较厚实，也采用 `draw_rect` 方法绘制实心矩形，蛇移动边界用 `draw_line` 绘制线段，控制蛇转向（虽然这一节中还没有介绍如何实现接受用户控制）的上下左右按钮则由 `draw_rect`，`calculate_text_origin` 和 `draw_text` 配合绘制，玩家得分则调用 `draw_text` 绘制在屏幕左下角。

在开始编程之前，建议开发者将一些常用的数值，包括颜色，蛇活动区域的大小，每个方格的大小等定义为函数（因为 MFP 暂时还不支持全局变量），将函数放在代码的最前部，方便对常量参数进行修改。这些定义常量的代码函数如下：

```
// Sleep interval between two updates of screen (ms)  
  
// 两次屏幕刷新之间的睡眠时间（毫秒）  
  
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.
```

```
// 在安卓系统上，睡眠时间较短，原因是MFP在安卓系统上运行得跟慢，需要更多  
多  
  
// 时间进行计算，留给睡眠的时间就少了。  
  
    return 50  
  
else  
  
    return 500  
  
endif  
  
endf
```

```
// width of the game display window in pixels (for pc only)
```

```
// 基于像素的游戏视频窗口的宽度（对安卓不起作用）
```

```
function WINDOWDEFAULTWIDTH()  
  
    return 1024  
  
endf
```

```
// height of the game display window in pixels (for pc only)
```

```
// 基于像素的游戏视频窗口的高度（对安卓不起作用）
```

```
function WINDOWDEFAULTHEIGHT()  
  
    return 480  
  
endf
```

```
// width of button
```

```
// 按钮的宽度
```

```
function BUTTONWIDTH()
```

```
    return 80
```

```
endf
```

```
// height of button
```

```
// 按钮的高度
```

```
function BUTTONHEIGHT()
```

```
    return 60
```

```
endf
```

```
// font size of button
```

```
// 按钮的字体大小
```

```
function BUTTONTXTFONT()
```

```
    return 20
```

```
endf
```

```
// font size of level information
```

```
// 等级信息的字体大小
```

```
function LEVELFONTSIZE()
```

```
    return 30
```

```
endf
```

```
// gap between buttons
```

```
// 按钮之间的间隔
```

```
function BUTTONGAP()

    return 20

endf

// number of columns in snake's moving space

// 蛇的移动空间包含小方格的列数

function GRIDWIDTHDIM()

    return 20

endf

// number of rows in snake's moving space

// 蛇的移动空间包含小方格的行数

function GRIDHEIGHTDIM()

    return 16

endf

// size of grid cell in snake's moving space

// 蛇移动空间的每一个单元小方格的大小

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.

// 蛇的移动空间的左边界到屏幕的左边界的距离。

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.

// 蛇的移动空间的上边界到屏幕的上边界的距离。

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

// 背景色

function BGCOLOR()

    return [170, 190, 255]

endf

// color of the board of snake's moving space

// 蛇的移动空间的边界的颜色。

function BOARDERCOLOR()

    return [125, 255, 100, 100]

endf

// color of the wall

// 墙的颜色。

function WALLCOLOR()

    return [125, 100, 100, 255]

endf

// color of the snake's body

// 蛇的身体的颜色。
```

```
function SNAKECOLOR()

    return [155, 100, 255, 100]

endf

// color of the score

// 玩家得分的颜色。

function SCORECOLOR()

    return [125, 90, 70, 0]

endf

// color of the text

// 文字的颜色。

function TEXTCOLOR()

    return [225, 20, 20, 20]

endf

// color of front edge (facing light edge) in the buttons

// 按钮上的向光边缘的颜色。

function BUTTONFRONTCOLOR()

    return [255, 255, 255]

endf

// color of back edge (not facing light edge) in the buttons
```

```
// 按钮上的背光边缘的颜色。
```

```
function BUTTONBACKCOLOR()
```

```
    return [0, 0, 0]
```

```
endf
```

```
// color of game over text
```

```
// 游戏结束文字的颜色
```

```
function GAMEOVERCOLOR()
```

```
    return [225, 230, 230, 230]
```

```
endf
```

```
// scaling ratio
```

```
// 缩放比例
```

```
function SCALINGRATIO()
```

```
    return 0.5
```

```
endf
```

在定义了常量函数之后，在游戏开始运行之前就应该把常量保存在 variable 中，这样可以避免重复调用函数进行计算，节省了计算时间。

```
// Initial set up color values
```

```
// 设置颜色变量
```

```
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.

    // 首先，我们把窗体的大小，蛇运动空间的网格尺寸，窗体和运动空间的边缘大小，
    按钮尺寸和间隔，缩放比例保存在变量中，这样避免了重复调用

    // 函数进行计算，节省了计算时间

    variable windowHeight = get_display_size(DISPLAYSURF)[0], windowHeight =
    get_display_size(DISPLAYSURF)[1] // window size // 显示窗体的大小

    variable gridWidthDim = GRIDWIDTHDIM(), gridHeightDim = GRIDHEIGHTDIM() // 
    dim of grid (i.e. snake's moving space) // 蛇的运动空间的网格维度

    variable cellSize = CELLSIZE(windowWidth, windowHeight, gridWidthDim,
    gridHeightDim)

    variable xMargin = XMARGIN(windowWidth, windowHeight, gridWidthDim,
    gridHeightDim) // left margin // 左边缘宽度

    variable yMargin = YMARGIN(windowWidth, windowHeight, gridWidthDim,
    gridHeightDim) // top margin // 上边缘宽度

    variable btnW = BUTTONWIDTH(), btnH = BUTTONHEIGHT() // width and height of
    buttons // 按钮尺寸

    variable btnGap = BUTTONGAP() // gap of button // 按钮间隔

    variable scalingRatio = SCALINGRATIO() // scaling ratio // 缩放比例

    variable oneOverScalingRatio = 1/scalingRatio // 1/scaling ratio // 缩放比例
    的倒数

    variable scaledCellSize = cellSize * scalingRatio // scaled cell size // 缩
    放后的网格单元的大小

```

在这之后，就可以对墙的位置，蛇的起始位置和食物的起始位置以及上下左右按钮的位置进行计算，墙体，蛇和食物位置的计算比较简单：

```

    // initial place of food

    // 食物起始位置

```

```

variable foodPlace = calcFoodInitPlace(gridWidthDim, gridHeightDim, level)

// initial place of wall

// 墙的起始位置

variable wallPlace = calcWallPlace(gridWidthDim, gridHeightDim, level)

// initial place of snake

// 蛇起始位置

variable snakePlace = [[3, 5], [3, 4], [3, 3]]

// moving direction: 1 right, -1 left, i up, -i down

// 移动方向, 1 表示向右, -1 表示向左, i 表示向上, -i 表示向下

variable moveDirection = -i

// calculate the grid cells that snake can move, i.e. the moving space of
snake excluding wall and snake body

// 蛇能够自由移动的网格位置, 也就是除去墙和蛇的身体之外的所有网格

variable excludeBarrierPlace = calcExcludeWallPlace(wallPlace, gridWidthDim,
gridHeightDim)

for variable idx = 0 to size(snakePlace)[0] - 1

excludeBarrierPlace[snakePlace[idx][0], snakePlace[idx][1]] = 2

next

```

上下左右按钮的位置则要考虑在安卓系统上是横屏还是竖屏，如果是竖屏，所有的按钮都必须放在屏幕的下部，如果是横屏，或者在 PC 上运行，所有按钮都必须放在屏幕的右侧。如果屏幕是正方形或者太接近正方形（比如黑莓手机的屏幕就是正方形，黑莓也是可以运行安卓应用的），按钮放哪里都不合适，所以干脆就不绘制按钮，玩家依然可以通过手指滑动控制蛇移动方向。计算上下左右按钮的左上角位置的代码如下：

```

// right and bottom of snake's moving space(in pixels)

// 蛇的运动空间的右侧和下侧（基于像素）

```

```

    variable gridRight = xMargin + gridWidthDim * cellSize, gridBottom = yMargin
+ gridHeightDim * cellSize

        // up, down, left, right buttons' left top positions

        // 上下左右按钮的左上角的位置

        variable upBtnLT = [-1, -1], downBtnLT = [-1, -1], leftBtnLT = [-1, -1],
rightBtnLT = [-1, -1]

        variable shouldDrawButtons = true // should we draw the button? // 需要绘制
按钮吗?

        // are the up, down, left, right buttons pushed?

        // 上下左右按钮是否被按下?

        variable upBtnPushed = false, downBtnPushed = false, leftBtnPushed = false,
rightBtnPushed = false

        if (windowWidth - gridRight > btnW * 3)      // buttons are on right hand side
// 按钮放在右边

            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)    // buttons are in bottom //
按钮放在屏幕下部

            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 // 不需要按钮

endif

```

知道了墙体，蛇和按钮的位置，就可以编写绘制它们的函数了。由于墙体，蛇和按钮均为一个一个的对象，所以应该为它们编写专门的绘制函数。墙体和蛇均由一个一个的网格单元组成，所以，为每一个单元格调用 draw_rect 函数即可。共享同样的绘制函数如下：

```

// draw snake or wall, points means the cells' grid coordinates

// 绘制蛇的身体或者墙，points 参数用于存储构成墙或蛇的身体的网格单元的网格坐标

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

        // 一个单元格一个单元格地画方块

        draw_rect(drawBGOwner, display, calcTopLeft(points[idx], cellSize,
xMargin, yMargin) * scalingRatio, cellSize * scalingRatio, cellSize *
scalingRatio, color, 0)

    next

endif

```

绘制按钮则相对复杂。首先，按钮的文字必须位于按钮中央，所以必须调用 calculate_text_origin 确定按钮文字的起始点，然后才能调用 draw_text 将文字写上去，其次按钮有两种状态，按下去和弹起来。在按下去时，左上边界背光，右下边界向光，两处边界的颜色不同；而弹起来时

则完全反过来。所以，不能够简单地调用 draw_rect 画一个空心的方块，而必须调用 draw_line 依次画 4 条不同的线段。其代码如下：

```
// draw button text which must be horizontally and vertically center aligned  
with the rectangular button border  
  
// 绘制按钮的文字。这写文字必须在水平和垂直方向都位于长方形按钮的中心  
  
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.  
  
// 在显示窗口绘制按钮边界。注意按钮有两种状态：按下和没有按下。按钮在按下状态  
时，左上边界是背光色，右下边界是向光色；反之，左上是向光色右下背光色。  
  
function drawButtonBorderOnScreen(info, display, topLeft, width, height,  
isPushed, btnFrontColor, btnBackColor)  
  
    variable color1 = btnFrontColor, color2 = btnBackColor  
  
    if isPushed // is button pushed? // 按钮按下了没有?  
  
        color1 = btnBackColor // back light color // 背光色  
  
        color2 = btnFrontColor // front light color // 向光色
```

```

        endif

        draw_line(info, display, topLeft, [topLeft[0], topLeft[1] + height], color1,
2) // left border // 左边界

        draw_line(info, display, topLeft, [topLeft[0] + width, topLeft[1]], color1,
2) // top border // 上边界

        draw_line(info, display, [topLeft[0], topLeft[1] + height], [topLeft[0] +
width, topLeft[1] + height], color2, 2) // bottom border // 下边界

        draw_line(info, display, [topLeft[0] + width, topLeft[1]], [topLeft[0] +
width, topLeft[1] + height], color2, 2) // right border // 右边界

    endf

```

以上辅助函数和常量定义完成之后，就可以开始绘制静态的图形元素了，这包括墙，蛇的移动网格空间的边界，按钮的文字，以及处于初始状态的蛇的身体，食物和按钮的边框。注意食物是通过 `draw_image` 函数将装入内存的图片对象画到屏幕上的。将图片从硬盘或 SD 卡读入内存并获取大小，需要调用 `load_image` 和 `get_image_size` 函数。这两个函数将在后面的章节中介绍。

```

// open game display window.

// 开启游戏显示窗口。

variable DISPLAYSURF = open_screen_display("Hungry snake", BGCOLOR(), true,
>windowWidth, windowHeight, false)

// draw border for snake's moving space (i.e. grid)

// 绘制蛇的移动空间的边界，也就是网格边界。

draw_rect("static element", DISPLAYSURF, [xMargin, yMargin], gridWidthDim *
cellSize, gridHeightDim * cellSize, boarderColor, 1)

// draw the wall

// 绘制墙

drawPoints("static element", DISPLAYSURF, wallPlace, wallColor, cellSize,
xMargin, yMargin, 1)

```

```
    if(shouldDrawButtons)

        // draw text and border of buttons if needed

        // 如果需要，绘制按钮的文字和边框

        // draw up button

        // 绘制向上按钮

        drawButtonText(DISPLAYSURF, upBtnLT, btnW, btnH, "Up", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, upBtnLT, btnW,
        btnH, upBtnPushed, btnFrontColor, btnBackColor)

        // draw down button

        // 绘制向下按钮

        drawButtonText(DISPLAYSURF, downBtnLT, btnW, btnH, "Down", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, downBtnLT, btnW,
        btnH, downBtnPushed, btnFrontColor, btnBackColor)

        // draw left button

        // 绘制向左按钮

        drawButtonText(DISPLAYSURF, leftBtnLT, btnW, btnH, "Left", false, 1)

        drawButtonBorderOnScreen("button boarder", DISPLAYSURF, leftBtnLT, btnW,
        btnH, leftBtnPushed, btnFrontColor, btnBackColor)

        // draw right button

        // 绘制向右按钮

        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
```

```

    // 一个一个单元格地绘制蛇的身体

    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)

    // 计算食物左上角基于像素的坐标

    variable foodPlaceXY = calcTopLeft(foodPlace, cellSize, xMargin, yMargin)

    // load food image

    // 将作为食物的图片装入内存

    variable foodImage = load_image(get_upper_level_path(get_src_file_path()) +
"food.png")

    // calculate image size

    // 计算食物图片的大小

    variable foodImageSize = get_image_size(foodImage)

    // draw food

    // 绘制食物

    draw_image("food", DISPLAYSURF, foodImage, foodPlaceXY[0], foodPlaceXY[1],
cellSize/foodImageSize[0], cellSize/foodImageSize[1])

```

截至目前，用户看到的依然只是静止的图像，为了让蛇能够动起来，首先必须将过去为了绘制蛇而调用 `draw_rect` 所生成的绘图事件从绘图事件调度器中移除，然后动态地修改蛇的位置并重新绘制蛇的身体（也就是通过调用 `draw_rect` 生成新的绘图事件），然后调用 `update_screen` 将屏幕更新，最后程序还要再休眠一段时间再进行蛇的下一次运动。如果没有休眠，屏幕刷新太快，玩家会觉得屏幕在闪烁。具体的代码如下：

```
variable deltaX = 0, deltaY = 0 // snake head movement // 蛇头的移动

while true

    update_display(DISPLAYSURF) // update game display window // 更新游戏显示窗口

    select (moveDirection) // determine snake head movement from move direction // 根据运动方向计算蛇头移动的量

        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)

    // 计算蛇头基于网格坐标的新位置

    variable newX = mod(snakePlace[0][0] + deltaX, gridWidthDim)

    variable newY = mod(snakePlace[0][1] + deltaY, gridHeightDim)

    variable head2Add = [newX, newY]

    // because snake head has occupied the cell, food or snake body cannot
    take it again

    // 由于新的蛇头占据了该单元格，蛇的身体和食物不可能出现在该单元格

    excludeBarrierPlace[newX, newY] = 2

    // insert the new snake head place into snakePlace list

    // 将新的蛇头的位置插入蛇的身体空间位置数组

    snakePlace = insert_elem_into_ablist(snakePlace, 0, head2Add)

    // remove tail of the snake

    // 将蛇尾的最后一个单元格移去

    variable tailIdx = size(snakePlace)[0] - 1

    variable tail2Remove = snakePlace[tailIdx]

    // now the cell is available for food and snake's body

    // 现在单元格空出来了，食物和蛇的身体可以占用该单元格了

    excludeBarrierPlace[tail2Remove[0], tail2Remove[1]] = 0

    // remove the old snake tail place into snakePlace list

    // 将旧的蛇尾的位置从蛇的身体空间位置数组中去掉

    snakePlace = remove_elem_from_ablist(snakePlace, tailIdx)

    sleep(MOVEINTERVAL()) // sleep a while //休眠一会儿

```

```

    // remove all the old painting event requests for snake from the
    painting request scheduler

    // 将旧的蛇的绘图事件从绘图事件调度器中移除

    drop_old_painting_requests("snake", DISPLAYSURF)

    // redraw snake body cell by cell

    // 重新一个一个单元格地绘制蛇的身体

    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



图 8.4： 让蛇在屏幕上动起来。

现在运行上述代码，可以看到一条绿色的“蛇”在屏幕上从上往下快速移动，移动到底部后又从顶部重新出现。粉红色的食物（一条鱼）位于蛇的移动空间的中央，蓝色的墙位于四个角，上下左右按钮位于窗口的右侧，注意此时虽然蛇可以移动，但是用户无法操控移动的方向。

第3节 处理图像和声音

在前面的章节中已经介绍了通过MFP编程的方式实现动画的原理。但是，上述编程方式存在一个问题。由于每一个物体，无论是墙，按钮还是蛇都需要多次调用绘图函数进行绘制，绘制一个比较复杂的图案，需要生成大量的绘图事件，如果直接绘制到屏幕上，频繁读写速度较慢的显存将占用大量的计算时间。更好的解决办法是将所有的图案绘制到一个图像对象中，然后直接将该图像对象“贴”到显示窗口上。这样就只需要一个绘图事件来读写显存，效率大大提高。这就涉及到了图像处理的问题。

此外，在运行游戏时常常需要发出各种声音，如何播放声音文件也将在本节中涉及。

1. 创建，装载，克隆和保存图像

以下函数用于创建，装载，克隆和保存图像，以及获取图像的大小和判断内存中的图像句柄是否依然有效。

函数名	函数帮助信息
create_image	::mfp::multimedia::image_lib::create_image(2)： create_image(w, h)返回一个全新的，空白的被包装过的JAVA图像对象。该图形对象的宽度为w，高度为h。
load_image	::mfp::multimedia::image_lib::load_image(1)： load_image(image_path)返回一个被包装过的JAVA图像对象。它有一个参数image_path。这个参数是一个基于字符串的，指向一个图形文件的路径。
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)返回一个被包装过

	的 JAVA 图像对象。该图形对象从一个 zip 压缩文件中读取。它的第一个参数是基于字符串的 zip 文件的压缩路径。它的第二个参数是图像在该 zip 文件中的压缩路径。它的第三个参数要么是 0，要么是 1。如果等于 0，表示普通的 zip 文件，而如果等于 1，表示 MFP App 的安卓 assets 中的 zip 文件。
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) 返回一个新的被包装过的 JAVA 图像对象。该图形对象的宽度为 dest_width，高度为 dest_height。这个被返回的图像是（被拉缩过的）原图像 image_src 的被选中的区块的拷贝。被选中的区块的左、上、右、下的坐标分别是 src_left、src_top、src_right 和 src_bottom。注意 src_left、src_top、src_right 和 src_bottom 是可选参数，它们的缺省值分别是 0、0、image_src 的宽度和 image_src 的高度。dest_width 和 dest_height 也是可选参数。它们的缺省值分别是 src_right - src_left 以及 src_bottom - src_top。本函数的一个例子为 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) 保存一个被包装过的 JAVA 图像对象至一个图形文件。本函数的第一个参数是被包装过的 JAVA 图像对象，第二个参数是基于字符串的图像文件的格式，当前仅支持 "png", "jpg" 以及 "bmp" 格式。第三个参数是图像文件的路径。如果成功保存，本函数返回 True，否则返回 False。本函数的一个例子为： save_image(img, "png", "C:\\Temp\\1.png")
get_image_size	::mfp::multimedia::image_lib::get_image_size(1

) :
	get_image_size(image_handle) 返回由 image_handle 所代表的被包装过的 JAVA 图像对象的长和宽组成的数组。
is_valid_image_handle	::mfp::multimedia::image_lib::is_valid_image_handle(1) : is_valid_image_handle(image_handle) 返回一个布尔量，用于告诉开发人员一个被包装过的 JAVA 图像对象，也就是 image_handle 参数，是否依然合法还是已经被关闭。

2. 修改图像

在图像装入内存之后，在将其贴到屏幕上之前，开发者往往需要对它进行修改，也就是在图像对象上进一步作图。为了方便开发者，MFP 提供了一套和在显示窗口作图完全一致的作图函数。用户需要调用 open_image_display 函数，为图像创建一个“显示窗口”（也就是 image display）。该显示窗口的大小和图像的大小一致（和真正的游戏显示窗口类似，图像的“显示窗口”也可以由 set_display_size 函数进行调整），图像对象则变成图像的“显示窗口”的背景图片，然后开发者调用诸如 draw_rect, draw_image 等函数在“显示窗口”绘图，和真正的游戏显示窗口类似，开发者也可以调用 update_display 和 drop_old_painting_requests 函数对绘图事件和绘图时间进行控制。绘图完成后，需要调用 get_display_snapshot 函数（该函数也适用于真正的游戏显示窗口）获得“显示窗口”的截图，也就是修改后的图像对象，这个截取的图像对象就可以为 draw_image 函数所用，直接往游戏显示窗口上“贴”。最后开发者调用 shutdown_display 关闭“显示窗口”。

这里需要注意几点。首先，为图像创建显示窗口并作图不会改变原有图像，换句话说，做完图，shutdown_display 之后，原有图像还是老样子。get_display_snapshot 只是返回一个新的图像对象的句柄。第二，和真正的游戏显示窗口类似，如果生成太多的绘图事件，更新图像将会花费很多计算时间，一个好的办法是调用 set_display_snapshot_as_bgrnd 函数。这个函数将显示窗口（不论是图像的“显示窗口”，也就是 image display，

还是真正的游戏显示窗口，也就是 screen display) 的截屏作为显示窗口新的背景图像。这样原有的绘图事件就可以移除而不必重绘了。

上述函数的详细说明如下：

函数名	函数帮助信息
open_image_display	<pre>::mfp::multimedia::image_lib::open_image_display(1) : open_image_display(image_path_or_handle) 创建一个 image display 供开发人员调用 MFP 函数绘图。它有一个参数。这个参数既可以是一个基于字符串的，指向一个图形文件的路径，也可以是 null，还可以是一个由 load_image, load_image_from_zip, create_image 或者 clone_image 函数返回的 JAVA image 对象的句柄。</pre>
get_display_snapshot	<pre>::mfp::graph_lib::display::get_display_snapshot(4) : get_display_snapshot(display, update_screen_or_not, width_ratio, height_ratio) 返回一个 display (既可以是 screen display，也可以是 image display) 的截屏。它的第二个参数，update_screen_or_not，告诉 MFP 在截屏之前该 display 是否需要刷新；它的第三个和第四个参数，是可选参数，缺省值均为 1，分别用于告诉 MFP 返回的截屏的长度和高度的缩放比例。比如，get_display_snapshot(d, true, 0.5, 3) 首先刷新屏幕 d，然后截取 d 的图像，最后将截取的图像长度上压缩为原来的一半，高度上拉伸为原来的三倍并返回新的图像。</pre>
set_display_snapshot_as_bgrnd	<pre>::mfp::graph_lib::display::set_display_snapshot_as_bgrnd(3) :</pre>

set_display_snapshot_as_bgrnd(display,
update_screen_or_not, clear_callbacks_or_not)
将一个 display (既可以是 screen display, 也可以是 image display) 的截屏设置为它的背景图案。它的第二个参数, update_screen_or_not, 告诉 MFP 在截屏之前该 display 是否需要刷新; 它的第三个参数, clear_callbacks_or_not, 告诉 MFP 是否需要将屏幕绘图事件序列清空。比如,
set_display_snapshot_as_bgrnd(d, true, true) 首先刷新屏幕 d, 然后将屏幕绘图事件序列清空, 最后截屏并将所得图像作为屏幕背景图像。

以下代码给出了贪吃蛇游戏中通过使用图像“显示窗口”(也就是 image display) 绘制该游戏的静止的物件的示例。图像“显示窗口”的截屏被贪吃蛇游戏拿来用作游戏的实际显示窗口(也就是 screen display) 的背景图案。这样有效避免了多步绘制各种小物件, 加快了计算速度:

```
// open an empty image display  
  
// 打开一个空的图像显示窗口 (image display)  
  
variable boardImageDisplay = open_image_display(null)  
  
// adjust it's size to game display window's size times scaling ratio  
  
// 将图像显示窗口的大小调整为游戏真实显示窗口的大小乘以缩放系数  
  
    set_display_size(boardImageDisplay, windowHeight * scalingRatio,  
* scalingRatio)  
  
// calculate text origin of level information  
  
// 计算通关级数信息文字的起始位置  
  
variable textOrigin = [10, 10]  
  
variable levelFontSize = LEVELFONTSIZE()  
  
if xMargin > yMargin  
  
// text is in the center of left edge rectangle
```

```

        // 文字位于左边缘长方形的正中

        textOrigin = calculate_text_origin(DISPLAYSURF, "level " + level, [0, 0],
xMargin, windowHeight, 0, 0, levelFontSize)

    else

        // text is in the center of top edge rectangle

        // 文字位于上边缘长方形的正中

        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.

    // 绘制通关级数信息文字。注意由于图像显示窗口的真实尺寸比游戏真实显示窗口的
尺寸要小，文字被相应缩小了。

    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.

    // 绘制蛇的移动空间的边界。注意由于图像显示窗口的真实尺寸比游戏真实显示窗口
的尺寸要小，边界矩形被相应缩小了。

    draw_rect("static element", boardImageDisplay, [xMargin, yMargin] *
scalingRatio, gridWidthDim * scaledCellSize, gridHeightDim * scaledCellSize,
borderColor, 1)

    // draw the wall. Note that the wall is scaled down to fit the image.

    // 绘制墙体。注意由于图像显示窗口的真实尺寸比游戏真实显示窗口的尺寸要小，墙
体被相应缩小了。

    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.  
  
// 在这里仅仅绘制按钮上的文字而不绘制按钮的边框因为文字不会变化，而边框  
会随着按下弹起发生变化。  
  
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  
  
// 取回图像显示窗口的截图。注意在获取截图前，先将图像显示窗口更新。  
  
variable boardImage = get_display_snapshot(boardImageDisplay, true)  
  
// shutdown image display  
  
// 关闭图像显示窗口  
  
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.  
  
// 将上述图像显示窗口的截屏设置为游戏真实显示窗口的背景图案。注意背景图案的  
设置模式是 1，也就是缩放背景图案让它和  
  
// 游戏真实显示窗口大小一致。
```

```
set_display_bgrnd_image(DISPLAYSURF, boardImage, 1)
```

3. 声音的播放

MFP 为开发者提供了一系列的声音处理函数可以实现声音的播放，循环播放，音量调整和停止。这些声音处理函数如下：

函数名	函数帮助信息
play_sound	::mfp::multimedia::audio_lib::play_sound(4) : play_sound(source_path, repeat_or_not, volume, create_new_or_not)演奏一个声音文件，该声音文件可以是 wave 文件，也可以是 midi 文件，还可以是 mp3 文件。该函数返回一个演奏器的句柄，该句柄指向一个 JAVA 或安卓的多媒体演奏器。由于多媒体演奏器的资源是有限的，本函数会尽可能的回收并重用以前生成的多媒体演奏器。本函数有 4 个参数。第一个参数是声音文件的路径。第二个参数是一个布尔值，表示该声音是否需要重复演奏，这是一个缺省参数，缺省值是 false。第三个参数是一个从 0 到 1 的浮点数，表示音量大小。这也是一个缺省参数，缺省值是 1。第四个参数是一个布尔值，表示是否无论如何都强制生产一个新的多媒体演奏器。这也是一个缺省参数，缺省值是 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)演奏一个从 zip 文件中抽出的声音文件，该声音文件可以是 wave 文件，也可以是 midi 文件，还可以是 mp3 文件。该函数返回一个演奏器的句柄，该句柄指向一个 JAVA 或安卓的多媒体演奏器。由于多媒体演奏器的资源是有限的，本函数会尽可能的回收并重用以前生成的多媒体

	演奏器。本函数有 6 个参数。第一个参数是 zip 文件的路径。第二个参数是被压缩的声音文件的在 zip 文件中的位置路径。第三个参数是一个布尔值，0 表示 zip 文件是普通的压缩文件，1 表示 zip 文件位于 MFP app 的安卓 asset 目录中。第四个参数是一个布尔值，表示该声音是否需要重复演奏，这是一个缺省参数，缺省值是 false。第五个参数是一个从 0 到 1 的浮点数，表示音量大小。这也是一个缺省参数，缺省值是 1。第六个参数是一个布尔值，表示是否无论如何都强制生产一个新的多媒体演奏器。这也是一个缺省参数，缺省值是 false。
start_sound	::mfp::multimedia::audio_lib::start_sound(1) : start_sound(sound_handle) 演奏 sound_handle 所指向的声音文件。如果该声音文件已经启动，这个函数什么也不做。
stop_all_sounds	::mfp::multimedia::audio_lib::stop_all_sounds(0) : stop_all_sounds() 停止所有正在播放的声音。
stop_sound	::mfp::multimedia::audio_lib::stop_sound(1) : stop_sound(sound_handle) 停止 sound_handle 所代表的声音的播放。如果该声音没有播放，这个函数什么也不做。
get_sound_path	::mfp::multimedia::audio_lib::get_sound_path(1) : get_sound_path(sound_handle) 返回 sound_handle 所指向的声音文件的路径。
get_sound_referenc e_path	::mfp::multimedia::audio_lib::get_sound_refere nce_path(1) : get_sound_reference_path(sound_handle) 返回

	sound_handle 所指向的声音引用文件的路径。如果声音文件不是从 zip 压缩的读入的，声音引用文件和 sound_handle 所指向的声音文件（也就是 get_sound_file 函数的返回值）是同一个文件。如果声音文件是从 zip 压缩的读入的，声音引用文件路径是压缩文件的路径加上声音文件的压缩路径，比如 "/folder1/folder2/snd.zip/zipped_folder/snd.wav"，这里"/folder1/folder2/snd.zip"是压缩文件路径，"zipped_folder/snd.wav"是声音文件的压缩路径。
get_sound_repeat	::mfp::multimedia::audio_lib::get_sound_repeat(1) : get_sound_repeat(sound_handle) 返回一个布尔量，表示参数 sound_handle 所代表的声音是否会被重复演奏。
get_sound_source_type	::mfp::multimedia::audio_lib::get_sound_source_type(1) : get_sound_source_type(sound_handle) 返回一个整数，代表 sound_handle 所指向的声音引用文件的类型。0 表示常规文件，1 表示压缩的 zip 文件，2 表示引用于 MFP App 中安卓 asset 中的 zip 文件。
get_sound_volume	::mfp::multimedia::audio_lib::get_sound_volume(1) : get_sound_volume(sound_handle) 返回参数 sound_handle 所代表的声音的音量（一个变化范围从 0 到 1 的浮点数）。
set_sound_repeat	::mfp::multimedia::audio_lib::set_sound_repeat(2) : set_sound_repeat(sound_handle, repeat_or_not) 设置一个 sound_handle 所代表的声音是否重复演

	奏。
set_sound_volume	::mfp::multimedia::audio_lib::set_sound_volume (2) : set_sound_volume(sound_handle, volume) 设置一个 sound_handle 所代表的声音的音量，注意音量参数 volume 的值变化范围是从 0 到 1。

在这里需要注意的是，由于资源有限，MFP 会尽可能重复使用声音演奏器。MFP 判断一个演奏器是否可以重复利用的标准是比较声音文件的引用路径（reference path），如果该声音文件的引用路径和一个已有的演奏器的引用路径完全一样，MFP 会认为已有的演奏器可以重复利用。否则 MFP 会生成一个新的演奏器资源。

声音文件的引用路径在大多数情况下和声音文件的实际路径一样。但如果声音文件是从一个 zip 文件，包括 MFP 应用的 asset 下的 zip 文件，中读出，该声音文件会被拷贝到 SD 卡或者硬盘上的一个临时的位置。那么该声音引用文件路径是压缩文件的路径加上声音文件的压缩路径，比如 “/folder1/folder2/snd.zip/zipped_folder/snd.wav” ，这里 “/folder1/folder2/snd.zip” 是压缩文件路径，“zipped_folder/snd.wav” 是声音文件的压缩路径。而该声音文件的路径则是拷贝到临时位置的路径。在这种情况下，引用路径和路径是不一样的。

4. 在不同的平台上装载声音图像附件

MFP 具有跨平台运行的能力。很显然在安卓平台上运行 APK 和在普通的 PC 机上运行脚本大不一样。在运行脚本时，可以把游戏需要的声音或图像文件放在随便一个目录下，然后在脚本中指出路径即可。但对于 APK 应用，不存在路径的说法，所有的附属文件均被放在 APK 包的 asset 文件夹中。

对于无论是在 JAVA 平台还是在安卓平台上的脚本，MFP 的解决办法是，把所有的附属文件都保存在脚本所在的目录中。然后通过调用 get_src_file_path() 函数（没有参数）获取当前运行的脚本文件的路径，然后再调用 get_upper_level_path 函数（该函数只有一个参数，就是文件的路径，在这里就是 get_src_file_path() 函数的返回值），返回当前运行的脚本文件的所在目录，通过目录，就可以找到附属文件完整路径，然后可以直接调用 load_image 和 play_sound 函数读入这些文件。

对于 MFP 应用，解决办法是在 APK 包的 asset 文件夹中建立一个叫做 resource 的 zip 文件，把所有的附属文件均压缩到 resource.zip 压缩包中。在 MFP 应用运行的时候，调用 load_image_from_zip 和 play_sound_from_zip 函数读取图像和声音附件。

在将一个脚本打包成 APK 安装文件时，开发者需要告诉 MFP 哪些资源文件将会被拷贝到新建 APP 的 asset 中的 resource.zip 压缩文件中，比如下述例子：

```
...
@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
...
```

资源文件的文件名叫做 eatfood.wav。这个资源文件被放在当前正在运行的 MFPS 脚本文件的所在目录中。这样一来，该资源文件的路径是 get_upper_level_path(get_src_file_path()) + "eatfood.wav"。如果不是运行 MFP 应用，开发者可以调用 play_sound 函数，根据该资源文件的路径，直接从硬盘或者 SD 卡中读取该资源文件并演奏。但是，如果是运行的 MFP 应用，MFPS 脚本文件被打包在 APK 文件中，开发者则必须从 APK 的 asset 下的 resource.zip 文件中加载资源。APK 的 asset 下的 resource.zip 文件的路径由 get_asset_file_path("resource") 给出，"sounds/eatfood.wav" 代表该资源文件在压缩文件中的位置，1 表示安卓 asset 资源文件，play_sound_from_zip 将该资源文件抽取出来并演奏。

当用户创建 APK 包时，@build_asset 标注所起的作用就是将资源文件从硬盘或者 SD 卡上拷贝到 APK 的 asset 下的 resource.zip 文件中。

`copy_to_resource` 实际上是一个 MFP 函数。它的第一个参数是资源文件的源路径（也就是在硬盘或 SD 卡上的路径），第二个参数是目标路径，也就是在 `resource.zip` 文件中的位置。和普通函数不一样的是，`copy_to_resource` 函数位于 `mfp_compiler` 引用空间中。该引用空间仅仅在打包链接的时候被加载，一般情况下用户不会用到。

还要注意`@build_asset` 指令必须位于一个函数的内部，如果它在 `function` 语句前面或者 `endf` 语句之后，它不会有任何作用。

第 4 节 玩家输入的事件处理

在现阶段，MFP 语言提供了 4 个函数用于读取和分析玩家的输入事件，它们是：

函数名	函数帮助信息
<code>pull_event</code>	<code>::mfp::graph_lib::event::pull_event(1) :</code> <code>pull_event(display)</code> 从 <code>screen display</code> 的输入事件（比如，鼠标事件或者触摸屏的手势事件）序列中按顺序取走一个事件。如果没有输入事件，或者不是 <code>screen display</code> 而是 <code>image display</code> ，它返回 <code>Null</code> ，否则返回输入事件。
<code>get_event_type</code>	<code>::mfp::graph_lib::event::get_event_type(1) :</code> <code>get_event_type(event)</code> 返回一个整数，表示事件 <code>event</code> 的类型。在现阶段支持以下事件： <code>GDI_INITIALIZE</code> （类型为 1，产生于 <code>screen display</code> 被创建的时候）， <code>GDI_CLOSE</code> （类型为 10，产生于 <code>screen display</code> 被关闭的时候）， <code>WINDOW_RESIZED</code> （仅仅用于 JAVA 平台，类型为 21，产生于 <code>screen display</code> 的窗口大小被调整的时候）， <code>POINTER_DOWN</code> （类型为 102，产生于鼠标按钮被按下或者用户手指点到触摸屏的时候）， <code>POINTER_UP</code> （类型为 103，产生于鼠标按钮弹起或者用户手指离开触摸屏的时候）， <code>POINTER_CLICKED</code> （类型为 104，产生于鼠标按

	钮快速按下再立即弹起或者用户手指轻敲触摸屏的时候），POINTER_DRAGGED（类型为105，产生于鼠标或者用户手指按下并拖动的时候。注意只要拖动还在进行中，这个事件就会连续地触发），POINTER_SLIDED（类型为106，产生于鼠标或者用户手指按下并拖动到终点，鼠标按钮弹起或者用户手指离开触摸屏的时候，注意这个事件只是在拖动事件结束的时候触发一次），POINTER_PINCHED（仅用于安卓平台，类型为201，当用户捏合手指进行类似缩放操作的时候触发）。
get_event_type_name	<code>::mfp::graph_lib::event::get_event_type_name(1) :</code> <code>get_event_type_name(event)</code> 返回一个字符串，表示事件 <code>event</code> 的类型的名字。在现阶段支持以下事件：“GDI_INITIALIZE”（产生于 <code>screen display</code> 被创建的时候），“GDI_CLOSE”（产生于 <code>screen display</code> 被关闭的时候），“WINDOW_RESIZED”（仅仅用于JAVA平台，产生于 <code>screen display</code> 的窗口大小被调整的时候），“POINTERT_DOWN”（产生于鼠标按钮被按下或者用户手指点到触摸屏的时候），“POINTERT_UP”（产生于鼠标按钮弹起或者用户手指离开触摸屏的时候），“POINTERT_CLICKED”（产生于鼠标按钮快速按下再立即弹起或者用户手指轻敲触摸屏的时候），“POINTERT_DRAGGED”（产生于鼠标或者用户手指按下并拖动的时候。注意只要拖动还在进行中，这个事件就会连续地触发），“POINTERT_SLIDED”（产生于鼠标或者用户手指按下并拖动到终点，鼠标按钮弹起或者用户手指离开触摸屏的时候，注意这个事件只是在拖动事件结束的时候触发一次），“POINTERT_PINCHED”（仅用于安卓平台，当用户捏合手指进行类似缩放操作的时候触发）。
get_event_info	<code>::mfp::graph_lib::event::get_event_info(2) :</code>

`get_event_info(event, property_name)` 返回 event 事件的一个特性。它的第一个参数是事件，第二个参数是基于字符串的特性的名字。GDI_INITIALIZE 和 GDI_CLOSE 事件没有单独的特性。WINDOW_RESIZED 事件有四个整数特性，它们是“width”，“height”，“last_width”，“last_height”。它们分别表示当前窗口的宽度和高度，以及事件触发前窗口的宽度和高度。POINT_DOWN，POINT_UP 和 POINT_CLICKED 事件均有三个特性。其中，“button”特性是一个整数，表示鼠标的哪一个键触发了这个事件。注意在安卓平台上，“button”特性总是 0。“x”和“y”表示事件触发时的坐标位子，它们均为浮点数。POINT_DRAGGED 和 POINT_SLICED 事件均有 5 个特性。其中，“button”特性是一个整数，表示鼠标的哪一个键触发了这个事件。注意在安卓平台上，“button”特性总是 0。“last_x”和“last_y”表示事件触发前的坐标位子，它们均为浮点数。“x”和“y”表示事件触发后的坐标位子，它们均为浮点数。PINCH_EVENT 事件有八个特性，它们是“last_x”、“last_y”、“last_x2”、“last_y2”、“x”、“y”、“x2”以及“y2”。这些特性定义了参与 PINCH_EVENT 事件的两根手指头在事件触发前和事件触发后的坐标位置。这些特性均为浮点数。

在游戏运行时，所有的用户输入事件均被保存在一个用户输入事件序列中，通过循环调用 `pull_event` 函数，用户的输入事件不断地被取出和处理。具体在贪吃蛇游戏中，代码的实现如下：

```
do // looping to read player's input events // 循环读取玩家输入事件
    variable giEvent = pull_event(DISPLAYSURF)
    if giEvent == Null
        // no event to handle // 没有输入事件
    break
```

```
    elseif get_event_type_name(giEvent) == "GDI_CLOSE"
        // quit // 游戏退出事件
        return -1

    elseif get_event_type(giEvent) == 106 // mouse or finger滑动 // 鼠标或手
指滑动事件
        // x1 and y1 are the coordinate when sliding starts, x2 and y2 are the
        // coordinate when sliding finishes
        // x1 和 y1 是滑动开始时的坐标位置, x2 和 y2 是滑动结束时的坐标位置
        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 doesn't happen
        // in button area.
        // 确保用于控制蛇移动方向的滑动事件不是在按钮区域发生
        if !(shouldDrawButtons, x1 < btbsLeft, x1 > btbsRight, y1 < btbsTop,
            y1 > btbsBottom)
            // calculate moving direction
            // 计算蛇的移动方向
            if abs(y2 - y1) > abs(x2 - x1)
                if y2 > y1      // move down //向下移动
                    moveDirection = -i
                else // move up //向上移动
                    moveDirection = i
            endif
```

```

        elseif abs(y2 - y1) < abs(x2 - x1)

            if x2 > x1    // move right //向右移动

                moveDirection = 1

            else // move left //向左移动

                moveDirection = -1

            endif

        endif // if y2 - y1 == x2 - x1, do nothing // 如果移动方向正好是
45° 或者 135° , 就不改变当前的运动方向

    endif

    elseif get_event_type_name(giEvent) == "POINTER_DOWN" // mouse or finger
tapped down // 鼠标或手指按下事件

    if gameIsOver

        // if game is over at this level, identify if player can go to next
level based on the score

        // 如果这一关游戏已经结束, 根据得分情况判断玩家是否能够通关

        if and(level < 2, score >= score_thresh)

            return 1    // can go to next level. // 能够进入下一关

        else

            return 0    // cannot go to next level. // 无法进入下一关

        endif

    elseif (shouldDrawButtons)

        // if game is not over at this level, identify if a button is hit

        // 如果这一关游戏还未结束, 判断是否有按钮被按下

        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.  
  
        // 这个事件发生在按钮区域。如果有按钮被按下，更新按钮的状态并重  
        置蛇的移动方向。  
  
        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  
until false
```

第5节 游戏示例贪吃蛇的编程逻辑

本章的前面几节介绍了如何打开游戏显示窗口，如何在游戏显示窗口绘制静态物体（比如按钮上的文字）和动画（比如运动的蛇和按钮在按下和弹起时的状态），以及如何处理玩家输入事件。有了这些作为基础，诸如贪吃蛇这样的简单小游戏很容易开发。

贪吃蛇的编程逻辑是，蛇在运动空间中运动，玩家通过滑动手指或者按按钮调整蛇的运动方向。蛇运动时，如果蛇头将要进入的单元格和食物所在单元格重合，首先播放吃到食物的声音，然后将蛇头绘制在该单元格，将该单元格的网格坐标插入蛇的身体空间位置数组，最后任选一个空白单元格绘制食物并更新分数；如果蛇头将要进入的单元格是墙体或者蛇自己的身体，就播放撞墙的声音，然后游戏退出；如果蛇头将要进入的单元格是一个空单元格，则将蛇头绘制在该单元格，将该单元格的网格坐标插入蛇的身体空间位置数组，然后将蛇尾单元格清除，并将蛇尾单元格的网格坐标从蛇的身体空间位置数组中删掉。注意所谓绘制蛇头，清除蛇尾都只是指的绘图事件。开发者必须调用 drop_old_painting_requests 清除旧的绘图事件，调用 update_display 函数应用新的绘图事件。具体的代码见下。

```
// calculate the new snake head place (in grid coordinate)

// 计算蛇头基于网格坐标的新位置

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. // 需要绘制的东西

if and(newX == foodPlace[0], newY == foodPlace[1])

// eat the food. play the eat food sound.

// 吃到食物，演奏吃到食物的声音。

@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

    // 蛇头现在在食物的旧的位置

    head2Add = [newX, newY]

    // because snake head has occupied the cell, food or snake body
cannot take it again

    // 由于新的蛇头占据了该单元格，蛇的身体和食物不可能出现在该单元
格

    excludeBarrierPlace[newX, newY] = 2

    // insert new head's position in the snake position list

    // 将新的蛇头的位置插入蛇的身体空间位置数组

    snakePlace = insert_elem_into_ablist(snakePlace, 0, head2Add)

    // calculate new food place

    // 计算食物的新的位置

    newFood = foodPlace = getNextFoodPlace(excludeBarrierPlace,
wallPlace, snakePlace, gridWidthDim, gridHeightDim)

    // update score

    // 更新分数

    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.

        // 蛇撞到了墙或它自己。演奏撞墙的声音。本关游戏结束。

        @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

            // 常规移动

            head2Add = [newX, newY]

            // because snake head has occupied the cell, food or snake body
            cannot take it again

            // 由于新的蛇头占据了该单元格，蛇的身体和食物不可能出现在该单元
            格

            excludeBarrierPlace[newX, newY] = 2

            // insert the new snake head place into snakePlace list

            // 将新的蛇头的位置插入蛇的身体空间位置数组

            snakePlace = insert_elem_into_ablist(snakePlace, 0, head2Add)

            // remove tail of the snake

```

```

        // 将蛇尾的最后一个单元格移去

        variable tailIdx = size(snakePlace)[0] - 1

        tail2Remove = snakePlace[tailIdx]

        // now the cell is available for food and snake's body

        // 现在单元格空出来了，食物和蛇的身体可以占用该单元格了

        excludeBarrierPlace[tail2Remove[0], tail2Remove[1]] = 0

        // remove the old snake tail place into snakePlace list

        // 将旧的蛇尾的位置从蛇的身体空间位置数组中去掉

        snakePlace = remove_elem_from_ablist(snakePlace, tailIdx)

        endif

        // game hasn't been over so that redraw the snake, food and score

        // 本关游戏还未结束，所以重画蛇，食物和分数

        if head2Add != null

            // remove all the old painting event requests for snake from the
            // painting request scheduler

            // 将旧的蛇的绘图事件从绘图事件调度器中移除

            drop_old_painting_requests(["snake", totalSnakeLen -
                size(snakePlace)[0] + 1], DISPLAYSURF)

            // redraw snake body cell by cell

            // 重新一个一个单元格地绘制蛇的身体

            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

            // 将旧的食物的绘图事件从绘图事件调度器中移除

            drop_old_painting_requests("food", DISPLAYSURF)

            // calculate food's position and redraw the food

            // 计算食物的位置并且重绘食物

            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

            // 重绘分数

            drop_old_painting_requests("score", DISPLAYSURF)

            draw_text("score", DISPLAYSURF, ""+newScore, [10, windowHeight -
            32], scoreColor, 25)

        endif

        update_display(DISPLAYSURF) // update game display window // 更新游
        戏显示窗口

```

贪吃蛇游戏完整的代码可在本手册自带的示例代码所在目录中的 2d games 子目录中 hungry_snake 子目录下的 hungry_snake.mfps 文件中找到。

贪吃蛇游戏在安卓手机中运行的视频截图如下：

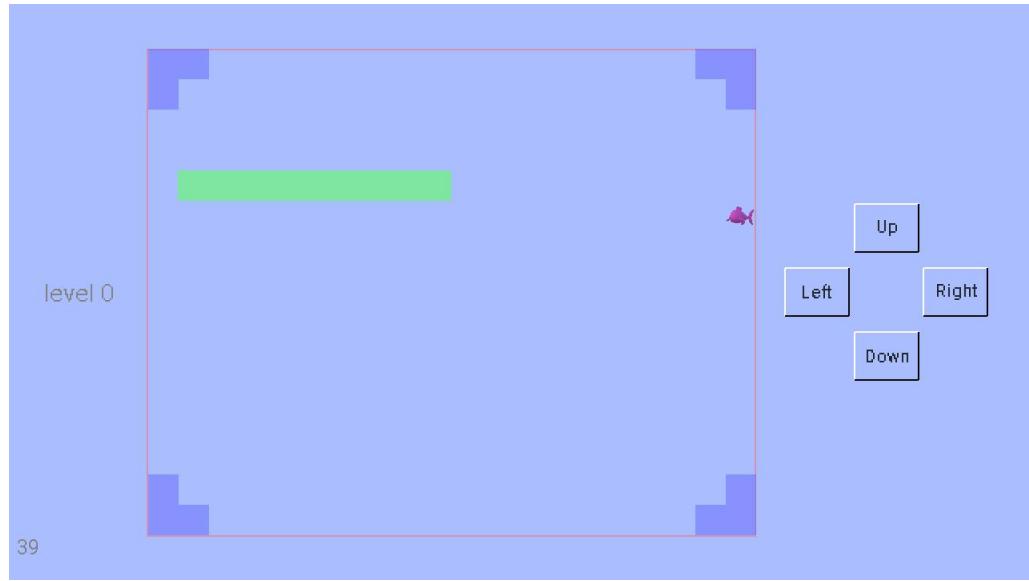


图 8.5：贪吃蛇游戏在安卓手机中运行的视频截图。

第 6 节 削宝石游戏简介

除了贪吃蛇游戏，可编程科学计算器还提供了一个削宝石的游戏的例子。削宝石游戏是一种非常流行的削削乐游戏，其在安卓上运行的游戏界面参见下图。



图 8.6：削宝石游戏在安卓手机中运行的视频截图。

其游戏规则是，MFP 生成一个 8 乘以 8 的游戏面板，面板中的每一个单元格内有一块宝石，玩家选择两块宝石互换位置，宝石换位之后，如果出现至少三个同样的宝石排成一行或者连成一列的情况，则换位成功，所有成串的不少于三个同样的宝石都被削去，MFP 发出成功削去宝石的声音。宝石削去后，上面的宝石落下填充空出来的单元格。如果宝石换位之后，没有出现至少三个同样的宝石排成一行或者连成一列的情况，则换位失败，MFP 发出失败的声音，换位的宝石再被换回来。

用 MFP 编写削宝石游戏的基本原理和贪吃蛇游戏类似，也是开启一个游戏显示屏幕，在游戏显示屏幕上绘制动画，根据用户输入事件（也就是点击了哪一块宝石），决定游戏的下一步动作，在生成 APK 安装包时用 MFP 标注语句将附属的声音和图像文件打入安装包等等。

编写这个游戏的难点，首先在于如何描述一块宝石。宝石有以下属性：第一是宝石的样子，是圆形的绿宝石，还是方形的蓝宝石，是六边形的粉红宝石还是菱形的橘红宝石，等等；第二是宝石在游戏面板上的行号（也就是 y）；第三是宝石在游戏面板上的列号（也就是 x）；第四是宝石的移动方向。对于这样的一个物件，最好的编程办法是面向对象，但是现在 MFP 语言还暂时不支持面向对象，所以，采取的办法是用字典或者数组的办法。如果使用字典的办法，通过基于字符串的 Key 可以找到相对应的值，比如，想要查询宝石的行号，可以调用 `get_value_from_abdict(gem_obj_dictionary, "x")`，这里 `gem_obj_dictionary` 就是用字典描述的宝石对象。

用字典的办法，比较直观，但是速度不够快。由于游戏对性能的要求高，所以更好的办法是直接采用数组。比如，用一个包含 4 个元素的数组表述宝石，第一个元素（索引是 0）是宝石的样式（也就是哪个图形对象），第二个元素是行号，第三个元素是列号，第四个元素是移动方向（基于字符串“up”，“down”，“left”或者“right”）。这样想要查询宝石的行号，直接用 `gem_obj_dictionary[1]`（表示数组的第二个元素），就可以了。削宝石游戏采用的就是这种编程办法。

编写这个游戏的另外一个难点，是如何有效读写宝石序列。注意到每一列的宝石落下的时候，是一串一串地落下的。一串宝石包括多于一个宝石。串和串之间有空白间隔。一般情况下一列只可能有一串宝石落下，但有时候一列也可能有两串，或者三串宝石一起落下。如果是多串宝石落下，第

一串宝石落到底后就不再下落了，第二串，第三串还在下落。在这种情况下，必须构造一个序列，序列中每一个元素是一串下落的宝石。一串宝石落到底后就不再下落，就等价于把数组中的第一个元素从序列中去掉，当序列变成空的时候，所有的宝石就都落地了。

为了方便开发者实现上述功能，MFP 提供了一整套新的基于数组的字典和序列函数如下。

函数名	函数帮助信息
append_elem_to_ablist	::mfp::data_struct::array_based::append_elem_to_ablist(2) : append_elem_to_ablist(array_based_list, ref_of_elem) 将 ref_of_elem 的引用添加到基于数组的序列 array_based_list 的尾部。它返回更新过的 array_based_list。作为参数的 array_based_list 不会发生改变。
concat_ablists	::mfp::data_struct::array_based::concat_ablists(2) : concat_ablists(list1, list2) 将基于数组的序列 list2 并入基于数组的序列 list1 的尾部并返回合并后的基于数组的序列。作为参数的 list1 和 list2 不会发生改变。
create_abdict	::mfp::data_struct::array_based::create_abdict(0) : create_abdict() 创建一个新的，基于数组的字典。
get_elem_from_ablist	::mfp::data_struct::array_based::get_elem_from_ablist(2) : get_elem_from_ablist(array_based_list, idx) 返回基于数组的序列 array_based_list 在 idx 处的值的引用。如果 idx 不合法，一个异常将会被抛出。

<code>get_value_from_abdict</code>	<code>::mfp::data_struct::array_based::get_value_from_abdict(2) :</code> <code>get_value_from_abdict(array_based_dictionary, key)</code> 返回基于数组的字典 <code>array_based_dictionary</code> 的 <code>key</code> 所对应的值的引用。如果 <code>key</code> 不存在，一个异常将将会被抛出。注意 <code>key</code> 只能是字符串。
<code>insert_elem_into_ablist</code>	<code>::mfp::data_struct::array_based::insert_elem_into_ablist(3) :</code> <code>insert_elem_into_ablist(array_based_list, idx, ref_of_elem)</code> 将 <code>ref_of_elem</code> 的引用插入基于数组的序列 <code>array_based_list</code> 的 <code>idx</code> 号元素之前并返回更新后的基于数组的序列。作为参数的 <code>array_based_list</code> 不会发生改变。注意 <code>idx</code> 必须是一个合法的索引。
<code>remove_elem_from_ablist</code>	<code>::mfp::data_struct::array_based::remove_elem_from_ablist(2) :</code> <code>remove_elem_from_ablist(array_based_list, idx)</code> 将基于数组的序列 <code>array_based_list</code> 的 <code>idx</code> 号元素删除并返回更新后的基于数组的序列。作为参数的 <code>array_based_list</code> 不会发生改变。注意 <code>idx</code> 必须是一个合法的索引。
<code>set_elem_in_ablist</code>	<code>::mfp::data_struct::array_based::set_elem_in_ablist(3) :</code> <code>set_elem_in_ablist(array_based_list, idx, ref_of_elem)</code> 将基于数组的序列 <code>array_based_list</code> 在 <code>idx</code> 处的值设置为 <code>ref_of_elem</code> 的引用。如果 <code>idx</code> 不合法，一个异常将将会被抛出。
<code>set_value_in_abdict</code>	<code>::mfp::data_struct::array_based::set_value_in_abdict(3) :</code>

set_value_in_abdict(array_based_dictionary, key, value) 将基于数组的字典 array_based_dictionary 的 key 所对应的值设置为 value 的引用并返回修改过的字典。如果 key 不存在，它将被创造出来。注意 key 只能是字符串但是 value 可以是任意数据类型。

需要注意的是，这些函数和已有的 MFP 数组操作函数不一样。这些函数向字典和序列设置或添加元素，或从字典和序列读取元素时，设置或添加或读取的是元素的引用。已有的 MFP 数组操作函数，设置或添加或读取的是元素的拷贝。显然，操作引用比操作拷贝更方便更快捷，但副作用是修改了引用的值，原始值也跟着改。但对于开发游戏来讲，这正是我们想要的一种类似于面向对象的编程，也就是数据无论放在哪里，都指向同一个对象。

编写这个游戏的第三个难点在于，游戏的绘图和计算量很大，MFP 现在的速度还不够快，如何在现有的条件下保证游戏能够顺畅运行。经过分析比较，我们发现，在计算机或手机屏幕上绘图是一个耗时大户。为了保证性能，需要尽可能少地在计算机或者手机的物理屏幕上绘图。为了实现这个目的，削宝石示例所采用的办法是，为整个游戏面板和面板网格的每一竖条都生成图形“显示窗口”，尽可能地把绘图事件都发送到这些图形“显示窗口”上，然后获取这些图形“显示窗口”的截屏，最后把截屏图形发给计算机或者手机的物理屏幕并在上面进行绘制。换句话说，把所有的绘图事件都并在一起，预先在一个图像对象上画出来，然后把该对象直接贴到图形“显示窗口”上。具体的代码见下：

A. 以下代码为每一列的单元空间生成图像“显示窗口”。

.....

```
// This function creates an image display for each column  
  
// 本函数为每一列单元格创建一个图像“显示窗口”  
  
variable thisColumnImgs = alloc_array([boardWidth])  
  
// initialize column image displays  
  
// 初始化每一列的图像“显示窗口”
```

```

    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. 以下代码将所有的绘图事件合并到一个图像“显示窗口”上，这样就得到了要绘制到物理屏幕上的图像。

```

.....
if and(boardCopy[x][y + 1] == emptySpace, boardCopy[x][y] != emptySpace)

// The gem in this space drops

// 这个单元格的宝石会落下

if y + 1 < yLast // no gem immediately below this gem is falling // 这个宝石的下面没有宝石

// start a new bunch, not add dropping direction because it is default.

// 开启一个新的宝石串，不用定义新的宝石串的运动方向，因为缺省就是下落

droppingGems[x] = append_elem_to_ablist(droppingGems[x],
[[boardCopy[x][y], x, y]])

else // a gem immediately below this gem is falling // 这个宝石的下面还有宝石也在往下掉

// append this gem to existing bunch // 将这个宝石加在已有的宝石串中

```

```

                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. 以下代码将生成的图像“显示窗口”截图直接贴到显示窗口上：

```

.....
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)

.....

```

削宝石游戏完整的代码和注释可在本手册自带的示例代码所在目录中的 2d games 子目录中 gemgem 子目录下的 gemgem.mfps 文件中找到。

小结

用 MFP 语言开发游戏，真正实现了脚本一次编写，跨平台运行。大大方便了开发人员编写和调试，提高了生产力。

要掌握 MFP 语言的游戏开发，首先必须掌握 MFP 语言开发游戏的编程流程，也就是开启游戏运行窗口→读取并处理用户输入→绘制动画→再次读取并处理用户输入→绘制动画……，往复循环。绘制动画有两种方式，第一种是直接在物理屏幕上绘制图形元素比如点，线，圆等等，这需要对物理屏幕的显存进行频繁读写，速度比较慢；第二种是将图形元素预先绘制到基于图像对象的“显示屏幕”上，绘制完成后再将整个图像贴到屏幕上，这只需要对物理屏幕的显存进行一次操作，速度更快。第一种方式适合简单的，或者静止的图像，第二种方式适合绘制复杂的动画。

其次，必须理解 MFP 语言开发游戏生成动画的原理。调用 MFP 作图函数，比如 draw_rect，draw_image 之类的函数，并非马上开始在物理屏幕或者图像对象的“显示屏幕”上绘图，而是将绘图请求事件放入一个调度器中，由调度器调用该事件绘图。开发者无法知道调度器会何时自己调用绘图事件。但是开发者可以调用 update_screen 函数要求调度器立即调用所有的绘图事件。动画效果，本质上是把调度器中旧的绘图事件清除，然后加入新的绘图事件，再调用 update_screen 函数。这样每次绘制出来的图像都和上一次不一样，看起来就像是动了起来。

第三，MFP 语言现在还无法支持面向对象，所以描绘物体时必须用字典或数组的方式。MFP 已经提供了一组新的函数去读取，修改字典或者数组序列。

第四，为了将游戏脚本编译为安卓 APK 包，必须使用 build_asset 标注，将附属的声音和图像文件拷贝到 APK 的 asset 下的 resource.zip 文件中。

详细介绍了 MFP 编写游戏的原理之后，现在最重要的，就是用户自己动手，开发一个小游戏，体验一下 MFP 语言的强大威力。

第9章 构建用户自己的应用

可编程科学计算器的一项重要的功能就是能够用任何 MFP 函数（包括用户自己开发的函数和系统提供的函数）生成一个独立的安卓应用，并安装和发布。把 MFP 函数打包为应用有几个好处：

1. MFP 函数打包生成的应用体积很小，却能够执行全部的功能，不会占据太多的存储空间；
2. MFP 开发人员不必再绞尽脑汁教其他人如何使用可编程科学计算器和启动 MFP 函数，直接发布或者共享 APP 安装包即可；
3. MFP 函数打包时，将进行编译，去除不使用的函数，所以，装载速度更快。

用户创建自己的应用非常简单，即可以在手机上点击“创建 MFP 应用”图标，也可以在电脑上运行可编程科学计算器，点击“工具”菜单，选择“创建 MFP 应用”子菜单，出现的界面如下：



图 9.1： 创建 MFP 应用第一步。

用户只需要按照提示输入相应的内容即可。这里需要注意的是以下几处：

1. 应用名是出现在手机应用列表主屏幕上位于图标下方的名字，这个名字也在应用管理列表中使用；
2. 应用包 ID 用于区别您的应用和别人的应用，所以必须独一无二。需要注意，在第 7 章第 2 节中，`system` 函数启动一个应用也需要知道应用包的 ID；
3. 应用在 SD 卡上的工作目录是用户生成的 App 启动的时候的初始目录，用户生成的图形，以及使用相对目录时产生的文件都保存在这个目录中。如果用户不设定该目录的名字，该目录将自动设置为用户的包 Id 最后一个点的后面的部分所构成的字符串。

用户输入完成之后，点击继续，则进入第二步，选择要打包的函数。参见下图：



图 9.2: 创建 MFP 应用第二步。

如果用户没有选中“使用可选参数”的选择框（红色的方框），那么，用户要添加的参数和用户选择要打包的函数所需要的参数的个数必须吻合。用户必须定义每一个参数，“关于该参数的信息”输入行（绿色方框）用于输入参数内容提示，“该参数的默认值”输入行（蓝色方框）用户用于在用户的用户（用户生成的 App 的使用者）不输入的情况下，确定参数的缺省值，“参数是一个字符串”选择框（紫色方框）如果被选中，那么用户的用户输入时，就不用输入界定字符串的双引号，“参数无需用户输入”选择框（黄色方框）如果被选中，用户的用户将看不到这个参数的输入框，App 运行时将自动使用该参数的缺省值。

如果用户选中了“使用可选参数”的选择框，最后一个参数就是可选参数，该参数不能够定义默认值（定义了也没有用），而且用户的用户也必须输入，在 App 启动以后用户的用户可以在最后一个参数输入框中输入多行，每一行表示一个参数。如果可选参数的“参数是一个字符串”选择框（紫色方框）被选中，那么用户的用户在输入所有可选参数时，都不用使用双引号来界定字符串，否则，所有可选参数中字符串的双引号都不能省略。

需要注意的是，当创建一个 APK 包的时候，可编程科学计算器不是拷贝所有用户自定义的 mfps 代码文件而仅仅抽取相关的代码。在某些时候，比如调用 integrate 或者 plot_exprs 函数时，函数参数是一个字符串或者基于字符串的变量。这样一来，可编程科学计算器在编译的时候无法判断哪些函数在运行时将会被调用。用户在这种情况下需要在代码中，最好在 integrate 或者 plot_exprs 函数调用语句的前一行，增加一个标注指令 @compulsory_link 告知可编程科学计算器哪些用户自定义的函数需要链接入 APK 包。比如

```
...
@compulsory_link
function(::mfpexample::expr1, ::mfpexample::expr2(2))

integrated_result = integrate(expression_str, variable_str)
...
```

在上面例子中，`::mfpxexample::expr1` 和 `::mfpxexample::expr2` 是用户自定义的函数。它们在运行时将会被 `integrate` 函数调用用于计算积分。所有的名字叫做`::mfpxexample::expr1` 的函数都会被链接入 APK 包。但是对于名字叫`::mfpxexample::expr2` 的函数，当且仅当它正好有两个参数或者有可选参数时，该函数才会被链接入 APK 包。

如果用户想把所有自己定义的函数链接入 APK 包，请用如下语句

```
@compulsory_link all_functions
```

这样一来，可编程科学计算器将链接所有函数。但是用户创建的应用装载速度会比较慢。并且，用户还必须保证所有的函数都必须已经定义，否则打包时会出现编译错误。

还要注意`@compulsory_link` 指令必须位于一个函数的内部，如果它在 `function` 语句前面或者 `endf` 语句之后，它不会有任何作用。

创建 App 的最后一步是设定 APK 包的文件名并签名，签名的关键是要选择密匙，用户可以选择测试密匙，这样用户就不用输入密码和个人信息，但是，用测试密匙签名的 APK 包，用户只能自己安装或者发送给朋友安装，不能在网上发布，如果用户自己创建一个新的 APK 密匙，则需要输入存储密匙文件密码和密匙密码，以及个人信息。用户创建的 APK 密匙，可以以后重复使用，但最好一个密匙对应一个 App（版本可以不同）。创建 App 的最后一步参见下图：



图 9.3： 创建 MFP 应用第三步。

上述步骤完成之后，用户点击确定，可编程科学计算器就开始创建自己的应用，如果一切顺利，将会出现如下对话框。用户可以选择安装或者共享自己创建的应用，或者仅仅记下 APK 文件的保存路径，然后点击确定回到可编程科学计算器的主屏幕。



图 9.4：成功创建 MFP 应用。

需要注意的是，如果用户选择安装创建的 App，有可能会收到提示，询问用户是否允许安装并非从官方渠道获得的 App，这时用户需要在安卓系统的设置里面激活“允许安装来源未知的应用”，安装才能继续。

如果用户想把自己的应用发布在网上，则需要选择在各大应用发布网站注册，然后提交创建的 APK。注意，由于用可编程科学计算器创建的 APK 不可避免地具有相似性，用户提交 APK 后，安卓应用的发布网站可能会询问用户该应用是否违反知识产权相关的法律，用户只需进行解释，这不会影响到应用的成功发布。

第 10 章 在你自己的应用中使用 MFP 安卓库

众所周知，可编程科学计算器是基于 MFP 编程语言的强大工具。MFP 语言是一种面向对象的脚本语言，它提供了丰富的函数，可以用于二维游戏开发，复数计算，矩阵计算，高价积分，二维三维和极坐标图像绘制，字符串，文件操作，JSON 数据读写以及基于 TCP 和 WebRTC 协议的通信编程。基于 Apache 2.0 许可，MFP 现在已经开放源代码。所以任何个人或公司均可利用这种编程语言。显然，如果能够将这种编程语言整合进自己的应用，开发人员可以节省大量的时间和资源。

MFP 安卓库代码可以在 <https://github.com/woshiwpa/MFPAndroLib> 下载。并且，从 2.1.1 版开始，可编程科学计算器开始提供二进制 MFP 安卓库文件。每次用户升级到新版后，可编程科学计算器会自动将最新版的 MFP 安卓库拷贝到安卓设备内存的 Android/data/com.cyzapps.AnMath/files/AnMath 目录中。用户也可以在启动可编程科学计算器后点击“在电脑上运行”图标，手动拷贝 MFP 安卓库文件。

将 MFP 库嵌入到 Android 项目中需要两个 aar 二进制文件。一个是 MFPAnLib-release.aar。另一个是 google-webrtc-x.x.xxxxx.aar。请将它们从安卓设备内存的 AnMath 文件夹复制到目标 Android 项目中。开发人员可以将它们放置在任何地方，只要 gradle 可以找到它们就没有问题。假设它们保存在名为 app 的模块的 libs 文件夹中。在这种情况下，开发人员需要在应用模块的 build.gradle 文件中添加以下两行：

```
// google-webrtc aar 在未来可能会发生变化  
implementation files('libs/google-webrtc-1.0.19742.aar')  
implementation files('libs/MFPAnLib-release.aar')
```

除了两个二进制文件之外，MFP 还有一些预定义的脚本程序。这些脚本程序为开发人员提供了许多有用的功能。在可编程科学计算器中，预定义的脚本被压缩在安卓设备内存的 Android/data/com.cyzapps.AnMath/files/AnMath 文件夹内的 assets.7z 文件中。在 assets.7z 内有一个名为 predef_lib 的文件夹。开发者应将整个 predef_lib 文件夹复制到开发者自己的项目的 assets 文件夹中。

开发者的应用的 Application 的实现的 onCreate 函数应该包括以下代码：

```
public class YourAppImplementationClass extends  
    androidx.multidex.MultiDexApplication {  
  
    @Override  
  
    public void onCreate() {  
  
        super.onCreate();  
  
        ... ...  
  
        MFPAndroidLib mfpLib = MFPAndroidLib.getInstance();  
  
        // initialize 函数有三个参数。第一个参数是一个 Application  
        Context; 第二个参数是你的 App  
  
        // 的共享的设置 (shared preference) 的名字, 最后一个是一个布尔  
        值, true 表示你的 MFP 代码和  
  
        // 资源文件均保存至你的应用的 assets 中, false 表示你的 MFP 代码和  
        资源文件保存在安卓设备  
  
        // 本地内存中。  
  
        // 以下代码适用于将 MFP 代码和资源文件保存在应用的 assets 中的情  
        况。如果你想把 MFP 代码和资  
  
        // 源文件保存在安卓设备的本地存储中, 请放回下面一行已经注释掉的  
        代码并传递 false 给  
  
        // initialize 函数的第三个参数。  
  
        // MFP4AndroidFileMan.msstrAppFolder = "My_App_MFP_Folder";  
  
        mfpLib.initialize(this, "Your_App_Settings", true);  
  
        MFP4AndroidFileMan mfp4AnFileMan = new  
        MFP4AndroidFileMan(getAssets());  
  
        // 平台的硬件管理器必须最先被初始化。这是因为它需要在装载代码时  
        用于分析代码中的标注。  
  
        // 其它的解释器运行环境管理器可以在运行程序之前再初始化。
```

```

        FuncEvaluator.msPlatformHWMgr = new
PlatformHWManager(mfp4AnFileMan);

        MFPAdapter.clear(CitingSpaceDefinition.CheckMFPSLibMode.CHECK_EVERYTHING);

        // 在启动时先装载预定义的 MFP 脚本

        mfp4AnFileMan.loadPredefLibs();

    }
    ...

```

如上所述，有两种方法可以保存用户定义的 MFP 脚本和相关资源文件。一个是保存在模块的 assets 中。在这种情况下，开发人员必须在模块的 assets 中创建一个名为 userdef_lib.zip 的压缩文件。此压缩包内有一个名为 scripts 的文件夹。所有用户定义的 MFP 脚本都在其中。

当开发人员编写 MFP 脚本时可能需要装载一些资源，比如图像或者声音。在这种情况下开发人员需要创建另外一个名为 resource.zip 的压缩包。该压缩包也放在 assets 中，资源文件保存至该压缩包内。

以下代码展示了正确调用资源文件的完整方法。

```

@build_asset copy_to_resource(if(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
```

上述代码的意思是，if ... elseif ... else ... endif 程序块告诉 MFP 当前脚本需要装载一个叫做 drop2death.wav 的声音文件。如果当前代码是在一个沙盒中运行，该 wav 文件被保存至沙盒会话的 sounds 文件夹中。注意这里的沙盒是 MFP 并行计算的术语，意思是远端的 MFP 实例发送到本地运行的 call 程序块。如果当前代码在 MFP 应用中运行（这也是当前示例的情况），则 wav 文件将保存在应用 assets.zip 文件的 sounds 文件夹中。请注意，函数 play_sound_from_zip 的第三个参数是 1，这意味着函数 get_asset_file_path 返回安卓应用的 assets 路径。如果当前代码在本地设备内存中运行，例如 SD 卡或 PC 的硬盘，则 wav 文件将与脚本放在同一文件夹中。

if 程序块上方的@build_asset 标注则是告诉 MFP，如果需要编译 MFP 应用，或者在远端沙盒中运行一个 call 程序块，资源文件应该被拷贝保存至目标侧的那个位置。MFP 安卓库不包括将 MFP 脚本编译成 MFP 应用的功能，但如果开发人员需要远程运行一些代码，上述标注还是必不可少的。

当然，如果完全没必要向远端发送代码，则用一行语句取代上述标注和 if 程序块就足够了：

```
play_sound_from_zip(get_asset_file_path("resource"), "sounds/drop2death.wav", 1,  
false)
```

如果开发人员将所有自己定义的 MFP 脚本放在安卓设备的本地内存而不是 assets 中，则需要告知 MFP 文件夹的位置。MFP 文件夹位于 Android/data/your.app.package.id/files/ 目录中。脚本应放置在 MFP 文件夹的 scripts 子文件夹中。此外，如果没有必要将代码发送到远端执行，则只需一行 MFP 语句即可加载资源。对于上面的示例，代码行应为

```
play_sound(get_upper_level_path(get_src_file_path()) + "drop2death.wav", false)
```

如以下截屏所示，在 MFP 安卓库的 github 代码项目中，示例应用的 assets 文件夹包含以下项：predef_lib、resource.zip、userdef_lib.zip、InternalFuncInfo.txt 和 MFPKeyWordsInfo.txt。如上所述，如果开发人员

决定将自定义的脚本放在安卓设备本地存储中，则不需要 resource.zip 和 userdef_lib.zip。此外，InternalFuncInfo.txt 和 MFPKeyWordsInfo.txt 包含 MFP 预定义函数和关键字的帮助信息。开发人员通常无需用到它们。

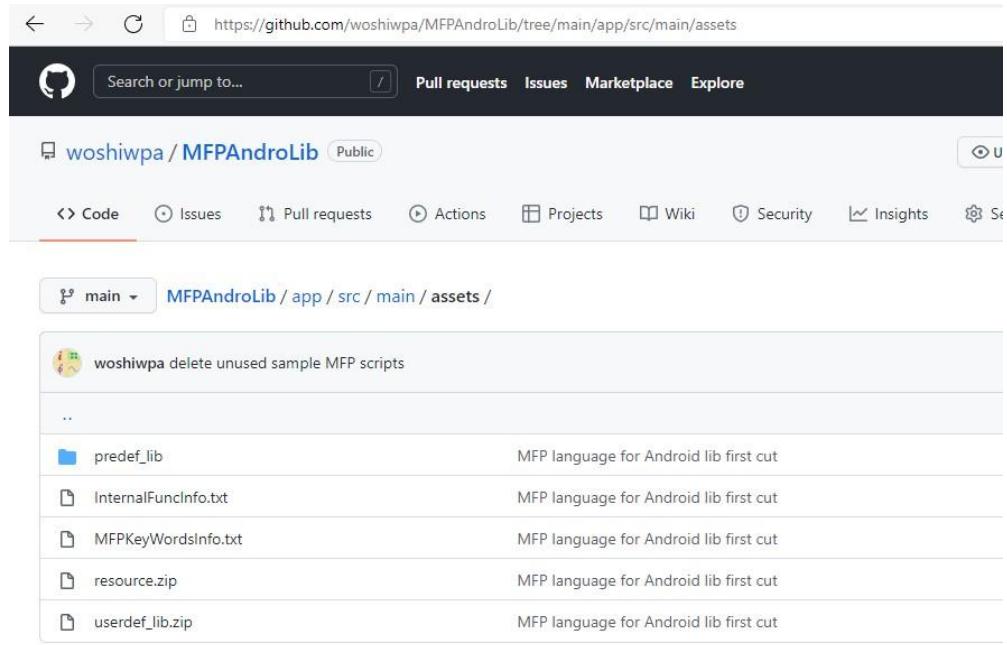


图 10.1：示例应用 assets 文件夹内容。

将所有二进制文件、MFP 脚本和资源复制到正确的位置后，开发人员需要加载自定义的 MFP 脚本。如果脚本保存在应用程序 assets 中，请调用 MFP4AndroidFileManager.loadZippedUsrDefLib 函数来加载脚本。否则，应该调用 MFP4AndroidFileManager.reloadAllUsrLibs 来完成这项工作：

```
// 现在开始装载函数  
  
MFP4AndroidFileManager mfp4AnFileMan = new MFP4AndroidFileManager(am);  
  
// 如果需要重复运行这个函数，我们必须调用 clear 函数以保持 MFP 引用空间  
// (citing space) 的洁净。  
  
// 但是，如果我们只运行本函数一次，MFP 引用空间肯定是处于洁净的初始状态，所以，  
// 下面一行代码  
  
// 无需被调用。
```

```
//  
MFPAdapter.clear(CitingSpaceDefinition.CheckMFPSLibMode.CHECK_USER_DEFINED_ONLY);  
  
// 调用用户自定义的函数。  
  
if (mfp4AnFileMan.isMFPAppl()) {  
  
    MFP4AndroidFileMan.loadZippedUsrDefLib(MFP4AndroidFileMan.STRING_ASSET_US  
ER_SCRIPT_LIB_ZIP, mfp4AnFileMan);  
  
} else {  
  
    // 如果你的 MFP 脚本保存在安卓设备的本地内存中，则使用以下代码载入自定义  
    // 的函数。  
  
    MFP4AndroidFileMan.reloadAllUsrLibs(ActivityAnMFPMain.this, -1, null);  
  
}
```

在运行 MFP 代码前的最后一步是初始化 MFP 解释器的运行环境。注意这一步不能够和 MFP 库的 initialize 函数合并，这是因为 MFP 库的 initialize 函数需要的是应用（Application）的 Context 而这里需要的是活动（Activity）的 Context。

```
// 现在初始化 MFP 解释器的运行环境  
  
MFPAndroidLib.getInstance().initializeMFPInterpreterEnv(ActivityAnMFPMain.this,  
new CmdLineConsoleInput(), new CmdLineLogOutput());
```

请注意，除了 Activity 的 Context 之外，开发人员还需要将 CmdLineConsoleInput 和 CmdLineLogOutput 传递到 initializeMFPInterpreterEnv 函数中。CmdLineConsoleInput 和 CmdLineLogOutput 分别派生自 MFP 安卓库的抽象类 ConsoleInputStream 和 LogOutputStream，它们告诉 MFP 如何从应用程序中读取 MFP 的 input 和 scanf 函数的输入，以及如何在应用程序中显示 MFP 的打印输出字符串。因此，两个类如何实现完全取决于开发人员。例如，开发人员可能希望丢弃所有输出，则可以实现一个不做任何事的 outputString 函数。如果自定义的 MFP 脚本从未调用过任何 MFP 的输入函数，开发人员甚至可以只是在 CmdLineConsoleInput 的 inputString 函数中引发异常，尽管这是一种强烈不推荐的编程方式。github 中的 MFP 安卓库源代码项目已经为这两个类提供了实现示例。

现在万事俱备，开发人员可以定义要运行的 MFP 语句了。所有的 MFP 语句都保存至 JAVA 字符串中，用断行字符（’\n’）分隔彼此。比如，“\n\nplot_exprs(\"x**2+y**2+z**2==9\")\ngdi_test::game_test::super_bunny::run()\n”就包含两行 MFP 语句。第一行调用 plot_exprs 函数，第二行运行一个超级小白兔的游戏。

取决于语句数目，开发人员需要调用 MFPAutoLib. processMFPStatement 或者 MFPAutoLib. processMFPSession 来解释 MFP 代码。这两个函数的返回值均为一个字符串。如果单行 MFP 代码不返回任何值，或者多行 MFP 会话没有调用 return 语句返回某个值，并且程序运行时也没有异常抛出，这两个函数的返回值为空字符串。否则，这两个函数的返回字符串为 MFP 的返回值或者抛出的异常的栈信息。varAns 是用于存放原始返回值的变量。如果 MFP 代码不返回任何值，varAns 则保持其原始值，也就是 MFP null，不变。varAns 对于开发人员来说非常重要，因为它保存了 MFP 返回值的原始类型信息。

```
String[] statements = str2Run.trim().split("\\\\n");

String strOutput;

Variable varAns = new Variable("ans", new DataClassNull()); // 这个变量保存了返回值

if (statements.length == 1) { // 运行单行 MFP 语句

    strOutput = MFPAutoLib.processMFPStatement(str2Run.trim(), new
LinkedList(), varAns);

} else { // 运行多行 MFP 语句

    strOutput = MFPAutoLib.processMFPSession(statements, new LinkedList(),
varAns);

} // 如果开发人员不想在应用中显示 MFP 返回的字符串，则下面一行代码是多余的

new CmdLineLogOutput().outputString(strOutput);
```

要了解上述代码的实现详细信息，请转到 [github](#)，下载 MFP 安卓库项目并亲自动手实践吧。

后记

可编程科学计算器是一个功能非常强大的安卓应用，相信大家如果能够耐心读完这本手册，一定能够找到很多其他的应用无法实现的功能，特别是其绘制三维图像和编程的功能。绘制三维图像甚至可以在一定程度上代替被学术和工业界广泛使用的 gnuplot 软件，而编程的功能是如此的强大，不但完全可以代替现在市面上贩售的任何基于硬件的可编程计算器，甚至还可以开发游戏。

但是，不能不遗憾地指出，现在可编程科学计算器的使用还不够广泛，原因主要在于：

1. 可编程科学计算器的界面还不够新颖别致。老式的图标和对话框让人觉得单调乏味；
2. 可编程科学计算器在没有详细说明的情况下并不好用，除非用户本身就是程序员，否则不可避免地会对各种各样复杂的操作和输入感到头晕；
3. 可编程科学计算器缺乏推广，现阶段虽然有几十万下载量，但考虑到一来很多下载只是对旧版本的升级，以及中国上千万大学生和工程技术人员群体，这样的下载量和使用频率不是让人满意的。

我作为可编程科学计算器的开发者，完全明白上述三点关系着这个软件的生存和死亡。我在百度创建了 [MFP 吧](#)，还有不知名的用户创建了 [可编程科学计算器吧](#)，这两个贴吧作为用户的讨论群，可以方便交流，互通有无。这也算是一种推广的方式。

此外，我停止所有的开发，专门花费几个月的时间，撰写这本详细的用户手册。我希望这本手册能够解答用户的大部分问题，甚至对没有编程基础的用户，这本手册也能够起到对用户的计算机编程的启蒙作用。MFP 编程语言和 Basic 语言其实挺像的。但是，MFP 有一些 Basic 语言不具备的东西比如 3D 绘图和生成自己的应用，这些东西对不会编程的用户来讲，是一种学习的奖励，或许这更能够激发用户学习的兴趣。

如果用户阅读了手册之后还有问题，可以到百度 MFP 吧或者可编程科学计算器吧提问，百度 MFP 吧似乎人气更高一些，我也在 MFP 吧中长期驻守，对所有的用户，耐心答疑。还有很多其他的热心用户会给予帮助。

最后就是可编程科学计算器界面的问题。这其实对于程序员来讲，并没有一个简单的解决方案。图标的绘制需要专门的软件，价格也不会很便宜，特别是对于安卓应用，在不同的屏幕上需要不同大小的图标，没有专门的软件，一个一个图标调整起来非常麻烦，而漂亮的安卓界面库，第一难找，第二，即便有，也往往不是免费的，作为个人开发者，在时间和金钱上都是难以支持的。但是，我还是会逐步对界面进行调整，尽可能让软件满足用户的审美观。

最后一点，关于软件的推广，我个人力量有限，只能通过建立用户交流群和个人主页的办法，吸引更多的人来使用，在这里，我希望在用户阅读了本手册，对这个软件和这种编程语言有了更多更详细地了解之后，能够口耳相传，向自己身边的朋友，同学，同事推荐，在这里拜托大家了。

纵观安卓计算器软件的发展，到现在，已经基本上到头了，再怎么创新，也都是细节上的完善和一些小的计算功能的增加，但是，MFP 作为一种编程语言，特别是和安卓紧密结合的脚本编程语言，前途是无量的，MFP 现在只能操作文件和绘制数学图像，但如果它能够操作浏览器，email，USB，蓝牙，短信，能够开发安卓的图形界面，则应用范围不可限量。我作为 MFP 的创造者，对 MFP 的发展，包括如何实现对安卓的全面操控，如何实现面向对象，如何实现对并行的支持，有着一整套规划和思路，有很多想法，我个人认为都是非常新颖的，如果能够实现，对于计算机编程语言来讲，都是很大的突破，但是，作为个人，要实现所有这些想法，就好比愚公移山，不知道要多久才能做到。但我会坚持下去，我也希望，能有志同道合的人，不论是程序员还是风险投资人，能够支持我，我欢迎所有的人给我来信，我的 email 是 cyzsoft@gmail.com。