



组 号 Z08

实验题目 使用 MatLab 制作简易计算器

1. 宋昭琰

队员姓名 2. 徐意

3. 李宗琳

4. 向飞宇

目 录

1 准备工作	2
1.1 GUIDE 介绍及工作流程	2
1.2 数据的传递形式	2
1.2.1 程序支持的数据类型	2
1.2.2 数据类型的相互转换	3
1.2.3 表达式的生成与计算	4
1.2.4 计算器的记忆功能	5
1.3 指令函数化	6
1.4 分辨率调整与 GUI 显示问题	6
2 模块设计	7
2.1 基本计算模块	7
2.2 矩阵模块	8
2.3 函数模块	9
2.3.1 函数定义的转换	9
2.3.2 函数图像绘制	9
2.3.3 函数的积分和微分	10
2.4 绘图模块	11
2.4.1 对绘图函数的分类	11
2.4.2 绘图函数的设计	11
2.4.3 其它绘图功能	12
2.5 数据存储模块	13
2.5.1 哈希表与元胞数组的双表查询与修改	13
2.5.2 用赋值方法将存储元素用于计算	15
2.6 其它模块	16
2.6.1 欢迎界面设计	16
2.6.2 保存界面设计	16
附录 A 部分实验代码	17
A.1 表达式的计算与数据赋值	17
A.2 绘图模块的函数封装	18
A.3 函数模块的函数封装	20
A.4 数据存储模块的双表查询	21

1 准备工作

1.1 GUIDE 介绍及工作流程

在计算器的制作过程中, 我们使用 MatLab 内置的 GUIDE 编辑器制作计算器的基本框架. GUIDE 中提供了多种控件的类型, 制作计算器需要可编辑文本框, 不可编辑文本框和按钮, 因为在本次实验中需要的按钮较多, 我们创建了几个面板来放置这些按钮, 并根据这些按钮讲计算器的设计分为了六个模块: 基本计算模块, 矩阵模块, 函数模块, 绘图模块, 数据存储模块和其它模块, 这些模块将在下文中具体解释. 同时我们需要一个输入框来输入运算表达式和一个输出框来得到运算结果. 最后, 我们需要一个画图窗口 (axes) 来显示绘制图像的结果. 根据以上的设计思路, 我们得出了以下的工作流程:

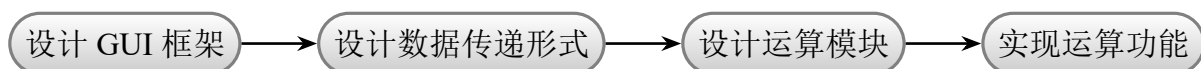


图 1 工作流程

1.2 数据的传递形式

设计数据的传递形式是实现计算器各项功能的基础, 它能够极大的简化模块设计的复杂度. 而我们经过讨论认为, 一个好的数据传递形式需要具备以下特征:

- **清晰的数据类型:** 每一个变量需要给出**唯一**的数据类型;
- **令人理解的数据形式:** 每种显示或输入的数据类型容易被用户理解和使用;
- **统一的数据计算方法:** 对于用户给出的指令, 需要使用**相同**的计算方法实现;
- **统一的数据可视化:** 对于每一种待显示的数据类型能够使用**相同**的显示方式呈现. 放在 MatLab 中, 就是需要将最终的计算结果使用字符串表示出来.

这四条特征中, 第一、三条是针对代码设计的可重用性和便捷性提出的, 第二、四条是针对用户体验提出的, 以下将给出基于这四条特征设计的计算机程序.

1.2.1 程序支持的数据类型

MatLab 中有很多定义的数据类型, 如 “double”, “(u)int8,16,32”, “function_handle” 等等, 在程序设计中, 因为 MatLab 对不同数据类型的设计难度不同, 我们分离了数据的传输与数据的显示, 结合我们刚刚提出的设计模块得出: 基本计算模块与矩阵模块需要使用 “double” 类型 (因为矩阵可以看成一个 “double” 数组, 数据类型依然是 “double”), 在基本计算模块, 实现数据显示功能需要将数值转化为字符串; 在矩阵模块, 实现数据显示功能需要将数值 (矩阵) 转化为字符串. 由于 MatLab 处理匿名函数与函数句柄比处理字符串和符号变量容易, 函数模块中的数据使用 “function_handle” 类型而不使

用“sym”类型; 由于 MatLab 显示函数句柄没有显示一个以符号变量为字符串的函数美观, 函数模块中的显示使用“sym”类型而不使用“function_handle”类型. 绘图模块主要是对绘图区域的操作, 所以使用“Figure”类型, 以便于能在绘图区上绘制即可. 数据存储模块是存储各种数据的模块, 为了与之前的模块支持的类型一致, 数据项只支持“double”和“function_handle”, 函数句柄显示时将转换为更美观的“sym”类型. 因为其它模块不涉及数据框的输入与输出, 它们不需要支持任何数据类型. 综上所述, 程序支持的数据类型如下表所示:

模块名称	数据传输的类型	数据显示的类型
基本计算模块	double	double→string
矩阵模块	double	double→string
函数模块	function_handle	sym→string
绘图模块	-	-
数据存储模块	double, function_handle	double→string, sym→string
其它模块	-	-

表 1 程序支持的数据类型, 其中“→”表示数据转换方向

1.2.2 数据类型的相互转换

如上一节所述, 由于数据传输与数据显示的区别, 我们需要对数据类型进行相互转换, 查阅 MatLab 相关文档后, 我们得出了以下的数据类型转换表实现数据类型的相互转换:

原数据类型	转换后数据类型	转换函数
double	string	mat2str()
function_handle	sym	sym()
sym	function_handle	matlabFunction()
sym	string	char()

表 2 数据类型的相互转换

1.2.3 表达式的生成与计算

为了实现统一的数据计算方法, 我们对于表达式的构造使用了统一的方式. 由于 MatLab 的 GUI 中允许将程序的部分数据存储在 handles 结构体中, 并通过 `guidata(hObject, handles)` 的方式来更新数据, 我们将计算的表达式作为一个字符串变量存储下来, 当用户点击某个按钮时, 将该按钮所代表的字符串粘连在原表达式的后面, 这样就能够持续的构造表达式. 但是这样操作会存在一个弊端, 就是光标的位置不好控制. 因为没有找到控制输入框光标的相关程序接口, 我们删除了输入框中的光标, 并使之无法编辑, 这导致用户之前的错误输入行为将需要通过持续删除表达式的项来修改, 效率较低. 为了提高效率, 我们设计了一个记忆功能, 使得用户一次可以删除多个字符, 这将在下一节中进行讨论. 表达式的生成原理如下图所示:

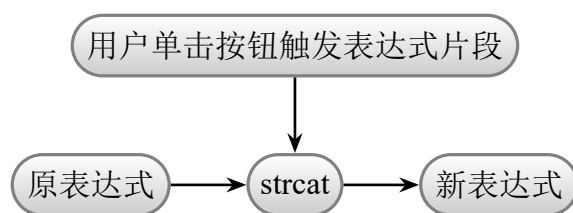


图 2 表达式的生成原理

表达式的计算是计算器中的核心部分, 这一部分中主要由异常处理, 表达式计算和数据分类输出构成. 以下将分别介绍这些部分.

异常处理:

异常处理是程序设计的重要组成部分, 通过异常处理能够针对程序给出的异常行为给用户相应的提示. 异常处理有两种类型: 面向过程的异常处理和面向对象的异常处理. 面向过程的异常处理是针对每个异常设置一个条件分支, 根据不同的条件分支跳转程序来处理异常, 这样处理使得程序的可读性降低; 而面向对象的异常处理是先运行语句, 针对不同的错误类型抛出相应异常, 再使用异常接收语句处理抛出的异常, 这样就使得程序条理清晰, 且对用户友好. MatLab 是基于 Java 的语言, 而 Java 语言是面向对象的, 所以 MatLab 的异常处理也是面向对象类型的. 具体而言, MatLab 中的异常被抽象为 `MException` 对象, 这个对象中包含错误代码, 错误信息, 错误时的堆栈状态等信息. 与 Java 不同, MatLab 的所有异常都是实时的, 任何异常都会导致报错, 但是报错窗口在命令行中, 对用户不友好, 针对这个问题, 我们重写了一个异常处理:

```
try
    ...
catch ME
    errordlg(ME.message, ME.identifier, 'modal');
end
```

这种方式将异常转化到一个模态对话框中显示, 使用户清楚自己输入错误的问题来源, 从而对用户友好.

表达式计算:

由于 MatLab 中提供了 `eval` 函数来求解一个字符串表达式的值, 对表达式的计算较为容易, 只需要 `eval(handles.command)` 就可以解决表达式计算的问题, 同时 `eval` 将会把所计算出的数值结果作为一个变量输出, 我们所要做的就是针对这个数值的不同的数据类型输出不同的结果.

数据分类输出:

数据类型的转换已经在表 (2) 中给出, 此处只说明如何对数据分类, 这里采用 MatLab 自带的 `isa` 函数, 判断数值是否属于 “double”, “function_handle”, “sym” 类, 根据不同类别分别加以转换然后通过 `set` 函数输出到结果框即可向用户展示所得答案.

1.2.4 计算器的记忆功能

这一部分中主要说明计算器在存储结果和存储之前的命令方面的记忆功能, 关于数据存储功能将在 “数据存储模块” 一节中讨论.

存储结果:

存储结果的实现方式与存储当前的命令的实现方式一样, 都是通过使用一个变量来存储结果. 只不过, 为了方便存储, 我们利用了 MatLab 中自动存储表达式计算结果的 `ans` 变量来存储结果, 这样做的好处在于: `ans` 是一个无论何时都可以使用在表达式的变量, 所以可以使用在表达式构造中. 本次实验中, 我们构造了一个 `ans` 按钮, 每次单击会将 “ans” 字符串至于表达式的末尾, 以便于再次计算. 同时, 在每次计算完成之后, 输入框中的变量永远是 `ans`, 这样能够更好地利用计算所得的结果. 最后, `ans` 变量被用于数据存储的交互, 提高了数据存储的效率.

虽然利用一个已有的变量非常容易, 但是仍然需要严谨地处理一个问题: 清空屏幕与清空结果的区别问题. 在清空屏幕时, `ans` 的值不会发生改变; 而在清空结果时, `ans` 的值需要为 0.

此外, 当一个作图命令给出时, `ans` 的值为 0, 因为图像不能应用于任何数据处理中.

存储之前的命令:

设计这个功能源于删除字符时的效率问题, 在之前的几个版本中, 我们设计删除字符使用了下面的方法 (即统计出指令的长度后, 再删除最后一个字母):

```
handles.command = eraseBetween(handles.command,  
    ↳ length(handles.command), length(handles.command));
```

这样作效率太低, 而且对于一些较长的命令而言, 删除字符会非常麻烦. 所以我们改进了删除字符的机制:

- 首先, 使用一个单独的变量记录按钮点击的次数. 初始次数为 0, 每次点击计数加 1;
- 其次, 使用一个元胞数组记录每一次点击按钮时, 输入框中的指令内容 (均为字符串类型);
- 然后, 在每一次执行删除时, 点击次数减少 1, 即“回退”到之前的状态, 同时将元胞数组末尾的元素置空;
- 最后, 如果执行清空屏幕指令或清空结果指令, 将元胞数组全部置空, 将点击次数置 0; 如果执行计算指令, 将元胞数组全部置空后, 使其第一个元素为“ans”, 将点击次数置 1.

通过以空间换时间的策略, 每一次删除字符将会删除一串字符而非一个字符 (除了数字与一些符号外), 这样大大增加了删除的效率.

虽然这样做仍然无法解决实时更改表达式内容的问题, 但是用户操作的便捷性已经提升.

1.3 指令函数化

指令函数化的目的是为了统一的数据计算方法. MatLab 中的许多计算指令都被封装为函数的形式, 这种形式可以直接使用 `eval` 求解. 但是我们的计算器设计中可能出现一些其它类型的指令. 我们认为, 尽可能地将指令封装为函数的形式, 有利于计算器的功能设计与修改. 所以我们针对一些特殊的指令封装了几个函数 (`generate_1,2,3d_plot`, `generate_function_plot`, `integrate`, `differentiate`), 这将在具体的模块设计中得以呈现.

1.4 分辨率调整与 GUI 显示问题

在对计算器进行测试时, 我们发现在不同的电脑上, 显示的计算器界面非常不同, 有时会出现部分元素无法呈现的情况. 我们发现这个问题是由电脑的分辨率过低导致的: 在高分辨率的机器上设计出的计算器过大 (横纵坐标超出了范围), 在低分辨率的机器下就无法正常显示. 我们在低分辨率下对 GUI 进行设计, 再移植到高分辨率下, 就能解决这个问题. 目前, 这个计算器需要电脑分辨率大于 1024×768 才能够完全显示出来.

有时候在分辨率正常的情况下, GUI 的显示仍然会出现问题, 主要原因在 GUI 的主区域过于靠近 GUIDE 的边界. 通过调整 GUI 的主区域, 能够解决一部分问题, 但是对于不同的计算机仍然可能需要重新适配显示方案.

2 模块设计

在具体介绍每个模块之前, 我们先给出 GUI 的一个示意图:

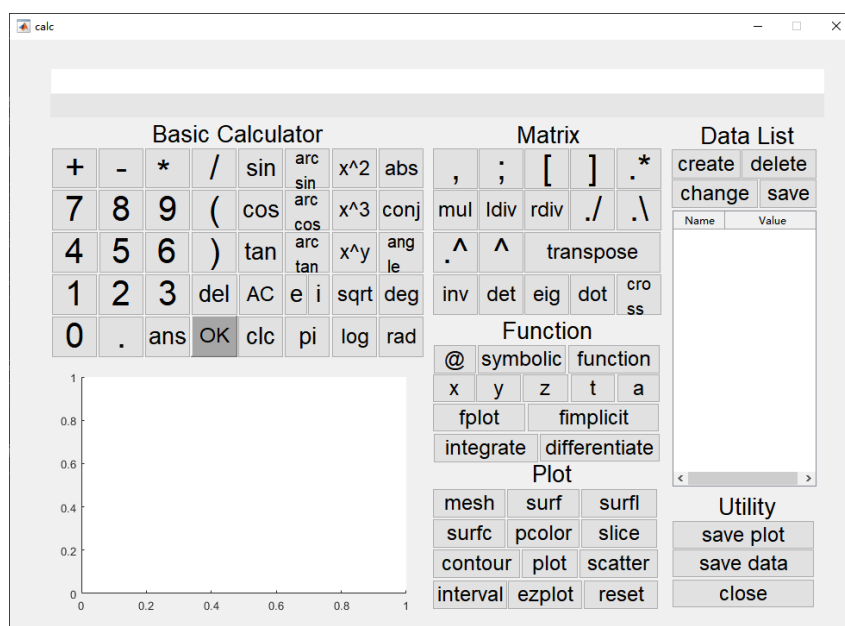


图 3 GUI 示意图

其中最上面的白色方框代表输入框, 输入框之下的灰色方框为输出框. Basic Calculator 为基本计算模块, Matrix 为矩阵模块, Function 为函数模块, Plot 为绘图模块, Data List 为数据存储模块, Utility 为实用功能 (其它) 模块.

2.1 基本计算模块

这一部分我们的目标是通过按钮输入运算式, 使运算式显示出来并得到运算结果. 本部分以字符串的形式显示运算符, 利用 `strcat` 函数, 将每一个输入的数字、字母或者运算符连接起来, 同时对输入字符的次数进行统计, 并通过 `set` 函数进行显示. 用户只需要保证输入的运算式能够被 MatLab 计算出来, 就能得到相应的结果, 否则会抛出一个异常, 并弹出错误对话框.

本部分通过输入得到的可以被 MatLab 识别的运算式, 点击 OK 得到运算结果, 并将其储存在 `ans` 变量之中, `ans` 可以不断被调用及更新.

这个部分包含的功能为: 数字 0 – 9, 符号 $+$, $-$, \times , $/$, $(,)$, 乘方符号, 虚数符号 i , 圆周率 π , 自然对数 e 的输入与显示, 基本的数学计算 (求三角函数, 反三角函数, 对数, 开根号, 取绝对值, 取幅角, 角度与弧度的转换).

在 MatLab 中自然对数 e 并不是一个变量, 但为了方便调用这个变量, 在我们的程序中将其看作一个常值. 所以在计算之前需要对其进行赋值: 我们将 e 记为全局变量:

```
handles.globalexp=exp(1);
```


然后在每次点击 OK 按钮时, 将 `handles.globalexp` 的值赋给变量 `e`:

```
e = handles.globalexp;
```

这种在计算之前给变量赋值的思路, 将会在之后的数据存储模块中得到充分运用.

此外, 我们还根据设计了清除屏幕的功能, 这一部分根据“计算器的记忆功能”一节思路, 设计了逐个删除, 清空屏幕与全部清空的功能, 分别用 `del`, `clc` 与 `AC` 键执行. `del` 键删除最后一次输入的字符串 (可能有多个字符); `clc` 键将字符串清空, 但不修改 `ans` 的结果, 即在下一次计算中, 上一次计算的结果仍然可以被调用; `AC` 键将输入框内所有字符清空, 并将 `ans` 归零, 并清空屏幕使得用户可以进行下一次计算.

2.2 矩阵模块

这一部分我们的目标是通过按钮输入矩阵运算式, 使运算式显示出来并得到运算结果, 数字的输入, 部分运算符 (如括号) 以及删除和清空等操作与基础运算共用同样的输入键.

本部分和基本运算模块一样, 以字符串的形式显示运算符, 利用 `strcat` 函数, 将每一个输入的数字, 字母或者运算符连接起来. 这个部分的功能为: 用 “,” 表示矩阵中行间元素的分隔, 用 “;” 表示矩阵中列间元素的分隔. 用 `mul` 表示矩阵乘法, 在输入框中显示为 “*”, 用 `ldiv` 表示矩阵左除, 即 $A/B = AB^{-1}$, 用 `rdiv` 表示矩阵右除, 即 $A \setminus B = A^{-1}B$. 用 “.*”, “./” 与 “.^” 分别表示矩阵相应位置相乘, 相除或求幂所得的新的矩阵, 如:

$$[a, b].*[c, d] = [a * c, b * d]$$

$$[a, b]./[c, d] = [a/c, b/d]$$

$$[a, b].^2 = [a^2, b^2]$$

用 `.\` 表示矩阵对应位置的反向相除, 如:

$$[a, b].\setminus[c, d] = [a \setminus c, b \setminus d]$$

针对矩阵的运算, `MatLab` 中有相关的函数, 我们使用 `dot` 函数用于计算矩阵的内积, `cross` 函数用于计算两个矩阵 (一般为三维矩阵) 的外积, 如果矩阵可逆, 使用 `inv` 函数用于计算矩阵的逆, `det` 函数用于计算方阵的行列式, `transpose` 函数用于计算矩阵的转置, `eig` 函数用于计算矩阵的特征值, 并将矩阵的所有特征值放在一起构成一个特征值矩阵.

由于基本计算模块与矩阵模块得到的数据的结果都是 “double” 类型, 而为了显示到输出框上的内容统一, 我们使用 `mat2str` 函数实现数据类型间的转换, 当数据只是一个数值时, 生成的输出结果没有 [,] 符号, 而数据是一个矩阵时, 生成的输出结果有 [,], 如果是多行矩阵, 也会使用 “;” 符号分隔每一行元素.

矩阵模块中的逗号(“,”)不仅仅在矩阵元素中起了作用, 它也同时在表达函数上发挥了重要作用, 以下我们将讨论函数模块的设计.

2.3 函数模块

这一部分我们将会创建函数句柄, 实现函数的符号定义法和匿名函数定义法的相互转换. 此外我们会分别对显函数和隐函数进行作图, 以及求解函数的定积分和(一阶和高阶)微分.

本部分将利用自行编写的函数将以上所要涉及的内容封装起来, 即只需要在输入栏里键入相应的函数自变量, 按下 OK 键后可以在输出栏或者图像绘制区域得到想要的结果. 我们限制了保存函数的数据类型, 如果需要保存函数, 其数据类型必须是“function_handle”类型而非“sym”类型.

2.3.1 函数定义的转换

MatLab 常见的函数表示方法有两种, 一是符号定义法, 二是匿名函数定义法. 符号定义法将非符号的函数表达式转换为符号对象, 并存储在符号变量中, 例如 $f(x) = x + x^2$ 的函数将被表示成 “ $x + x^2$ ” 的形式.

匿名函数法也就是创建函数句柄, Matlab 的 @ 符号是定义句柄的运算符, 匿名函数不以文件形式驻留在文件夹上; 它的生成方式最简捷, 可在指令窗或任何函数体内通过指令直接生成, 例如 $f(x) = x + x^2$ 的函数的匿名函数为 $@(x)x + x.^2$. 在各类数学问题中, 两种表示方法交替出现. 针对计算器中可能存储的各种函数, 我们需要对不同的表示方法进行快速转换, 从而实现高效性.

具体方法是要调用 Matlab 里已有的两个函数: sym(s) 和 matlabFunction. 前者是匿名函数定义法转换成符号定义法的函数, 后者是符号定义法转换成匿名函数定义法的函数.

2.3.2 函数图像绘制

对于显函数和隐函数, 我们分别使用 fplot, fimplicit 来绘制, fplot 可以同时使用符号定义的函数和匿名函数, 而 fimplicit 只能使用同时包含 x 和 y 的句柄匿名函数. 我们对与函数板块要求全部使用句柄函数, 那么正好可以将二者封装成方法不同的同一个函数. 我们定义这个函数为 generate_function_plot, 它需要针对显函数与隐函数分别处理, 这个函数的定义为:

```
function result = generate_function_plot(method, funct, start, stop)
```

其中 `method` 有两种取值, “normal” 对应显函数的绘制, “implicit” 对应隐函数的绘制, 而且这个取值将在点击对应按钮时体现出来. `funct` 为函数句柄. `start` 和 `stop` 为可选项, 默认值为:

```
start = -5; stop = 5;
```

我们通过 MatLab 中的 `nargin` 变量确定传入的参数个数, 确定采用默认值还是用户设定的参数.

具体来说, 在计算器我们用 `fplot` 和 `fimplicit` 两个按钮来调取这两个方法, 对应的在输入栏中通过点击按钮, 补充剩下的函数句柄和起点终点, 单击 OK 按钮得到图像.

此处需要注意的是, `fimplicit` 函数在 MatLab R2016b 后才被引入, 如果版本不正确, 只能使用绘图模块的 `ezplot` 绘制图形.

2.3.3 函数的积分和微分

Matlab 的函数句柄的积分用的是 `integral` 函数, 符号定义函数积分用的是 `int` 函数. 因为 `int` 函数能够支持定积分与不定积分两种功能, 而 `integral` 只能进行数值积分的计算, 本计算器我们使用函数句柄传入参数, 再将传入的函数句柄由 `sym` 函数转换为符号函数, 若不包含起点值和终点值, 则得到不定积分; 若包含起点值和终点值, 则得到定积分. 如果进行了不定积分运算, 则将符号变量转换回匿名函数变量, 如果进行了定积分计算, 则将计算所得的 `sym` 类型转换成 `double` 类型. 这个函数的定义为:

```
function result = integrate(funct, start, stop)
```

此处默认用户进行的是定积分, 所以仍然需要 `nargin` 变量来判断输入参数个数.

对应地, 我们也想得到一个函数的一阶或者是 n 阶导数, 或者是函数上某个点对应的导数值. Matlab 自带的 `diff` 函数可以求得一个函数的 n 阶导数, 配套的 `subs` 指令可以得出一个点对应的导数值. 设计函数时, 我们仍然将函数句柄作为传入变量, 并将函数定义为:

```
function result = differentiate(funct, dorder, point)
```

其中, `dorder` 表示对函数求导的阶数 (默认为一阶求导), `point` 为在某个点处求 (一阶或高阶) 导数. 我们默认用户所求的是一个函数的导数, 而不是某点处的导数值, 所以仍然需要 `nargin` 变量来判断输入参数个数. 如果用户所求的是一个函数的导数, 则将最终所得的符号函数转换为匿名函数形式; 如果用户所求是某点处的导数值, 则将最终所得的符号值转换为 “double” 类型.

2.4 绘图模块

绘图模块主要实现的功能为: 根据数据向量 x, y, z 与函数句柄 f , 画出各种类型的图像 (包括 `mesh`, `surf`, `surfl`, `surfc`, `pcolor`, `slice`, `contour`, `plot`, `scatter`), 除了 `plot` 和 `scatter` 能够直接通过使用 `eval` 函数求解表达式后绘图, 其余的图像绘制均需要使用数据网格化 (`meshgrid`) 的方法, 将数据点转换为网格点后, 通过函数句柄得出函数在网格点处的值, 再根据这些值绘制各种类型的函数图像. 而执行 `meshgrid` 函数需要以下指令:

```
[xx, yy] = meshgrid(x, y); % 2D  
[xx, yy, zz] = meshgrid(x, y, z); % 3D
```

因为新生成的变量可能各不一致, `eval` 函数不好处理新生成的变量, 后续使用上存在困难. 所以需要使用“指令函数化”的方法, 将各种指令封装为函数来处理.

2.4.1 对绘图函数的分类

既然需要将各种指令封装为函数来处理, 我们需要对所有的绘图函数加以分类, 找到它们的共同点后, 分别设计相应的函数加以处理. 我们将函数分为三大类:

- 一元函数绘制: 即绘制 $y = f(x)$ 的图像, 包括 `plot` 和 `scatter`;
- 二元函数绘制: 即绘制 $z = f(x, y)$ 的图像, 包括 `mesh`, `surf`, `surfl`, `surfc`, `pcolor` 和 `contour`;
- 三元函数绘制: 即绘制 $w = f(x, y, z)$ 在某个方向上的投影 (此时无法绘制图像), 包括 `slice`.

二元函数绘制又可以分为两小类:

- 绘制结果是立体的: 包括 `mesh`, `surf`, `surfl`, `surfc`;
- 绘制结果是平面的: 包括 `pcolor` 和 `contour`.

2.4.2 绘图函数的设计

通过以上的分类, 我们设计了三个函数分别处理上述的三个大分类:

```
generate_1dplot(method, x, y);  
generate_2dplot(method, x, y, f);  
generate_3dplot(method, x, y, z, f, xi, yi, zi);
```

而对每一个单独的绘图指令, 则使用参数 `method` 的值控制不同的绘图指令. 对于绘图区间, 我们使用整个区间内变量 x, y, z 的最大, 最小值作为绘图的区间. 针对二元函数绘制所分的两小类, 我们直接通过 `method` 的内容进行相应处理:

```

if (strcmp(method, 'pcolor') || strcmp(method, 'contour'))
    axis([min(x), max(x), min(y), max(y)]);
else
    zlabel('z-axis');
    axis([min(x), max(x), min(y), max(y), min(min(z)), max(max(z))]);
end

```

最后, 根据所设计的函数, 给出相应按钮的回调函数, 就能得到所需的绘图结果.

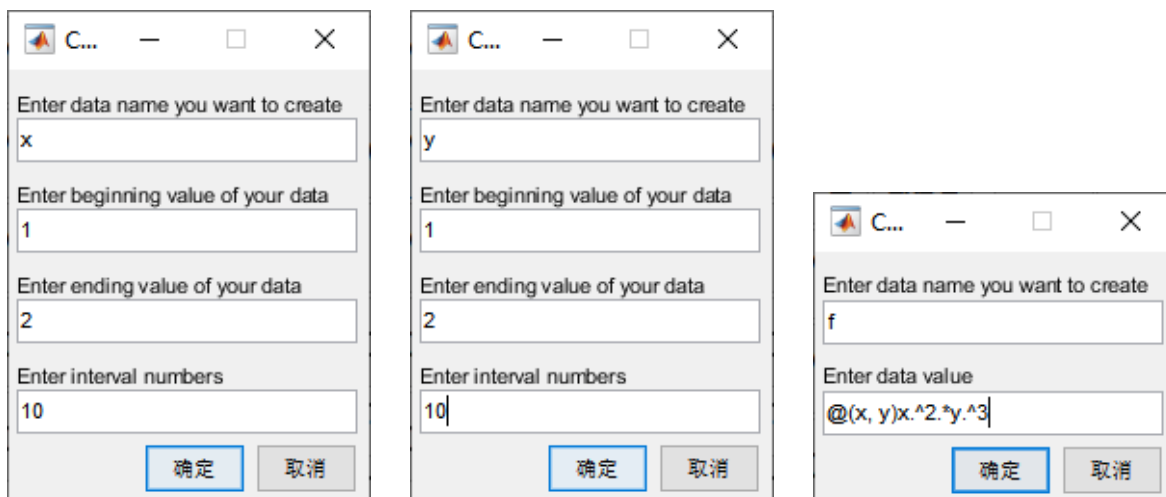
2.4.3 其它绘图功能

因为绘制图像的坐标向量一般是均匀的, 为了方便用户操作, 我们设计了一个类似于 MatLab 中 `linspace` 的功能, 通过一个输入框, 让用户创建一个 $[a, b]$ 中 n 等分的变量到数据存储区, 这样提高了变量创建的效率.

有时候用户只想绘制一个非常简单的图像, 或者用户不希望使用匿名函数的方式绘制图像, 为此我们设计了一个 `ezplot` 功能: 这个功能直接调用 MatLab 中的 `ezplot` 函数, 用户只需要在输入框中输入一个表达函数的字符串就可以绘制在 $[-2\pi, 2\pi]$ 的图像, 当然用户也可以自己制定图像的绘制区间. 绘制图像后, 用户可能需要清除所绘制的图像, 此时我们使用 `cla` 函数, 清除绘图区的图像.

一个绘制图像的例子:

以绘制 $f(x, y) = x^2y^3$ 在 $[1, 2] \times [1, 2]$ 上的表面图为例, 首先点击 `interval` 按钮, 分别以 $[1, 2]$ 区间十等分创建变量 x, y . 然后构造函数句柄: $f = @(x, y)x.^2.*y.^3$. 最后点击 `mesh` 按钮, 可以发现输入框处以及生成了一段初始代码 “`generate_2dplot('mesh',`”, 将这部分代码补全为 “`generate_2dplot('mesh',x,y,f)`”, 再单击 `OK` 按钮, 就能在绘图区中得到相应的结果. 以下是整个操作过程的示意图:



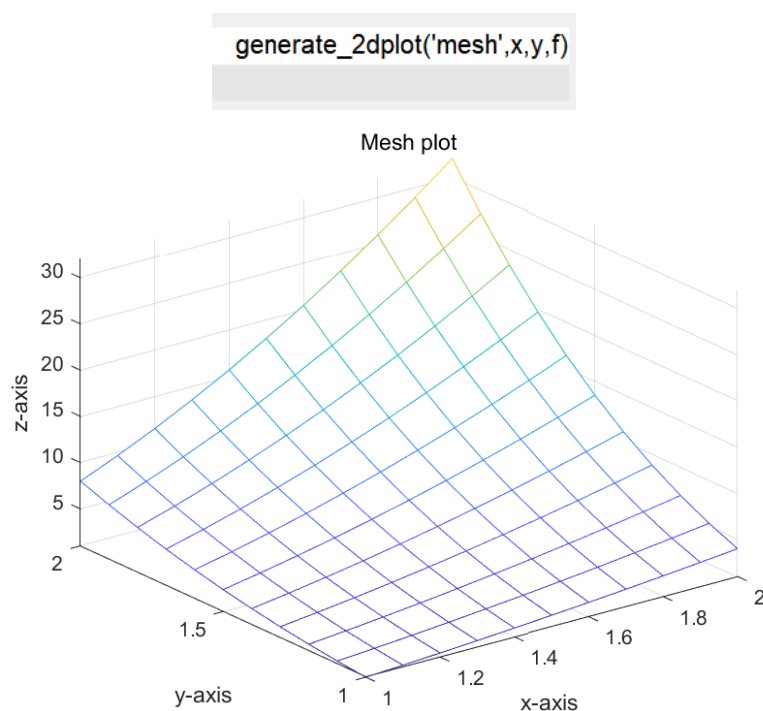


图 4 绘制图像的示意图

2.5 数据存储模块

因为有时用户可能希望在计算中使用一些自己定义的数据, 我们构造了数据存储模块. 这个模块主要实现存储计算数据, 创建变量, 修改变量, 删除变量. 在功能的实现上面, 我们主要采用了以下方法.

2.5.1 哈希表与元胞数组的双表查询与修改

使用哈希表主要是为了方便数据的查找, 修改与删除, 而使用元胞数组主要是为了方便数据作为一个表显示出来.

首先给出哈希表的定义:

定义 2.1 哈希表 (hash table) 是根据关键码值 (Key value) 而直接进行访问的数据结构. 也就是说, 它通过把关键码值映射到表中一个位置来访问记录, 以加快查找的速度. 这个映射函数叫做哈希函数, 存放记录的数组叫做哈希表.

简单地说, 就是构造一个映射函数, 实现两个集合之间元素的多对一关系. 这个函数的构造较为复杂, 但是 MatLab 中为我们创建了一个这样的容器, `containers.Map`, 这是一个高效的哈希表, 我们只需要指定键的数据类型 (字符串类型), 再指定值的数据类型 (任意类型) 就可以实现这个哈希表.

下表是 MatLab 中有关哈希表的函数:

函数名	函数实现的功能
<code>containers.Map()</code>	构造函数, 创建一个默认类型的哈希表
<code>containers.Map(key)</code>	根据 key 查询哈希表内元素值
<code>isKey()</code>	查询某个值是否为键
<code>remove()</code>	移除哈希表中的一个键值对
<code>keys()</code>	得到哈希表中键的元胞数组
<code>values()</code>	得到哈希表中值的元胞数组
<code>length()</code>	得到哈希表中的元素个数 (键的个数)

表 3 哈希表中的相关函数

通过哈希表中的相关函数可以方便地实现创建变量, 修改变量, 删除变量和存储计算数据的功能, 我们首先实现了一个数据输入框来得到用户输入的结果:

```
prompt = {'...', '...'};
title = '...';
answer = inputdlg(prompt, title);
```

结果中一般会有变量名称与变量表达式 (或变量的值), 通过访问上面得到的 `answer` 变量可以得出这些结果. 变量名称是 “string” 类型, 应作为哈希表中的键, 通过 `eval` 函数计算变量表达式, 所得到的数据类型是 “double”, “function_handle” 或 “sym” 类型, 应作为哈希表中的值.

然后实现相关功能:

- 创建变量: 首先检查待创建的变量名是否存在, 如果存在则抛出异常. 然后用 `eval` 函数计算表达式的值, 最后将变量名作为键, 计算结果作为值赋给哈希表.
- 修改变量: 首先检查待创建的变量名是否存在, 如果不存在则抛出异常. 然后用 `eval` 函数计算表达式的值, 最后将变量名作为键, 计算结果作为值修改哈希表中的相关元素.
- 删除变量: 首先检查待创建的变量名是否存在, 如果不存在则抛出异常. 然后用 `remove` 函数删除哈希表中的相应键值对.
- 存储计算数据: 首先检查待创建的变量名是否存在, 如果存在则抛出异常. 然后将变量名作为键, 计算结果 (即 `ans` 中的结果) 作为值赋给哈希表.

至此, 我们实现了数据的查找, 修改与删除, 下面将实现数据的可视化.

首先, 在 GUI 面板中创建一个表 (table), 将一个 0×2 的空元胞数组作为表中的值. 然后, 在每次数据进行修改时, 通过线性查找的方式对表内的元素进行相关的修改. 这个部分不能像哈希表一样具有 $O(1)$ 的较高效率, 但是由于数据的传输与数据的显示已经分离, 由于搜索造成的 $O(n)$ 的时间复杂度可以容忍.

最后, 为了提高交互性, 我们对 CellSelectionCallback 这个回调函数进行了修改, 使得当用户选中数据时, 输入框中显示这个数据代表的变量名. 由于 MatLab 允许多项选中, 而输入数据不可以多项输入, 我们在程序中限制了最大选择数量. 由于 MatLab 解除选择的按键为 Ctrl + 鼠标左键, 操作较为复杂, 我们在每次选择后, 自动刷新数据, 从而避免了解除选择数据的复杂问题.

2.5.2 用赋值方法将存储元素用于计算

上面构造的哈希表能够传输数据, 但是每次访问数据需要使用 handles.data(key) 的方式来访问, 这样不够简洁, 所以我们希望像 ans 变量一样, 输入一个用户定义的变量名, 就能够使用这个变量. 首先我们考虑了 eval 函数, eval 函数可以处理以下的计算式:

```
eval('var = value');
```

这样可以将 value 的值赋给 var, 且 var 会在当前函数区中作为变量出现. 但是 eval 函数无法正确处理以下的计算式:

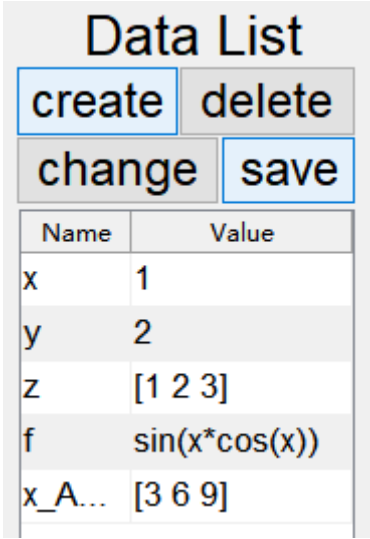
```
datakeys = handles.data.keys();
for i = 1: size(datakeys, 1)
    eval('datakeys(i) = value');
end
```

这是因为 datakeys 是一个已经存在的数组变量, MatLab 会将 value 赋给 datakeys 中的第 i 个元素, 但是我们实际想执行的代码是, 定义一个名为 datakeys 中第 i 个元素指向的字符串为变量名, value 作为值的变量. 这样就导致了计算错误. 我们采用了 MatLab 中的赋值方法来解决这个问题:

```
assignin(ws, var, val); % assigns the value val to the variable var
↳ in the workspace ws.
```

通过上面的赋值方法, 我们能够构造出一个名为 datakeys 中第 i 个元素指向的字符串为变量名, value 作为值的变量, 但是它所处的工作区是基础工作区, 不能被独立的函数工作区使用, 所以我们只能将 ws 的值赋为 “caller”, 写一个函数来调用这个赋值方法, 然后将参数回传给调用者, 这样才能够将所存储的元素用于计算.

以下是数据存储模块所得的效果图:



Data List	
create	delete
change	save
Name	Value
x	1
y	2
z	[1 2 3]
f	$\sin(x \cdot \cos(x))$
x_A...	[3 6 9]

图 5 数据存储模块效果图

2.6 其它模块

2.6.1 欢迎界面设计

欢迎界面设计的目的是对程序进行简介与版权声明, 对用户起到提示作用. 这个界面应该在主程序创建之前显示. 此处我们使用了一个模态的消息对话框 (msgbox) 实现欢迎界面, 使用 uiwait 函数监测用户行为, 需要用户做出确认操作后才能启动计算器.

2.6.2 保存界面设计

我们认为, 用户可能希望图像数据与所存储的数据能够脱离 MatLab 使用, 所以设置两个按钮, 分别将这两部分的数据保存下来很有必要. 首先, 我们使用了 MatLab 中的 uiputfile 函数获取保存文件的位置, 通过查找相关资料, uiputfile 函数能够通过元胞数组, 对保存的文件类型进行限制. 所以, 对于保存图片, 我们限制文件格式只能为 “*.jpg”, “.png”, “.eps” 与 “*.bmp” 四种类型之一; 对于保存数据, 文件格式只能为 “*.xls” 类型. 通过 uiputfile, 可以得到用户期望的文件保存路径. 得到路径后, 对于保存图片, 我们使用 saveas 函数, 通过用户选择的不同格式保存不同类型的图片; 对于保存数据, 我们使用 xlswrite 函数, 将数据项写入 excel 表格, 此时再利用异常处理机制, 捕获因为文件操作时产生的错误, 就能达到设计要求.

对于保存图片还有一个问题: 使用 saveas 函数会保存整个计算器的界面, 所以需要做一些调整. 新增一个空白且不可见的 Figure 句柄, 通过 copyobj 将图片绘制区域的图形句柄复制到空白的 Figure 句柄中, 再将复制后的 Figure 句柄的值传给 saveas 函数, 就能达到只保存绘图区域图像的目的.

附录 A 部分实验代码

由于整个 GUI 过长, 无法将所有代码放在这里, 所以我们选取了有代表性的代码进行展示.

A.1 表达式的计算与数据赋值

```
% --- Executes on button press in okbutton.
function okbutton_Callback(hObject, eventdata, handles)
% hObject    handle to okbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
e = handles.globalexp;
try % catching mistakes
    format;
    ans = handles.answer;
    % assigning data in the data list
    datakeys = keys(handles.data);
    for m = 1: size(datakeys, 2)
        assign_callback(char(datakeys{1, m}), handles.data(datakeys{1,
            ↪ m})));
    end
    if (isempty(handles.command))
        handles.answer = 0;
    else
        axes(handles.mainplot);
        handles.answer = eval(handles.command);
    end
    if isa(handles.answer, 'double')
        set(handles.answindow, 'string', mat2str(handles.answer));
    elseif isa(handles.answer, 'function_handle')
        set(handles.answindow, 'string', char(sym(handles.answer)));
    elseif isa(handles.answer, 'sym')
        set(handles.answindow, 'string', char(handles.answer));
    else
        throw(MException('calc:ValueError', 'Unsupported value.'));
    end
end
```

```

end
% set ans to command window
handles.commandcounts = 1;
handles.commandlist = {};
handles.commandlist{1} = 'ans';
set(handles.commandwindow, 'string', 'ans');
handles.command = 'ans';
catch ME
    errorDlg(ME.message, ME.identifier, 'modal');
end
guidata(hObject, handles);

% --- Especially for assigning values
function assign_callback(varname, value)
    assignin('caller', varname, value);

```

A.2 绘图模块的函数封装

```

% 1d-plot function
function result = generate_1dplot(method, x, y)
    if isa(y, 'function_handle')
        y = y(x);
    end
    if (strcmp(method, 'plot'))
        plot(x, y);
        title('Plot');
    elseif (strcmp(method, 'scatter'))
        scatter(x, y);
        title('Scatter plot');
    end
    xlabel('x-axis');
    ylabel('y-axis');
    axis([min(x), max(x), min(y), max(y)]);
    result = 0;

% 2d-plot function

```

```
function result = generate_2dplot(method, x, y, f)
    [xx, yy] = meshgrid(x, y); % create a meshgrid
    z = f(xx, yy);
    if (strcmp(method, 'mesh'))
        mesh(xx, yy, z);
        title('Mesh plot');
    elseif (strcmp(method, 'surf'))
        surf(xx, yy, z);
        title('Surf plot');
    elseif (strcmp(method, 'surfl'))
        surfl(xx, yy, z);
        title('Surfl plot');
    elseif (strcmp(method, 'surfc'))
        surfc(xx, yy, z);
        title('Surfc plot');
    elseif (strcmp(method, 'pcolor'))
        pcolor(xx, yy, z);
        title('Pcolor plot');
    elseif (strcmp(method, 'contour'))
        contour(xx, yy, z);
        title('Contour plot');
    end
    xlabel('x-axis');
    ylabel('y-axis');
    if (strcmp(method, 'pcolor') || strcmp(method, 'contour'))
        axis([min(x), max(x), min(y), max(y)]);
    else
        zlabel('z-axis');
        axis([min(x), max(x), min(y), max(y), min(min(z)), max(max(z))]);
    end
    result = 0;

% 3d-plot function
function result = generate_3dplot(method, x, y, z, f, xi, yi, zi)
    [xx, yy, zz] = meshgrid(x, y, z); % create a meshgrid
```

```
v = f(xx, yy, zz);
if (strcmp(method, 'slice'))
    slice(xx, yy, zz, v, xi, yi, zi);
    title('Slice plot');
end
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
axis([min(x), max(x), min(y), max(y), min(min(z)), max(max(z))]);
result = 0;
```

A.3 函数模块的函数封装

% generate function plot

```
function result = generate_function_plot(method, funct, start, stop)
    if (nargin == 2)
        start = -5;
        stop = 5;
    end
    if (strcmp(method, 'normal'))
        fplot(funct, [start, stop]);
        title('Function plot');
    elseif (strcmp(method, 'implicit'))
        fimplicit(funct, [start, stop]);
        title('Implicit function plot');
    end
    xlabel('x-axis');
    ylabel('y-axis');
    result = 0;
```

% integrate function

```
function result = integrate(funct, start, stop)
    sym_funct = sym(funct);
    if (nargin == 1)
        sym_int = int(sym_funct);
        result = matlabFunction(sym_int);
    end
```

```

else
    result = double(int(sym_funcnt, start, stop));
end

% differentiate function
function result = differentiate(funcnt, dorder, point)
    sym_funcnt = sym(funcnt);
    if (nargin == 1)
        sym_diff = diff(sym_funcnt);
        result = matlabFunction(sym_diff);
    elseif (nargin == 2)
        sym_diff = diff(sym_funcnt, dorder);
        result = matlabFunction(sym_diff);
    else
        sym_diff = diff(sym_funcnt, dorder);
        result = double(subs(sym_diff, point));
    end
end

```

A.4 数据存储模块的双表查询

```

function createdatabutton_Callback(hObject, eventdata, handles)
% hObject    handle to createdatabutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
prompt = {'Enter data name you want to create', 'Enter data value'};
title = 'Create Data';
answer = inputdlg(prompt, title);
if (~isempty(answer))
    try
        name = answer{1};
        % checking key existance
        if (isKey(handles.data, name))
            throw(MException('calc:KeyError', 'The key has been created
                ↳ before. '));
        end
        stringvalue = answer{2};
    end
end

```

```

value = eval(answer{2});
if isa(value, 'double')
    stringvalue = mat2str(value); % making matrix uniformly
elseif isa(value, 'function_handle')
    stringvalue = char(sym(value)); % making function uniformly
else
    throw(MException('calc:ValueError', 'Unsupported value.'));
end
handles.data(name) = value;
datalist = get(handles.datalist, 'Data'); % fetching data
newdata = {name, stringvalue}; % generating newdata
datalist = [datalist; newdata];
set(handles.datalist, 'Data', datalist); % setting data
catch ME
    errordlg(ME.message, ME.identifier, 'modal');
end
end
guidata(hObject, handles);

% --- Executes on button press in changedatabutton.
function changedatabutton_Callback(hObject, eventdata, handles)
% hObject    handle to changedatabutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
prompt = {'Enter data name you want to change', 'Enter new data'};
title = 'Change Data';
answer = inputdlg(prompt, title);
if (~isempty(answer))
    try
        name = answer{1};
        % checking key existance
        if (~isKey(handles.data, name))
            throw(MException('calc:KeyError', 'The key does not
                ↪ exist.'));
        end
    end
end

```

```

newstringvalue = answer{2};
newvalue = eval(answer{2});
if isa(newvalue, 'double')
    newstringvalue = mat2str(newvalue); % making matrix
    ↪ uniformly
elseif isa(newvalue, 'function_handle')
    newstringvalue = char(sym(newvalue)); % making function
    ↪ uniformly
else
    throw(MException('calc:ValueError', 'Unsupported value.'));
end
handles.data(name) = newvalue;
datalist = get(handles.datalist, 'Data'); % fetching data
datalist{char(datalist(:, 1)) == name, 2} = newstringvalue;
set(handles.datalist, 'Data', datalist); % setting data
catch ME
    errordlg(ME.message, ME.identifier, 'modal');
end
end
guidata(hObject, handles);

% --- Executes on button press in deletedatabutton.
function deletedatabutton_Callback(hObject, eventdata, handles)
% hObject    handle to deletedatabutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
prompt = 'Enter data name you want to delete';
title = 'Delete Data';
answer = inputdlg(prompt, title);
if (~isempty(answer))
    try
        name = answer{1};
        % checking key existance
        if (~isKey(handles.data, name))

```



```

        throw(MException('calc:KeyError', 'The key does not
        ↪ exist.'));
    end
    remove(handles.data, name); % remove data in mapping table
    datalist = get(handles.datalist, 'Data'); % fetching data
    datalist(cell2mat(datalist(:, 1)) == name, :) = [];
    set(handles.datalist, 'Data', datalist); % setting data
catch ME
    errordlg(ME.message, ME.identifier, 'modal');
end
end
guidata(hObject, handles);

% --- Executes on button press in savedatatoFilebutton.
function savedatabutton_Callback(hObject, eventdata, handles)
% hObject    handle to savedatatoFilebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
prompt = 'Enter data name you want to save data from answer zone';
title = 'Save Data';
answer = inputdlg(prompt, title);
if (~isempty(answer))
    try
        name = answer{1};
        value = handles.answer;
        if isa(value, 'double')
            stringValue = mat2str(value); % making matrix uniformly
        elseif isa(value, 'function_handle')
            stringValue = char(sym(value)); % making function uniformly
        elseif isa(value, 'sym')
            throw(MException('calc:ValueError', 'Symbolic value cannot
            ↪ be saved. Try converting it to a function_handle.'));
        else
            throw(MException('calc:ValueError', 'Unsupported value.'));
        end
    end
end

```

```

    % checking key existance
    if (isKey(handles.data, name))
        handles.data(name) = value;
        datalist = get(handles.datalist, 'Data'); % fetching data
        datalist{cell2mat(datalist(:, 1)) == name, 2} = stringvalue;
        set(handles.datalist, 'Data', datalist); % setting data
    else
        handles.data(name) = value;
        datalist = get(handles.datalist, 'Data'); % fetching data
        newdata = {name, stringvalue}; % generating newdata
        datalist = [datalist; newdata];
        set(handles.datalist, 'Data', datalist); % setting data
    end
catch ME
    errordlg(ME.message, ME.identifier, 'modal');
end
end
guidata(hObject, handles);

% --- Executes when selected cell(s) is changed in datalist.
function datalist_CellSelectionCallback(hObject, eventdata, handles)
% hObject    handle to datalist (see GCBO)
% eventdata  structure with the following fields (see
%   ↳ MATLAB.UI.CONTROL.TABLE)
% Indices: row and column indices of the cell(s) currently selecteds
% handles    structure with handles and user data (see GUIDATA)
rind = eventdata.Indices;
if (size(rind, 1) == 1 && size(rind, 2) == 2 && rind(1, 2) == 1) %
%   ↳ validating selection
    varname = handles.datalist.Data{rind(1), 1};
    handles.command = strcat(handles.command, varname);
    handles.commandcounts = handles.commandcounts + 1;
    handles.commandlist{handles.commandcounts} = handles.command;
    set(handles.commandwindow, 'string', handles.command);
end

```

```
% time consuming flushing data to implement auto-deselecting data.  
handles.datalist.Data = handles.datalist.Data;  
guidata(hObject, handles);
```