

MATLAB® Coder™

用户指南



MATLAB®

R2022b



## 如何联系 MathWorks



最新动态: [www.mathworks.com](http://www.mathworks.com)  
销售和服务: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
用户社区: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
技术支持: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



电话: 010-59827000



迈斯沃克软件 (北京) 有限公司  
北京市朝阳区望京东园四区 6 号楼  
北望金辉大厦 16 层 1604

MATLAB® Coder™ 用户指南

© COPYRIGHT 2011–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### 商标

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### 专利

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## 修订历史记录

2011 年 4 月	仅限在线版本	版本 2 (版本 2011a) 中的新增内容
2011 年 9 月	仅限在线版本	版本 2.1 (版本 2011b) 中的修订内容
2012 年 3 月	仅限在线版本	版本 2.2 (版本 2012a) 中的修订内容
2012 年 9 月	仅限在线版本	版本 2.3 (版本 2012b) 中的修订内容
2013 年 3 月	仅限在线版本	版本 2.4 (版本 2013a) 中的修订内容
2013 年 9 月	仅限在线版本	版本 2.5 (版本 2013b) 中的修订内容
2014 年 3 月	仅限在线版本	版本 2.6 (版本 2014a) 中的修订内容
2014 年 10 月	仅限在线版本	版本 2.7 (版本 2014b) 中的修订内容
2015 年 3 月	仅限在线版本	版本 2.8 (版本 2015a) 中的修订内容
2015 年 9 月	仅限在线版本	版本 3.0 (版本 2015b) 中的修订内容
2015 年 10 月	仅限在线版本	版本 2.8.1 (版本 2015aSP1) 中的再发布内容
2016 年 3 月	仅限在线版本	版本 3.1 (版本 2016a) 中的修订内容
2016 年 9 月	仅限在线版本	版本 3.2 (版本 2016b) 中的修订内容
2017 年 3 月	仅限在线版本	版本 3.3 (版本 2017a) 中的修订内容
2017 年 9 月	仅限在线版本	版本 3.4 (版本 2017b) 中的修订内容
2018 年 3 月	仅限在线版本	版本 4.0 (版本 2018a) 中的修订内容
2018 年 9 月	仅限在线版本	版本 4.1 (版本 2018b) 中的修订内容
2019 年 3 月	仅限在线版本	版本 4.2 (版本 2019a) 中的修订内容
2019 年 9 月	仅限在线版本	版本 4.3 (版本 2019b) 中的修订内容
2020 年 3 月	仅限在线版本	版本 5.0 (版本 2020a) 中的修订内容
2020 年 9 月	仅限在线版本	版本 5.1 (版本 2020b) 中的修订内容
2021 年 3 月	仅限在线版本	版本 5.2 (版本 2021a) 中的修订内容
2021 年 9 月	仅限在线版本	版本 5.3 (版本 2021b) 中的修订内容
2022 年 3 月	仅限在线版本	版本 5.4 (R2022a) 中的修订内容
2022 年 9 月	仅限在线版本	版本 5.5 (版本 2022b) 中的修订内容



## 查看 Bug 报告以确定并解决问题

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at [www.mathworks.com/support/bugreports/](http://www.mathworks.com/support/bugreports/). In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.



## 关于 MATLAB Coder

1

<b>MATLAB Coder 产品描述</b> .....	<b>1-2</b>
<b>产品概述</b> .....	<b>1-3</b>
何时使用 MATLAB Coder .....	1-3
嵌入式软件应用程序的代码生成 .....	1-3
定点算法的代码生成 .....	1-3

## 有关 C/C++ 代码生成的设计注意事项

2

<b>可从 MATLAB 算法生成代码的情况</b> .....	<b>2-2</b>
不从 MATLAB 算法生成代码的情况 .....	2-2
<b>使用合适的代码生成功能</b> .....	<b>2-3</b>
<b>从 MATLAB 生成 C/C++ 代码的前提条件</b> .....	<b>2-4</b>
<b>针对代码生成的 MATLAB 代码设计注意事项</b> .....	<b>2-5</b>
另请参阅 .....	2-5
<b>生成的代码和 MATLAB 代码之间的差异</b> .....	<b>2-6</b>
具有多种可能输出的函数 .....	2-7
写入 ans 变量 .....	2-7
逻辑短路 .....	2-7
循环索引溢出 .....	2-8
使用单精度操作数对 for 循环进行索引 .....	2-9
未执行的 for 循环的索引 .....	2-9
字符大小 .....	2-10
表达式中的计算顺序 .....	2-10
构造函数句柄时的名称解析 .....	2-11
终止行为 .....	2-13
可变大小 N 维数组的大小 .....	2-13
空数组的大小 .....	2-13
删除数组元素所生成的空数组的大小 .....	2-13
单精度和双精度操作数的二元按元素运算 .....	2-13
浮点数值结果 .....	2-14
NaN 和无穷 .....	2-14
负零 .....	2-15
代码生成目标 .....	2-15
MATLAB 类属性初始化 .....	2-15
具有 set 方法的嵌套属性赋值中的 MATLAB 类 .....	2-15

MATLAB 句柄类析构函数 .....	2-15
可变大小数据 .....	2-16
复数 .....	2-16
将使用连续一元运算符的字符串转换为 double .....	2-16
显示函数 .....	2-16
<b>潜在差异消息 .....</b>	<b>2-18</b>
自动维度不兼容 .....	2-18
mtimes 没有动态标量扩展 .....	2-18
矩阵-矩阵索引 .....	2-18
向量-向量索引 .....	2-19
循环索引溢出 .....	2-19
<b>支持 C/C++ 代码生成的 MATLAB 语言功能 .....</b>	<b>2-22</b>
代码生成支持的 MATLAB 功能 .....	2-22
代码生成不支持的 MATLAB 语言功能 .....	2-23

## 支持代码生成的函数、类和 System object

### 3

C/C++ 代码生成支持的函数和对象 .....	3-2
--------------------------	-----

## 为 C/C++ 代码生成定义 MATLAB 变量

### 4

用于代码生成的变量定义 .....	4-2
<b>为 C/C++ 代码生成定义变量的最佳做法 .....</b>	<b>4-3</b>
在使用变量之前通过赋值来定义变量 .....	4-3
对变量重新赋值时务必小心 .....	4-4
在变量定义中使用类型转换运算符 .....	4-5
在对索引变量赋值之前定义矩阵 .....	4-5
使用常量值向量对数组进行索引 .....	4-5
<b>变量属性的重新赋值 .....</b>	<b>4-7</b>
<b>通过不同属性重用同一变量 .....</b>	<b>4-8</b>
可通过不同属性重用同一变量的情形 .....	4-8
无法重用变量的情形 .....	4-8
变量重用的限制 .....	4-9
<b>支持的变量类型 .....</b>	<b>4-11</b>



## 5

代码生成的数据定义注意事项 .....	5-2
<b>复数数据的代码生成 .....</b>	<b>5-6</b>
定义复变量时的限制 .....	5-6
具有零值虚部的复数数据的代码生成 .....	5-6
含复数操作数的表达式的结果 .....	5-8
具有非有限值的复数乘法的结果 .....	5-9
<b>代码生成中的字符编码 .....</b>	<b>5-10</b>
<b>字符串的代码生成 .....</b>	<b>5-11</b>
限制 .....	5-11
生成的代码和 MATLAB 代码之间的差异 .....	5-11

## 可变量大小数据的代码生成

## 6

<b>可变量大小数组的代码生成 .....</b>	<b>6-2</b>
可变量大小数组的内存分配 .....	6-2
启用和禁用对可变量大小数组的支持 .....	6-3
代码生成报告中的可变量大小数组 .....	6-3
<b>为代码生成定义可变量大小数据 .....</b>	<b>6-4</b>
使用具有非常量维度的矩阵构造函数 .....	6-4
为同一个变量分配多个大小 .....	6-4
使用 coder.varsize 显式定义可变量大小数据 .....	6-5
<b>在代码生成的可变量大小支持方面与 MATLAB 的不兼容性 .....</b>	<b>6-8</b>
在标量扩展方面与 MATLAB 的不兼容性 .....	6-8
在确定可变量大小 N 维数组的大小方面与 MATLAB 的不兼容性 .....	6-9
在确定空数组的大小方面与 MATLAB 的不兼容性 .....	6-9
在确定空数组的类方面与 MATLAB 的不兼容性 .....	6-10
在矩阵-矩阵索引方面与 MATLAB 的不兼容性 .....	6-11
在建立向量-向量索引方面与 MATLAB 的不兼容性 .....	6-11
在代码生成的矩阵索引操作方面与 MATLAB 的不兼容性 .....	6-12
在串联可变量大小矩阵方面与 MATLAB 的不兼容性 .....	6-12
当串联内可变量大小元胞数组的花括号索引不返回任何元素时的差异 .....	6-12

## MATLAB 结构体的代码生成

## 7

用于代码生成的结构体定义 .....	7-2
代码生成允许的结构体操作 .....	7-3

<b>为代码生成定义标量结构体</b> .....	<b>7-4</b>
通过赋值定义标量结构体时的限制 .....	7-4
在每个控制流路径中以一致的顺序添加字段 .....	7-4
首次使用后添加新字段的限制 .....	7-4
<b>为代码生成定义结构体数组</b> .....	<b>7-6</b>
确保字段的一致性 .....	7-6
使用 repmat 定义具有一致字段属性的结构体数组 .....	7-6
使用 struct 定义结构体数组 .....	7-6
使用串联定义结构体数组 .....	7-7

## 8

### 分类数组的代码生成

## 9

### 元胞数组的代码生成

<b>代码生成的元胞数组限制</b> .....	<b>9-2</b>
元胞数组元素赋值 .....	9-2
可变大小元胞数组 .....	9-3
使用 cell 定义可变大小元胞数组 .....	9-3
元胞数组索引 .....	9-6
使用 {end + 1} 增大元胞数组 .....	9-6
元胞数组内容 .....	9-7
将元胞数组传递给外部 C/C++ 函数 .....	9-7

## 10

### 日期时间数组的代码生成

<b>代码生成的日期时间数组限制</b> .....	<b>10-2</b>
----------------------------	-------------

## 11

### 持续时间数组的代码生成

<b>持续时间数组的代码生成</b> .....	<b>11-2</b>
为代码生成定义持续时间数组 .....	11-2
允许对持续时间数组执行的操作 .....	11-2
支持持续时间数组的 MATLAB 工具箱函数 .....	11-3
<b>代码生成的持续时间数组限制</b> .....	<b>11-6</b>

## 12

<b>表的代码生成</b> .....	12-2
为代码生成定义表 .....	12-2
允许对表执行的操作 .....	12-2
支持表的 MATLAB 工具箱函数 .....	12-3

## 时间表的代码生成

## 13

## 枚举数据的代码生成

## 14

<b>在生成的代码中自定义枚举类型</b> .....	14-2
指定默认枚举值 .....	14-2
指定头文件 .....	14-3
在生成的枚举类型值名称中包含类名称前缀 .....	14-3
生成包含普通 C 枚举的 C++11 代码 .....	14-4

## MATLAB 类的代码生成

## 15

<b>用于代码生成的 MATLAB 类定义</b> .....	15-2
语言限制 .....	15-2
与类不兼容的代码生成功能 .....	15-2
为代码生成定义类属性 .....	15-3
不支持从内置 MATLAB 类继承 .....	15-5
<b>为 MATLAB 句柄类和 System object 生成代码</b> .....	15-6
<b>MATLAB 代码生成中的 System object</b> .....	15-9
代码生成中对 System object 的使用规则和限制 .....	15-9
通过 codegen 使用 System object .....	15-11
通过 MATLAB Function 模块使用 System object .....	15-11
通过 MATLAB System 模块使用 System object .....	15-11
System object 和 MATLAB Compiler 软件 .....	15-11

16

## 函数句柄的代码生成

17

代码生成的函数句柄限制 .....	17-2
-------------------	------

## 深度学习数组的代码生成

18

## 为代码生成定义函数

19

匿名函数的代码生成 .....	19-2
用于代码生成的匿名函数限制 .....	19-2

## 为代码生成调用函数

20

代码生成的函数调用解析 .....	20-2
函数调用解析的要点 .....	20-2
编译路径搜索顺序 .....	20-2
何时使用代码生成路径 .....	20-2
代码生成路径中文件类型的解析 .....	20-3
编译指令 %#codegen .....	20-4
使用 MATLAB 引擎在生成的代码中执行函数调用 .....	20-5
将函数声明为外部函数的情形 .....	20-5
使用 coder.extrinsic 构造 .....	20-6
使用 feval 调用 MATLAB 函数 .....	20-8
使用 mxArray .....	20-8
使用外部函数的限制 .....	20-9

21

22

23

24

<b>设置 MATLAB Coder 工程</b> .....	24-2
创建工程 .....	24-2
打开现有工程 .....	24-2
<b>使用 App 指定全局变量类型和初始值</b> .....	24-3
为什么要为全局变量指定类型定义? .....	24-3
指定全局变量类型 .....	24-3
通过示例定义全局变量 .....	24-3
定义或编辑全局变量类型 .....	24-4
定义全局变量初始值 .....	24-4
定义全局变量常量值 .....	24-5
删除全局变量 .....	24-5

25

<b>准备 MATLAB 代码以用于代码生成的工作流</b> .....	25-2
另请参阅 .....	25-2
<b>代码生成就绪工具</b> .....	25-3
“问题”选项卡 .....	25-3
“文件”选项卡 .....	25-4
<b>使用 MATLAB Coder App 生成 MEX 函数</b> .....	25-5
使用 MATLAB Coder App 生成 MEX 函数的工作流 .....	25-5
使用 MATLAB Coder App 生成 MEX 函数 .....	25-5
配置工程设置 .....	25-7
编译 MATLAB Coder 工程 .....	25-7
另请参阅 .....	25-7

修复在代码生成时检测到的错误 .....	25-9
另请参阅 .....	25-9

## 在 MATLAB 中测试 MEX 函数

### 26

为什么要在 MATLAB 中测试 MEX 函数? .....	26-2
--------------------------------	------

## 从 MATLAB 代码生成 C/C++ 代码

### 27

代码生成工作流 .....	27-2
另请参阅 .....	27-2
从 MATLAB 代码生成独立的 C/C++ 可执行文件 .....	27-3
使用 MATLAB Coder App 生成 C 可执行文件 .....	27-3
在命令行中生成 C 可执行文件 .....	27-9
指定 C/C++ 可执行文件的主函数 .....	27-10
指定主函数 .....	27-10
配置编译设置 .....	27-12
指定编译类型 .....	27-12
指定用于代码生成的语言 .....	27-14
指定输出文件名 .....	27-15
指定输出文件位置 .....	27-15
参数设定方法 .....	27-16
指定编译配置参数 .....	27-16
在 macOS 平台上安装 OpenMP 库 .....	27-20
生成代码以检测图像边缘 .....	27-21
为 MATLAB 卡尔曼滤波算法生成 C 代码 .....	27-27
使用 Black Litterman 方法生成用于优化投资组合的代码 .....	27-36

29

## MATLAB 代码的代码替换

30

## 自定义工具链注册

31

创建和编辑工具链定义文件 .....	31-2
使用 MSVC 工具链在 64 位 Windows® 平台上编译 32 位 DLL .....	31-3

## 部署生成的代码

32

在生成的函数接口中使用 C 数组 .....	32-2
生成的 C/C++ 代码中数组的实现 .....	32-2
emxArray 动态数据结构体定义 .....	32-3
与 emxArray 数据交互的工具函数 .....	32-4
示例 .....	32-5
使用示例主函数合并生成的代码 .....	32-13
使用示例主函数的工作流 .....	32-13
使用 MATLAB Coder App 控制示例主函数生成 .....	32-13
使用命令行界面控制示例主函数生成 .....	32-14
在应用程序中使用示例 C 主函数 .....	32-15
前提条件 .....	32-15
创建文件夹并复制相关文件 .....	32-15
对图像运行 Sobel 滤波器 .....	32-17
生成并测试 MEX 函数 .....	32-18
为 sobel.m 生成示例主函数 .....	32-18
复制示例主文件 .....	32-21
修改生成的示例主函数 .....	32-21
生成 Sobel 滤波器应用程序 .....	32-29
运行 Sobel 滤波器应用程序 .....	32-29
显示生成的图像 .....	32-29
生成的示例 C/C++ 主函数的结构 .....	32-31
文件 main.c 或 main.cpp 的内容 .....	32-31
文件 main.h 的内容 .....	32-33

## 33

<b>加速 MATLAB 算法的工作流</b> .....	33-2
另请参阅 .....	33-2
<b>使用 MATLAB 探查器探查 MEX 函数</b> .....	33-3
生成 MEX 探查文件 .....	33-3
示例 .....	33-3
折叠表达式对 MEX 代码覆盖率的影响 .....	33-5
<b>使用并行 for 循环 (parfor) 的算法加速</b> .....	33-7
生成的代码中的并行 for 循环 (parfor) .....	33-7
parfor 循环如何提高执行速度 .....	33-7
何时使用 parfor 循环 .....	33-8
何时不使用 parfor 循环 .....	33-8
parfor 循环语法 .....	33-8
parfor 限制 .....	33-8

## 外部代码集成

## 34

<b>从生成的代码中调用自定义 C/C++ 代码</b> .....	34-2
调用 C 代码 .....	34-2
从一个 C 函数返回多个值 .....	34-3
按引用传递数据 .....	34-4
集成使用自定义数据类型的外部代码 .....	34-5
集成使用指针、结构体和数组的外部代码 .....	34-6

## 生成高效且可重用的代码

## 35

<b>优化策略</b> .....	35-2
<b>内联代码</b> .....	35-4
<b>使用并行 for 循环 (parfor) 生成代码</b> .....	35-5
<b>MATLAB Coder 对生成代码进行优化</b> .....	35-6
常量折叠 .....	35-6
循环融合 .....	35-6
合并的连续矩阵运算 .....	35-7
排除不可达代码 .....	35-7
memcpy 调用 .....	35-7
memset 调用 .....	35-8



36

## 代码生成问题排查

37

无法确定元胞数组的每个元素都已赋值 .....	37-2
问题 .....	37-2
原因 .....	37-2
解决方法 .....	37-3

## 行优先数组布局

38

行优先和列优先数组布局 .....	38-2
计算机内存中的数组存储 .....	38-2
不同数组布局之间的转换 .....	38-2

## 使用 MATLAB Coder 进行深度学习

39

使用 MATLAB Coder 进行深度学习的前提条件 .....	39-2
MathWorks 产品 .....	39-2
第三方硬件和软件 .....	39-2
环境变量 .....	39-4
代码生成支持的网络和层 .....	39-7
支持的预训练网络 .....	39-7
支持的层 .....	39-8
支持的类 .....	39-16
int8 代码生成 .....	39-22
加载预训练网络以用于代码生成 .....	39-23
使用 coder.loadDeepLearningNetwork 加载网络 .....	39-23
为代码生成指定网络对象 .....	39-23
为代码生成指定 dlnetwork 对象 .....	39-24

## 生成 C++ 代码

40

C++ 代码生成 .....	40-2
生成 C++ 代码 .....	40-2

生成的代码中支持的 C++ 语言功能 .....	40-2
生成的 C 代码和 C++ 代码之间的其他区别 .....	40-2

## 仿真数据检查器

# 41

<b>在仿真数据检查器中查看数据 .....</b>	<b>41-2</b>
查看记录的数据 .....	41-2
从工作区或文件导入数据 .....	41-3
查看复数数据 .....	41-5
查看字符串数据 .....	41-5
查看基于帧的数据 .....	41-8
查看基于事件的数据 .....	41-8
<b>将 CSV 文件中的数据导入仿真数据检查器 .....</b>	<b>41-10</b>
基本文件格式 .....	41-10
多个时间向量 .....	41-10
信号元数据 .....	41-11
从 CSV 文件导入数据 .....	41-12
<b>仿真数据检查器如何比较数据 .....</b>	<b>41-15</b>
信号对齐 .....	41-15
同步 .....	41-16
插值 .....	41-17
容差设定 .....	41-17
限制 .....	41-18
<b>保存和共享仿真数据检查器数据和视图 .....</b>	<b>41-19</b>
保存和加载仿真数据检查器会话 .....	41-19
共享仿真数据检查器视图 .....	41-20
共享仿真数据检查器绘图 .....	41-20
创建仿真数据检查器报告 .....	41-21
将数据导出到工作区或文件 .....	41-22
将视频信号导出到 MP4 文件 .....	41-23
<b>以编程方式检查和比较数据 .....</b>	<b>41-25</b>
创建运行并查看数据 .....	41-25
比较同一运行中的两个信号 .....	41-26
使用全局容差比较各运行 .....	41-27
使用信号容差分析仿真数据 .....	41-28

# 关于 MATLAB Coder

---

- “MATLAB Coder 产品描述” (第 1-2 页)
- “产品概述” (第 1-3 页)

# MATLAB Coder 产品描述

## 从 MATLAB 代码生成 C 和 C++ 代码

MATLAB Coder 可从 MATLAB 代码生成适用于各种硬件平台（从桌面计算机系统到嵌入式硬件）的 C 和 C++ 代码。它支持大多数 MATLAB 语言和广泛的工具箱。您可以将生成的代码作为源代码、静态库或动态库集成到您的工程中。生成的代码是可读且可移植的。您可以将它与现有 C 和 C++ 代码及库的关键部分结合使用。您还可以将生成的代码打包为 MEX 函数以在 MATLAB 中使用。

与 Embedded Coder® 结合使用时，MATLAB Coder 可提供代码自定义、特定于目标的优化、代码可追溯性以及软件在环 (SIL) 和处理器在环 (PIL) 验证。

要将 MATLAB 程序部署为独立应用程序，请使用 MATLAB Compiler™。要生成与其他编程语言集成的软件组件，请使用 MATLAB Compiler SDK™。

## 产品概述

本节内容
“何时使用 MATLAB Coder” （第 1-3 页）
“嵌入式软件应用程序的代码生成” （第 1-3 页）
“定点算法的代码生成” （第 1-3 页）

### 何时使用 MATLAB Coder

MATLAB Coder 的用途：

- 从 MATLAB 代码生成可读、高效的独立 C/C++ 代码。
- 从 MATLAB 代码生成 MEX 函数以便：
  - 提高 MATLAB 算法的执行速度。
  - 验证在 MATLAB 内生成的 C 代码。
- 将自定义 C/C++ 代码集成到 MATLAB 中。

### 嵌入式软件应用程序的代码生成

Embedded Coder 产品是对 MATLAB Coder 产品的扩展，它提供了对嵌入式软件开发很重要的功能。使用 Embedded Coder 附加产品，您可以生成具有专业人工代码的清晰度和效率的代码。例如，您可以：

- 生成紧凑、快速的代码，这对实时仿真器、目标系统快速原型构建板、大规模生产中使用的微处理器以及嵌入式系统至关重要。
- 自定义生成的代码的外观。
- 针对特定的目标环境优化生成的代码。
- 启用追溯选项，帮助您验证生成的代码。
- 生成可重用的可重入代码。

### 定点算法的代码生成

使用 Fixed-Point Designer™ 产品，您可以生成：

- 能提高定点算法执行速度的 MEX 函数。
- 能按位匹配 MEX 函数结果的定点代码。



## 有关 C/C++ 代码生成的设计注意事项

---

- “可从 MATLAB 算法生成代码的情况”（第 2-2 页）
- “使用合适的代码生成功能”（第 2-3 页）
- “从 MATLAB 生成 C/C++ 代码的前提条件”（第 2-4 页）
- “针对代码生成的 MATLAB 代码设计注意事项”（第 2-5 页）
- “生成的代码和 MATLAB 代码之间的差异”（第 2-6 页）
- “潜在差异消息”（第 2-18 页）
- “支持 C/C++ 代码生成的 MATLAB 语言功能”（第 2-22 页）

## 可从 MATLAB 算法生成代码的情况

针对桌面和嵌入式系统基于 MATLAB 算法生成代码，您可以在 MATLAB 工作区内执行软件设计、实现和测试等所有工作。您可以：

- 验证您的算法是否适用于代码生成
- 自动生成高效、可读且紧凑的 C/C++ 代码，而无需手动转换您的 MATLAB 算法，并最大限度地减少在代码中引入错误的风险。
- 根据桌面和嵌入式应用程序的具体要求（如数据类型管理、内存使用量和速度），在 MATLAB 代码中修改您的设计。
- 测试生成的代码，轻松验证修改后的算法是否与原始 MATLAB 算法等效。
- 生成 MEX 函数以：
  - 在某些应用程序中提高 MATLAB 算法的执行速度。
  - 提高定点 MATLAB 代码的执行速度。
- 从 MATLAB 代码生成硬件描述语言 (HDL)。

## 不从 MATLAB 算法生成代码的情况

对于以下应用程序，不要从 MATLAB 算法生成代码。请改用推荐的 MathWorks® 产品。

要执行的操作	使用...
部署使用 Handle Graphics 的应用程序	MATLAB Compiler
使用 Java®	MATLAB Compiler SDK
使用不支持代码生成的工具箱函数	您为桌面和嵌入式应用程序重写的工具箱函数
在支持的 MATLAB 主机上部署基于 MATLAB 的 GUI 应用程序	MATLAB Compiler
部署基于 Web 的应用程序或 Windows® 应用程序	MATLAB Compiler SDK
将 C 代码与 MATLAB 对接	MATLAB mex 函数



## 使用合适的代码生成功能

如需执行以下操作...	使用...	需要的产品	如需了解详情...
生成 MEX 函数，用于验证生成的代码	<code>codegen</code> 函数	MATLAB Coder	请在“通过生成 MEX 函数加快 MATLAB 算法的执行速度”中尝试此项。
基于 MATLAB 算法生成可读、高效且紧凑的代码，以部署到台式机和嵌入式系统。	MATLAB Coder App	MATLAB Coder	请在“使用 MATLAB Coder App 生成 C 代码”中尝试此项。
	<code>codegen</code> 函数	MATLAB Coder	请在“通过命令行生成 C 代码”中尝试此项。
生成 MEX 函数以提高 MATLAB 算法的执行速度	MATLAB Coder App	MATLAB Coder	请参阅“Accelerate MATLAB Algorithms”。
	<code>codegen</code> 函数	MATLAB Coder	
将 MATLAB 代码集成到 Simulink® 中	MATLAB Function 模块	Simulink	请在“Call MATLAB Function Files in MATLAB Function Blocks” (Simulink) 中尝试此项。
加快定点 MATLAB 代码的执行速度	<code>fiaccel</code> 函数	Fixed-Point Designer	在“Code Acceleration and Code Generation from MATLAB” (Fixed-Point Designer) 中了解详细信息。
将自定义 C 代码集成到 MATLAB 中，并生成高效、可读的代码	<code>codegen</code> 函数	MATLAB Coder	在“从生成的代码中调用自定义 C/C++ 代码” (第 34-2 页) 中了解详细信息。
将自定义 C 代码集成到从 MATLAB 生成的代码中	<code>coder.ceval</code> 函数	MATLAB Coder	在 <code>coder.ceval</code> 中了解详细信息。
从 MATLAB 代码生成 HDL	MATLAB Function 模块	Simulink 和 HDL Coder™	在 <a href="http://www.mathworks.com/products/slhdlcoder">www.mathworks.com/products/slhdlcoder</a> 上了解详细信息。

### 从 MATLAB 生成 C/C++ 代码的前提条件

要从 MATLAB 算法生成 C/C++ 或 MEX 代码，您必须安装以下软件：

- MATLAB Coder 产品
- C/C++ 编译器

## 针对代码生成的 MATLAB 代码设计注意事项

在编写您要转换为高效的独立 C/C++ 代码的 MATLAB 代码时，必须考虑以下事项：

- 数据类型

C 和 C++ 使用静态类型。为了在使用之前确定您的变量类型，MATLAB Coder 需要每个变量的完整赋值。

- 数组大小

代码生成支持可变大小数组和矩阵。您可以在 MATLAB 函数中定义输入、输出和局部变量，以表示在运行时大小各不相同的数据。

- 内存

您可以选择生成的代码是使用静态还是动态内存分配。

使用动态内存分配，您可以减少内存使用量，但代价是需要花时间管理内存。使用静态内存分配，您可以获得更快的速度，但内存使用量较高。大多数 MATLAB 代码会利用 MATLAB 中的动态调整大小功能，因此，借助动态内存分配，您通常能够从现有 MATLAB 代码生成代码，而无需进行太多修改。动态内存分配甚至还允许某些程序在找不到上界时进行编译。

静态分配可减少所生成代码的内存使用量，因此适用于可用内存量有限的应用程序（如嵌入式应用程序）。

- 速度

由于嵌入式应用程序必须实时运行，因此代码必须足够快才能满足所需的时钟频率。

要提高生成的代码的执行速度：

- 选择合适的 C/C++ 编译器。不要使用 MathWorks 随 MATLAB 为 Windows 64 位平台提供的默认编译器。
- 考虑禁用运行时检查。

默认情况下，出于安全考虑，为您的 MATLAB 代码生成的代码包含内存完整性检查和响应性检查。通常情况下，这些检查会导致生成更多代码和减慢仿真速度。禁用运行时检查通常可以简化生成的代码并提高仿真速度。仅当您已确认数组边界和维度检查不必要时，才可以禁用这些检查。

### 另请参阅

- “数据定义”
- “可变大小数组的代码生成”（第 6-2 页）
- “Control Run-Time Checks”

# 生成的代码和 MATLAB 代码之间的差异

为了将 MATLAB 代码转换为高效的 C/C++ 代码，代码生成器引入了优化，有意使生成的代码与原始源代码具有不同的行为，并且有时产生不同的结果。

以下是一些差异：

- “具有多种可能输出的函数”（第 2-7 页）
- “写入 ans 变量”（第 2-7 页）
- “逻辑短路”（第 2-7 页）
- “循环索引溢出”（第 2-8 页）
- “使用单精度操作数对 for 循环进行索引”（第 2-9 页）
- “未执行的 for 循环的索引”（第 2-9 页）
- “字符大小”（第 2-10 页）
- “表达式中的计算顺序”（第 2-10 页）
- “构造函数句柄时的名称解析”（第 2-11 页）
- “终止行为”（第 2-13 页）
- “可变大小 N 维数组的大小”（第 2-13 页）
- “空数组的大小”（第 2-13 页）
- “删除数组元素所生成的空数组的大小”（第 2-13 页）
- “单精度和双精度操作数的二元按元素运算”（第 2-13 页）
- “浮点数值结果”（第 2-14 页）
- “NaN 和无穷”（第 2-14 页）
- “负零”（第 2-15 页）
- “代码生成目标”（第 2-15 页）
- “MATLAB 类属性初始化”（第 2-15 页）
- “具有 set 方法的嵌套属性赋值中的 MATLAB 类”（第 2-15 页）
- “MATLAB 句柄类析构函数”（第 2-15 页）
- “可变大小数据”（第 2-16 页）
- “复数”（第 2-16 页）
- “将使用连续一元运算符的字符串转换为 double”（第 2-16 页）
- “显示函数”（第 2-16 页）

在以下情形中会出现这些差异：

- 使用 `codegen` 命令或 MATLAB Coder 生成 MEX 和独立的 C/C++ 代码。
- 通过使用 `fiaccel` 命令生成 MEX 来实现定点代码加速。
- 使用 Simulink 进行 MATLAB Function 模块仿真。

当您运行生成的 `fiaccel` MEX、C/C++ MEX 或独立的 C/C++ 代码时，运行时错误检查可以检测到其中的一些差异。默认情况下，对 MEX 代码启用运行时错误检查，对独立 C/C++ 代码禁用运行时错误检查。为帮助您在部署代码之前识别和解决差异，代码生成器将差异的一部分报告为潜在差异。

## 具有多种可能输出的函数

某些数学运算，如矩阵的奇异值分解和特征值分解，可以有多个答案。实现此类运算的两种不同算法可能会为相同的输入值返回不同的输出。相同算法的两种不同实现也可能会出现这种行为。

对于此类数学运算，使用生成代码中的对应函数与使用 MATLAB 可能会为相同的输入值返回不同的输出。要查看某个函数是否有这种行为，请在对应的函数参考页中，参阅[扩展功能](#)下的 **C/C++ 代码生成** 部分。此类函数的示例包括 `svd` 和 `eig`。

## 写入 ans 变量

当您在未指定输出参数的情况下运行 MATLAB 代码并返回输出时，MATLAB 会将输出隐式写入 `ans` 变量中。如果工作区中已存在变量 `ans`，MATLAB 会将其值更新为返回的输出。

从此类 MATLAB 代码生成的代码不会将输出隐式写入 `ans` 变量。

例如，定义 MATLAB 函数 `foo`，它在第一行中显式创建 `ans` 变量。在第二行执行时，该函数会隐式更新 `ans` 的值。

```
function foo %#codegen
ans = 1;
2;
disp(ans);
end
```

在命令行中运行 `foo`。`ans` 的最终值，即 2，显示在命令行上。

```
foo
```

```
2
```

从 `foo` 生成一个 MEX 函数。

```
codegen foo
```

运行生成的 MEX 函数 `foo_mex`。此函数显式创建 `ans` 变量，并为其赋值 1。但是，`foo_mex` 并不将 `ans` 的值隐式更新为 2。

```
foo_mex
```

```
1
```

## 逻辑短路

假设您的 MATLAB 代码中逻辑运算符 `&` 和 `|` 放在方括号 (`[` 和 `]`) 内。对于这种代码模式，生成的代码不会对这些逻辑运算符采用短路行为，但某些 MATLAB 执行会采用短路行为。请参阅“[Tips](#)”和“[提示](#)”。

例如，定义 MATLAB 函数 `foo`，该函数在 `if...end` 模块的条件表达式中使用方括号内的 `&` 运算符。

```
function foo
if [returnsFalse() & hasSideEffects()]
end
end
```

```
function out = returnsFalse
out = false;
```

```
end

function out = hasSideEffects
out = true;
disp('This is my string');
end
```

& 运算符的第一个参数始终是 `false`，并确定条件表达式的值。因此，在 MATLAB 执行中，使用短路，并且不计算第二个参数。因此，`foo` 在执行期间不会调用 `hasSideEffects` 函数，也不会显示任何内容。

为 `foo` 生成一个 MEX 函数。调用生成的 MEX 函数 `foo_mex`。

```
foo_mex

This is my string
```

在生成的代码中，没有使用短路。因此，调用 `hasSideEffects` 函数，字符串显示在命令行中。

循环索引溢出

假设 `for` 循环结束值等于或接近循环索引数据类型的最大值或最小值。在生成的代码中，循环索引的最后一次递增或递减可能会导致索引变量溢出。索引溢出可能导致无限循环。

启用内存完整性检查时，如果代码生成器检测到循环索引可能溢出，它会报告错误。软件错误检查是保守型的。它可能会误报循环索引溢出。默认情况下，内存完整性检查对 MEX 代码启用，对独立 C/C++ 代码禁用。请参阅“为什么要在 MATLAB 中测试 MEX 函数？”（第 26-2 页）和“Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”。

为了避免循环索引溢出，请使用下表中的解决方法。

可能导致溢出的循环条件	解决方法
<ul style="list-style-type: none"><li>• 循环索引递增 1。</li><li>• 结束值等于整数类型的最大值。</li></ul>	如果循环不必覆盖整数类型的全部范围，请重写循环，使结束值不等于整数类型的最大值。例如，将：  N=intmax('int16') for k=N-10:N  替换为：  for k=1:10
<ul style="list-style-type: none"><li>• 循环索引递减 1。</li><li>• 结束值等于整数类型的最小值。</li></ul>	如果循环不必覆盖整数类型的全部范围，请重写循环，使结束值不等于整数类型的最小值。例如，将：  N=intmin('int32') for k=N+10:-1:N  替换为：  for k=10:-1:1

可能导致溢出的循环条件	解决方法
<ul style="list-style-type: none"><li>• 循环索引递增或递减 1。</li><li>• 起始值等于整数类型的最小值或最大值。</li><li>• 结束值等于整数类型的最大值或最小值。</li></ul>	<p>如果循环必须覆盖整数类型的全部范围，请将循环的开始值、步长值和结束值的类型转换为更大的整数或双精度值。例如，重写：</p> <pre>M= intmin('int16'); N= intmax('int16'); for k=M:N     % Loop body end</pre> <p>为：</p> <pre>M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N)     % Loop body end</pre>
<ul style="list-style-type: none"><li>• 循环索引按不等于 1 的值递增或递减。</li><li>• 在最后一次循环迭代中，循环索引不等于结束值。</li></ul>	<p>重写循环，使最后一次循环迭代中的循环索引等于结束值。</p>

使用单精度操作数对 for 循环进行索引

假设在您的 MATLAB 代码中，您正在对具有冒号运算符的 for 循环进行索引，其中至少一个冒号操作数是 single 类型的操作数，并且迭代次数大于 flintmax('single') = 16777216。当所有这些条件都满足时，代码生成可能会产生运行时或编译时错误，因为生成代码为循环索引变量计算的值与 MATLAB 计算的值不同。

以如下 MATLAB 代码为例：

```
function j = singlePIndex  
n = flintmax('single') + 2;  
j = single(0);  
for i = single(1):single(n)  
    j = i;  
end  
end
```

此代码段会在 MATLAB 中执行，但它会导致编译时或运行时错误，因为循环索引变量 i 的值在生成代码中的计算方式不同。代码生成器显示编译时或运行时错误，并停止代码生成或执行以防止出现这种差异。

为了避免这种差异，请用双精度类型或整数类型操作数替换单精度类型操作数。

有关运行时错误的详细信息，请参阅“Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”。

未执行的 for 循环的索引

在您的 MATLAB 代码和生成代码中，在一个 for 循环执行完成后，索引变量的值等于其在 for 循环的最后一次迭代中的值。

在 MATLAB 中，如果循环不执行，则索引变量的值存储为 []（空矩阵）。在生成代码中，如果循环不执行，则索引变量的值不同于 MATLAB 索引变量。

- 如果在运行时提供 for 循环的开始和结束变量，则索引变量的值等于范围的开始值。以如下 MATLAB 代码为例：

```
function out = indexTest(a,b)
for i = a:b
end
out = i;
end
```

假设 a 和 b 作为 1 和 -1 传递。则 for 循环不会执行。在 MATLAB 中，out 被赋值为 []。在生成代码中，out 被赋予 a 的值，即 1。

- 如果在编译时间之前提供 for 循环的开始和结束值，则在 MATLAB 和生成的代码中都会赋予索引变量值 []。以如下 MATLAB 代码为例：

```
function out = indexTest
for i = 1:-1
end
out = i;
end
```

在 MATLAB 和生成的代码中，out 都被赋值为 []。

## 字符大小

MATLAB 支持 16 位字符，但生成的代码以 8 位字符表示，这是适用于大多数嵌入式语言（如 C）的标准大小。请参阅“代码生成中的字符编码”（第 5-10 页）。

## 表达式中的计算顺序

生成的代码不强制指定表达式中的计算顺序。对于大多数表达式，计算的顺序并不重要。对于具有副作用的表达式，生成的代码可能以不同于原始 MATLAB 代码的顺序产生副作用。产生副作用的表达式包括：

- 修改持久性或全局变量的表达式
- 将数据显示到屏幕的表达式
- 将数据写入文件的表达式
- 修改句柄类对象属性的表达式

此外，生成的代码不强制非短路逻辑运算符的计算顺序。

为获得更可预测的结果，最好在编码时根据计算顺序将表达式拆分为多个语句。

- 重写

```
A = f1() + f2();
```

为

```
A = f1();
A = A + f2();
```



使生成的代码在调用 `f2` 之前调用 `f1`。

- 将多输出函数调用的输出赋给不相互依赖的变量。例如，重写

```
[y, y.f, y.g] = foo;
```

为

```
[y, a, b] = foo;  
y.f = a;  
y.g = b;
```

- 当您访问元胞数组的多个元胞的内容时，将结果赋给不相互依赖的变量。例如，重写

```
[y, y.f, y.g] = z{:};
```

为

```
[y, a, b] = z{:};  
y.f = a;  
y.g = b;
```

构造函数句柄时的名称解析

MATLAB 和代码生成遵循不同优先级规则来解析符号 `@` 后面的名称。这些规则不适用于匿名函数。下表总结了这些优先级规则。

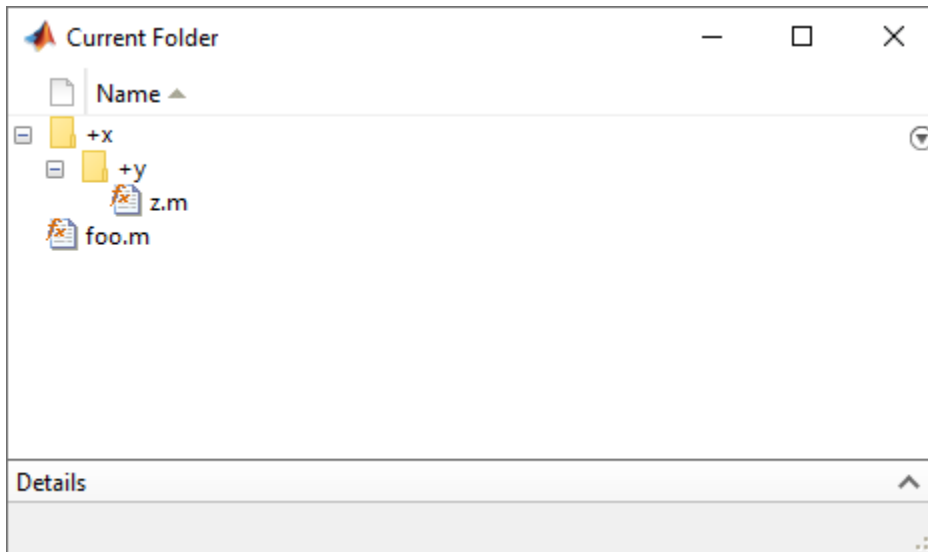
表达式	MATLAB 中的优先顺序	代码生成中的优先顺序
不包含句点的表达式，例如 <code>@x</code>	嵌套函数、局部函数、私有函数、路径函数	局部变量、嵌套函数、局部函数、私有函数、路径函数
仅包含一个句点的表达式，例如 <code>@x.y</code>	局部变量、路径函数	局部变量、路径函数（与 MATLAB 相同）
包含多个句点的表达式，例如 <code>@x.y.z</code>	路径函数	局部变量、路径函数

如果 `x` 是局部变量且本身就是函数句柄，则生成代码和 MATLAB 对表达式 `@x` 的解释将有所不同：

- MATLAB 生成错误。
- 生成代码将 `@x` 解释为 `x` 本身的函数句柄。

以下示例说明对于包含两个句点的表达式在行为上的差异。

假设您的当前工作文件夹包含一个包 `x`，该包又包含另一个包 `y`，其中再包含函数 `z`。当前工作文件夹还包含您要为其生成代码的入口函数 `foo`。



以下是文件 `foo` 的定义：

```
function out = foo
    x.y.z = @() 'x.y.z is an anonymous function';
    out = g(x);
end
```

```
function out = g(x)
    f = @x.y.z;
    out = f();
end
```

以下是函数 `z` 的定义：

```
function out = z
    out = 'x.y.z is a package function';
end
```

为 `foo` 生成一个 MEX 函数。分别调用生成的 MEX 函数 `foo_mex` 和 MATLAB 函数 `foo`。

```
codegen foo
foo_mex
foo

ans =

    'x.y.z is an anonymous function'

ans =

    'x.y.z is a package function'
```

生成代码产生第一个输出。MATLAB 产生第二个输出。代码生成将 `@x.y.z` 解析为在 `foo` 中定义的局部变量 `x`。MATLAB 将 `@x.y.z` 解析为在 `x.y` 包内的 `z`。

终止行为

生成的代码与 MATLAB 源代码的终止行为不一致。例如，如果无限循环不具有副作用，则优化会从生成的代码中删除它们。因此，即使对应的 MATLAB 代码不会终止，生成的代码也可能会终止。

可变大小 N 维数组的大小

对于可变大小 N 维数组，size 函数在生成的代码中返回的结果可能与在 MATLAB 源代码中不同。size 函数有时会在生成的代码中返回尾部维（单一维），但在 MATLAB 中始终会丢弃尾部维。例如，对于维度为 [4 2 1 1] 的 N 维数组 X，size(X) 可能会在生成的代码中返回 [4 2 1 1]，但在 MATLAB 中始终返回 [4 2]。请参阅“在确定可变大小 N 维数组的大小方面与 MATLAB 的不兼容性”（第 6-9 页）。

空数组的大小

空数组的大小在生成的代码中可能与 MATLAB 源代码中不同。请参阅“在确定空数组的大小方面与 MATLAB 的不兼容性”（第 6-9 页）。

删除数组元素所生成的空数组的大小

删除数组的所有元素将生成空数组。此空数组的大小在生成的代码中可能与在 MATLAB 源代码中不同。

情形	示例代码	MATLAB 中空数组的大小	生成的代码中空数组的大小
使用 colon 运算符 (:) 删除 m×n 数组的所有元素。	<code>coder.varsize('X',[4,4],[1,1]); X = zeros(2); X(:) = [];</code>	0-by-0	1-by-0
使用 colon 运算符 (:) 删除行向量的所有元素。	<code>coder.varsize('X',[1,4],[0,1]); X = zeros(1,4); X(:) = [];</code>	0-by-0	1-by-0
使用 colon 运算符 (:) 删除列向量的所有元素。	<code>coder.varsize('X',[4,1],[1,0]); X = zeros(4,1); X(:) = [];</code>	0-by-0	0-by-1
通过一次删除一个元素来删除列向量的所有元素。	<code>coder.varsize('X',[4,1],[1,0]); X = zeros(4,1); for i = 1:4     X(i) = []; end</code>	1-by-0	0-by-1

单精度和双精度操作数的二元按元素运算

如果您的 MATLAB 代码包含涉及单精度类型操作数和双精度类型操作数的二元按元素运算，则生成代码的运算结果与 MATLAB 的运算结果可能会稍有不同。

对于此类运算，MATLAB 会先将两项操作数都转换为双精度类型，并用双精度类型执行运算。然后，MATLAB 再将运算结果转换为单精度类型并返回结果。

而生成代码则是将双精度类型操作数转换为单精度类型。然后用两个单精度类型执行运算并返回结果。

例如，定义一个 MATLAB 函数 foo，用它调用二元按元素运算 plus。

```
function out = foo(a,b)
out = a + b;
end
```

定义单精度类型的变量 `s1` 和双精度类型的变量 `v1`。为 `foo` 生成一个 MEX 函数，该函数接受单精度类型输入和双精度类型输入。

```
s1 = single(1.4e32);  
d1 = -5.305e+32;  
codegen foo -args {s1, d1}
```

使用 `s1` 和 `d1` 作为输入调用 `foo` 和 `foo_mex`。比较两个结果。

```
ml = foo(s1,d1);  
mlc = foo_mex(s1,d1);  
ml == mlc
```

```
ans =
```

```
logical
```

```
0
```

比较的输出是逻辑值 `0`，这表明生成代码和 MATLAB 对这些输入分别产生了不同结果。

### 浮点数值结果

在下列情况下，生成的代码可能不会产生与 MATLAB 相同的浮点数值结果：

#### 计算机硬件使用扩展精度寄存器时

结果取决于 C/C++ 编译器如何分配扩展精度浮点寄存器。计算结果可能与 MATLAB 计算不一致，因为对浮点计算采用了不同的编译器优化设置，或浮点计算前后的代码不同。

#### 对于某些高级库函数

生成的代码可能使用不同算法来实现某些高级库函数，例如 `fft`、`svd`、`eig`、`mldivide` 和 `mrdivide`。

例如，生成的代码使用更简单的算法来实现 `svd` 以减小内存使用量。结果也可能根据矩阵属性而变化。例如，MATLAB 可能在运行时检测到对称矩阵或 Hermitian 矩阵，并切换到比生成的代码中的实现更快地执行计算的专用算法。

#### 对于 BLAS 库函数的实现

对于 BLAS 库函数的实现，生成的 C/C++ 代码使用 BLAS 函数的参考实现。这些参考实现可能会产生与 MATLAB 中特定于平台的 BLAS 实现不同的结果。

### NaN 和无穷

当 NaN 和 Inf 值在数学上没有意义时，它们在生成的代码中的模式可能不会与在 MATLAB 代码中完全相同。例如，如果 MATLAB 输出包含 NaN，生成的代码的输出也应包含 NaN，但不一定在相同的位置。

NaN 的位模式在 MATLAB 代码输出和生成的代码输出之间可能有所不同，这是因为用于生成代码的 C99 语言标准未对所有实现中的 NaN 指定唯一位模式。请避免比较不同实现中的位模式，例如，在 MATLAB 输出和 SIL 或 PIL 输出之间进行比较。

## 负零

在浮点类型中，值 0 具有正号或负号。在算术上，0 等于 -0，但某些运算对 0 输入符号敏感。示例包括 `rdivide`、`atan2`、`atan2d` 和 `angle`。除以 0 会生成 `Inf`，而除以 -0 会生成 `-Inf`。同样，`atan2d(0,-1)` 生成 180，而 `atan2d(-0,-1)` 生成 -180。

如果代码生成器检测到某浮点变量只接受适当范围的整数值，则代码生成器可以在生成的代码中对该变量使用整数类型。如果代码生成器对该变量使用整数类型，则该变量将 -0 存储为 +0，因为整数类型不存储值 0 的符号。如果生成的代码将该变量强制转换回浮点类型，则 0 的符号为正。除以 0 会生成 `Inf`，而不是 `-Inf`。同样，`atan2d(0,-1)` 生成 180，而不是 -180。

在其他情况下，生成的代码可能会以不同于 MATLAB 的方式处理 -0。例如，假设您的 MATLAB 代码使用 `z = min(x,y)` 计算两个双精度标量值 `x` 和 `y` 的最小值。生成的 C 代码中对应的行可能是 `z = fmin(x,y)`。函数 `fmin` 是在 C 编译器的运行时数学库中定义的。由于比较运算 `0.0 == -0.0` 在 C/C++ 中返回 `true`，因此编译器对 `fmin` 的实现可能会为 `fmin(0.0,-0.0)` 返回 0.0 或 -0.0。

## 代码生成目标

`coder.target` 函数在 MATLAB 中返回的值与在生成的代码中返回的值不同。这是为了帮助您确定您的函数是在 MATLAB 中执行，还是已针对仿真或代码生成目标进行编译。请参阅 `coder.target`。

## MATLAB 类属性初始化

在代码生成之前，MATLAB 在类加载时计算类默认值。代码生成器使用 MATLAB 计算的值。它不会重新计算默认值。如果属性定义使用函数调用来计算初始值，则代码生成器不会执行此函数。如果函数具有修改全局变量或持久变量等副作用，则生成的代码可能会产生与 MATLAB 不同的结果。有关详细信息，请参阅“为代码生成定义类属性”（第 15-3 页）。

## 具有 set 方法的嵌套属性赋值中的 MATLAB 类

当您为句柄对象属性赋值时，如果该属性本身就是另一个对象的属性（依此类推），则生成的代码可以调用 MATLAB 不调用的句柄类 `set` 方法。

例如，假设您定义一组变量，使得 `x` 是句柄对象，`pa` 是对象，`pb` 是句柄对象，`pc` 是 `pb` 的属性。然后进行嵌套属性赋值，例如：

```
x.pa.pb.pc = 0;
```

在本例中，生成的代码调用对象 `pb` 的 `set` 方法和 `x` 的 `set` 方法。MATLAB 仅调用 `pb` 的 `set` 方法。

## MATLAB 句柄类析构函数

在下列情况下，生成的代码中句柄类析构函数的行为可能与 MATLAB 中的行为不同：

- 有些独立对象在 MATLAB 中的销毁顺序可能与生成的代码中的顺序不同。
- 生成的代码中的对象的生命周期可能与它们在 MATLAB 中的生命周期不同。
- 生成的代码不销毁部分构造的对象。如果句柄对象在运行时未完全构造，则生成的代码将生成错误消息，但不为该对象调用 `delete` 方法。对于 System object™，如果在 `setupImpl` 中存在运行时错误，则生成的代码不为该对象调用 `releaseImpl`。

MATLAB 确实会调用 `delete` 方法来销毁部分构造的对象。

有关详细信息，请参阅“Code Generation for Handle Class Destructors”。

### 可变大小数据

请参阅“在代码生成的可变大小支持方面与 MATLAB 的不兼容性”（第 6-8 页）。

### 复数

请参阅“复数数据的代码生成”（第 5-6 页）。

### 将使用连续一元运算符的字符串转换为 double

如果将包含多个连续一元运算符的字符串转换为 `double`，则在 MATLAB 和生成的代码之间可能产生不同结果。以如下函数为例：

```
function out = foo(op)
out = double(op + 1);
end
```

对于输入值“-”，该函数将字符串“-1”转换为 `double`。在 MATLAB 中，结果为 NaN。在生成的代码中，结果为 1。

### 显示函数

MATLAB 代码中省略分号的语句和表达式隐式调用 `display` 函数。您也可以显式调用 `display`，如下所示：

```
display(2+3);
```

5

为调用 `display` 函数的 MATLAB 代码生成的 MEX 代码将保留对此函数的调用并显示输出。在为无权访问 MATLAB Runtime 的目标生成的独立代码中，删除了对 `display` 的隐式和显式调用。这包括对 `display` 的覆盖类方法的调用。

要在为其他目标生成的代码中显示文本，请在 MATLAB 类中覆盖 `disp` 函数。例如：

**%MATLAB Class**

```
classdef foo
    methods
        function obj = foo
            end
        function disp(self)
            disp("Overridden disp");
        end
    end
end
```

**%Entry-point Function**

```
function callDisp
a = foo;
disp(a);
end
```

为入口函数生成的代码如下所示：

```
/* Include Files */
#include "callDisp.h"
#include <stdio.h>

/* Function Definitions */
/*
 * Arguments   : void
 * Return Type : void
 */
void callDisp(void)
{
    printf("%s\n", "Overridden disp");
    fflush(stdout);
}
```

## 函数句柄差异

通过 MATLAB 中的函数句柄调用 `display` 也会输出变量的名称。例如，在 MATLAB 中运行此函数会产生以下输出：

```
function displayDiff
z = 10;
f = @display;
f(z)
end
```

```
z =
```

```
    10
```

但是，为此代码段生成的代码只输出值 10。

## 另请参阅

### 详细信息

- “Potential Differences Reporting”
- “潜在差异消息”（第 2-18 页）
- “为什么要在 MATLAB 中测试 MEX 函数？”（第 26-2 页）
- “Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”

# 潜在差异消息

当您启用潜在差异报告时，代码生成器会报告生成的代码的行为和 MATLAB 代码的行为之间的潜在差异。在生成独立代码之前检查并解决潜在差异有助于避免生成的代码中出现错误和不正确的答案。

以下是一些潜在差异消息：

- “自动维度不兼容”（第 2-18 页）
- “mtimes 没有动态标量扩展”（第 2-18 页）
- “矩阵-矩阵索引”（第 2-18 页）
- “向量-向量索引”（第 2-19 页）
- “循环索引溢出”（第 2-19 页）

## 自动维度不兼容

In the generated code, the dimension to operate along is selected automatically, and might be different from MATLAB. Consider specifying the working dimension explicitly as a constant value.

此限制适用于接受工作维度（运算所基于的维度）作为输入的函数。在 MATLAB 和代码生成中，如果您不提供工作维度，函数会选择工作维度。在 MATLAB 中，函数选择其大小不等于 1 的第一个维度。对于代码生成，函数选择具有可变大小或不等于 1 的固定大小的第一个维度。如果该工作维度具有可变大小并且在运行时大小变为 1，则该工作维度不同于 MATLAB 中的工作维度。因此，如果启用了运行时错误检查，可能会发生错误。

例如，假设 X 是维度为 1x:3x:5 的可变大小矩阵。在生成的代码中，`sum(X)` 的行为类似于 `sum(X,2)`。在 MATLAB 中，`sum(X)` 的行为类似于 `sum(X,2)`，除非 `size(X,2)` 为 1。在 MATLAB 中，当 `size(X,2)` 为 1 时，`sum(X)` 的行为类似于 `sum(X,3)`。

要避免此问题，请将需要的工作维度显式指定为一个常量值。例如，`sum(X,2)`。

## mtimes 没有动态标量扩展

The generated code performs a general matrix multiplication. If a variable-size matrix operand becomes a scalar at run time, dimensions must still agree. There will not be an automatic switch to scalar multiplication.

以 A\*B 相乘为例。如果代码生成器发现 A 是标量，B 是矩阵，则代码生成器会生成用于标量矩阵乘法的代码。但是，如果代码生成器发现 A 和 B 是可变大小矩阵，它会生成用于一般矩阵乘法的代码。在运行时，如果发现 A 是标量，则生成的代码不会更改其行为。因此，当启用运行时错误检查时，可能会出现大小不匹配错误。

## 矩阵-矩阵索引

For indexing a matrix with a matrix, `matrix1(matrix2)`, the code generator assumed that the result would have the same



size as matrix2.If matrix1 and matrix2 are vectors at run time, their orientations must match.

在矩阵-矩阵索引中，您可以使用一个矩阵来索引另一个矩阵。在 MATLAB 中，矩阵-矩阵索引的一般规则是，结果的大小和方向与索引矩阵的大小和方向保持一致。例如，如果 **A** 和 **B** 是矩阵，则 **size(A(B))** 等于 **size(B)**。当 **A** 和 **B** 是向量时，MATLAB 将应用特殊规则。特殊的向量-向量索引规则是，结果的方向等于数据矩阵的方向。例如，如果 **A** 是  $1 \times 5$ ，**B** 是  $3 \times 1$ ，则 **A(B)** 是  $1 \times 3$ 。

代码生成器应用与 MATLAB 相同的矩阵-矩阵索引规则。如果 **A** 和 **B** 是可变大小矩阵，为了应用矩阵-矩阵索引规则，代码生成器会假设 **size(A(B))** 等于 **size(B)**。如果 **A** 和 **B** 在运行时变成向量且方向不同，则该假设不正确。因此，如果启用了运行时错误检查，可能会发生错误。

为了避免此问题，请通过使用冒号操作符进行索引，强制您的数据为向量。例如，假设您的代码在运行时有意要在向量和常规矩阵之间进行切换。您可以对向量-向量索引执行显式检查：

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
    D = A(B);
end
...
```

第一个分支中的索引指定 **C** 和 **B(:)** 是编译时向量。因此，代码生成器将应用使用一个向量索引另一个向量的索引规则。结果的方向等于数据向量 **C** 的方向。

## 向量-向量索引

For indexing a vector with a vector, vector1(vector2), the code generator assumed that the result would have the same orientation as vector1.If vector1 is a scalar at run time, the orientation of vector2 must match vector1.

在 MATLAB 中，特殊的向量-向量索引规则是，结果的方向等于数据向量的方向。例如，如果 **A** 是  $1 \times 5$ ，**B** 是  $3 \times 1$ ，则 **A(B)** 是  $1 \times 3$ 。但是，如果数据向量 **A** 是标量，则 **A(B)** 的方向等于索引向量 **B** 的方向。

代码生成器应用与 MATLAB 相同的向量-向量索引规则。如果 **A** 和 **B** 是可变大小向量，为了应用索引规则，代码生成器会假设 **B** 的方向与 **A** 的方向一致。如果 **A** 在运行时变成标量，并且 **A** 和 **B** 的方向不一致，则该假设不正确。因此，如果启用了运行时错误检查，可能会发生运行时错误。

为了避免出现此问题，应使向量的方向保持一致。或者通过指定行和列来索引单个元素。例如，**A(row, column)**。

## 循环索引溢出

The generated code assumes the loop index does not overflow on the last iteration of the loop.If the loop index overflows, an infinite loop can occur.

假设 **for** 循环结束值等于或接近循环索引数据类型的最大值或最小值。在生成的代码中，循环索引的最后一次递增或递减可能会导致索引变量溢出。索引溢出可能导致无限循环。

启用内存完整性检查时，如果代码生成器检测到循环索引可能溢出，它会报告错误。软件错误检查是保守型的。它可能会误报循环索引溢出。默认情况下，内存完整性检查对 MEX 代码启用，对独立 C/C++ 代码禁用。请参阅“为什么要在 MATLAB 中测试 MEX 函数？”（第 26-2 页）和“Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”。

为了避免循环索引溢出，请使用下表中的解决方法。

可能导致溢出的循环条件	解决方法
<ul style="list-style-type: none"><li>循环索引递增 1。</li><li>结束值等于整数类型的最大值。</li></ul>	<p>如果循环不必覆盖整数类型的全部范围，请重写循环，使结束值不等于整数类型的最大值。例如，将：</p> <pre>N=intmax('int16') for k=N-10:N</pre> <p>替换为：</p> <pre>for k=1:10</pre>
<ul style="list-style-type: none"><li>循环索引递减 1。</li><li>结束值等于整数类型的最小值。</li></ul>	<p>如果循环不必覆盖整数类型的全部范围，请重写循环，使结束值不等于整数类型的最小值。例如，将：</p> <pre>N=intmin('int32') for k=N+10:-1:N</pre> <p>替换为：</p> <pre>for k=10:-1:1</pre>
<ul style="list-style-type: none"><li>循环索引递增或递减 1。</li><li>起始值等于整数类型的最小值或最大值。</li><li>结束值等于整数类型的最大值或最小值。</li></ul>	<p>如果循环必须覆盖整数类型的全部范围，请将循环的开始值、步长值和结束值的类型转换为更大的整数或双精度值。例如，重写：</p> <pre>M= intmin('int16'); N= intmax('int16'); for k=M:N     % Loop body end</pre> <p>为：</p> <pre>M= intmin('int16'); N= intmax('int16'); for k=int32(M):int32(N)     % Loop body end</pre>
<ul style="list-style-type: none"><li>循环索引按不等于 1 的值递增或递减。</li><li>在最后一次循环迭代中，循环索引不等于结束值。</li></ul>	<p>重写循环，使最后一次循环迭代中的循环索引等于结束值。</p>

## 另请参阅

### 详细信息

- “Potential Differences Reporting”
- “生成的代码和 MATLAB 代码之间的差异”（第 2-6 页）
- “在代码生成的可变大小支持方面与 MATLAB 的不兼容性”（第 6-8 页）
- “为什么要在 MATLAB 中测试 MEX 函数？”（第 26-2 页）
- “Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”

# 支持 C/C++ 代码生成的 MATLAB 语言功能

## 代码生成支持的 MATLAB 功能

从 MATLAB 代码生成代码支持许多主要的语言功能，包括：

- n 维数组（请参阅“Array Size Restrictions for Code Generation”）
- 矩阵运算，包括删除行和列
- 可变大小数据（请参阅“可变大小数组的代码生成”（第 6-2 页））
- 下标（请参阅“在代码生成的矩阵索引操作方面与 MATLAB 的不兼容性”（第 6-12 页））
- 复数（请参阅“复数数据的代码生成”（第 5-6 页））
- 数值类（请参阅“支持的变量类型”（第 4-11 页））
- 双精度、单精度和整数数学
- 枚举（请参阅“Code Generation for Enumerations”）
- 定点算术
- 程序控制语句 `if`、`switch`、`for`、`while` 和 `break`
- 算术、关系和逻辑运算符
- 局部函数
- 持久变量
- 全局变量（请参阅“使用 App 指定全局变量类型和初始值”（第 24-3 页））
- 结构体（请参阅“用于代码生成的结构体定义”（第 7-2 页））
- 元胞数组（请参阅“元胞数组”）
- 表（请参阅“表的代码生成”（第 12-2 页））
- 时间表（请参阅“Code Generation for Timetables”）
- 字符（请参阅“代码生成中的字符编码”（第 5-10 页））
- 字符串标量（请参阅“字符串的代码生成”（第 5-11 页））
- `categorical` 数组（请参阅“Code Generation for Categorical Arrays”）
- `datetime` 数组（请参阅“Code Generation for Datetime Arrays”）
- `duration` 数组（请参阅“持续时间数组的代码生成”（第 11-2 页））
- 稀疏矩阵（请参阅“Code Generation for Sparse Matrices”）
- 函数句柄（请参阅“代码生成的函数句柄限制”（第 17-2 页））
- 匿名函数（请参阅“匿名函数的代码生成”（第 19-2 页））
- 递归函数（请参阅“Code Generation for Recursive Functions”）
- 嵌套函数（请参阅“Code Generation for Nested Functions”）
- 可变长度输入和输出参数列表（请参阅“Code Generation for Variable Length Argument Lists”）
- 函数参数验证（请参阅“Generate Code for arguments Block That Validates Input and Output Arguments”）
- MATLAB 工具箱函数的子集（请参阅“C/C++ 代码生成支持的函数和对象”（第 3-2 页））
- 多个工具箱中的函数和 System object 的子集（请参阅“C/C++ 代码生成支持的函数和对象”（第 3-2 页））

- 函数调用（请参阅“代码生成的函数调用解析”（第 20-2 页））
- 类别名
- MATLAB 类（请参阅“用于代码生成的 MATLAB 类定义”（第 15-2 页））

## 代码生成不支持的 MATLAB 语言功能

基于 MATLAB 的代码生成不支持以下常用 MATLAB 功能（以上所列并非全部）：

- 脚本
- GPU 数组

MATLAB Coder 不支持 GPU 数组。但是，如果您有 GPU Coder™，则可以生成接受 GPU 数组输入的 CUDA® MEX 代码。


- **calendarDuration** 数组
- Java
- 映射容器
- 时间序列对象
- tall 数组
- **try/catch** 语句
- **import** 语句
- **pattern** 数组



## 支持代码生成的函数、类和 System object

# C/C++ 代码生成支持的函数和对象

您可以为从 MATLAB 代码调用的 MATLAB 内置函数和工具箱函数及 System object 的子集生成高效的 C/C++ 代码。

下表列出了这些函数和 System object。在这些表中，函数或 System object 的名称前的  图标表示该函数或 System object 有特定的与 C/C++ 代码生成相关的用法说明和限制。要查看这些用法说明和限制，请在对应的参考页中，向下滚动到底部的**扩展功能**部分，并展开 **C/C++ 代码生成**部分。

- C/C++ 代码生成支持的函数和对象（按类别排列）
- C/C++ 代码生成支持的函数和对象（按字母顺序排列）

## 另请参阅

## 相关示例

- “支持 C/C++ 代码生成的 MATLAB 语言功能”（第 2-22 页）



## 为 C/C++ 代码生成定义 MATLAB 变量

---

- “用于代码生成的变量定义” (第 4-2 页)
- “为 C/C++ 代码生成定义变量的最佳做法” (第 4-3 页)
- “变量属性的重新赋值” (第 4-7 页)
- “通过不同属性重用同一变量” (第 4-8 页)
- “支持的变量类型” (第 4-11 页)

### 用于代码生成的变量定义

在 MATLAB 中，变量可以在运行时动态更改其属性，因此您可以使用同一个变量来保留任何类、大小或复/实性的值。例如，以下代码可在 MATLAB 中正常运行：

```
function x = foo(c) %#codegen
if(c>0)
    x = 0;
else
    x = [1 2 3];
end
disp(x);
end
```

然而，像 C 这样的静态类型语言必须能够在编译时确定变量属性。因此，对于 C/C++ 代码生成，在使用变量之前，必须在 MATLAB 源代码中显式定义变量的类、大小和复/实性。例如，用 x 的定义重写上述源代码：

```
function x = foo(c) %#codegen
x = zeros(1,3);
if(c>0)
    x = 0;
else
    x = [1 2 3];
end
disp(x);
end
```

### 另请参阅

### 相关示例

- “为 C/C++ 代码生成定义变量的最佳做法”（第 4-3 页）

# 为 C/C++ 代码生成定义变量的最佳做法

本节内容
“在使用变量之前通过赋值来定义变量” （第 4-3 页）
“对变量重新赋值时务必小心” （第 4-4 页）
“在变量定义中使用类型转换运算符” （第 4-5 页）
“在对索引变量赋值之前定义矩阵” （第 4-5 页）
“使用常量值向量对数组进行索引” （第 4-5 页）

## 在使用变量之前通过赋值来定义变量

对于 C/C++ 代码生成，应显式且明确地定义变量的类、大小和复/实性，然后才能在运算中使用这些变量或将它们作为输出返回。通过赋值定义变量，但请注意，赋值不仅会复制值，还会将该值表示的大小、类和复/实性复制到新变量。例如：

赋值	定义：
<code>a = 14.7;</code>	将 <b>a</b> 定义为双精度实数标量。
<code>b = a;</code>	使用 <b>a</b> （双精度实数标量）的属性定义 <b>b</b> 。
<code>c = zeros(5,2);</code>	将 <b>c</b> 定义为由双精度实数值构成的 5×2 数组。
<code>d = [1 2 3 4 5; 6 7 8 9 0];</code>	将 <b>d</b> 定义为由双精度实数值构成的 5×2 数组。
<code>y = int16(3);</code>	将 <b>y</b> 定义为 16 位整数实数标量。

以这种方式定义属性，在 C/C++ 代码生成期间便会在所需的执行路径上定义该变量。

赋给变量的数据可以是标量、矩阵或结构体。如果您的变量是结构体，请显式定义每个字段的属性。

将新变量初始化为已分配数据的值有时会导致生成的代码中出现冗余副本。要避免冗余副本，可以使用 `coder.nullcopy` 构造来定义变量而不初始化其值，如 “Eliminate Redundant Copies of Variables in Generated Code” 中所述。

定义变量时，它们默认为局部变量；它们不会在函数调用之间持久存在。要使变量持久存在，请参阅 `persistent`。

### 例 4.1. 为多个执行路径定义一个变量

以如下 MATLAB 代码为例：

```
...
if c > 0
    x = 11;
end
% Later in your code ...
if c > 0
    use(x);
end
...
```

此处，`x` 仅在 `c > 0` 时赋值，且仅在 `c > 0` 时使用。此代码可在 MATLAB 中正常工作，但在代码生成期间会生成编译错误，因为它检测到 `x` 在某些执行路径上未定义（当 `c <= 0` 时）。

要使此代码适合代码生成，请在使用 `x` 前先对其进行定义：

```
x = 0;
...
if c > 0
    x = 11;
end
% Later in your code ...
if c > 0
    use(x);
end
...
```

### 例 4.2. 定义结构体中的字段

以如下 MATLAB 代码为例：

```
...
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    s.b = 12;
end
% Try to use s
use(s);
...
```

此处，`if` 语句的第一部分仅使用字段 `a`，`else` 子句使用字段 `a` 和 `b`。此代码可在 MATLAB 中正常工作，但在 C/C++ 代码生成期间会生成编译错误，因为它检测到结构体类型不匹配。为了防止此错误，请不要在对某结构体执行某些操作后向该结构体添加字段。有关详细信息，请参阅“用于代码生成的结构体定义”（第 7-2 页）。

要使此代码适合 C/C++ 代码生成，请在使用 `s` 之前定义其所有字段。

```
...
% Define all fields in structure s
s = struct('a',0,'b',0);
if c > 0
    s.a = 11;
    disp(s);
else
    s.a = 12;
    s.b = 12;
end
% Use s
use(s);
...
```

### 对变量重新赋值时务必小心

通常，对于 C/C++ 代码生成，您应该遵循“一个变量/一个类型”规则；也就是说，每个变量必须具有特定的类、大小和复/实性。通常，如果在初始赋值后对变量属性重新赋值，则在代码生成期间会出现编译错误，但也有例外，如“变量属性的重新赋值”（第 4-7 页）中所述。

## 在变量定义中使用类型转换运算符

默认情况下，常量的类型为 `double`。要定义其他类型的变量，可以在变量定义中使用类型转换运算符。例如，以下代码将变量 `y` 定义为整数：

```
...
x = 15; % x is of type double by default.
y = uint8(x); % y has the value of x, but cast to uint8.
...
```

## 在对索引变量赋值之前定义矩阵

从 MATLAB 生成 C/C++ 代码时，不能通过写入超出变量当前大小的元素来扩展变量。这种索引操作会产生运行时错误。必须先定义矩阵，然后再为矩阵的元素赋值。

例如，代码生成不允许以下初始赋值：

```
g(3,2) = 14.6; % Not allowed for creating g
           % OK for assigning value once created
```

有关索引矩阵的详细信息，请参阅“在代码生成的矩阵索引操作方面与 MATLAB 的不兼容性”（第 6-12 页）。

## 使用常量值向量对数组进行索引

最佳做法是使用常量值向量对数组进行索引，而不是使用包含非常量对象的范围。

在某些情况下，代码生成器无法确定包含 `colon` 运算符的表达式是固定大小还是可变大小。请使用常量值向量对数组进行索引，以防止它们在生成的代码中不必要地创建为可变大小数组。

例如，数组 `out` 是使用通过随机行向量 `A` 进行索引的变量 `i` 创建的。

```
...
% extract elements i through i+5 for processing
A = rand(1,10);
out = A(i:i+5); % If i is unknown at compile time, out is variable-size
...
```

如果 `i` 是编译时常量值，代码生成器将为 `out` 生成固定大小对象。如果 `i` 在编译时未知，代码生成器将在生成的代码中为 `out` 生成可变大小数组。

为了防止代码生成器创建可变大小数组，上述代码段重写为以下模式：

```
...
% extract elements i through i+5 for processing
A = rand(1,10);
out = A (i+(0:5)); % out is fixed-size, even if i is unknown at compile time
...
```

此模式能够在迭代器值在编译时未知的情况下生成固定大小数组。推荐重写的另一个示例如下：

```
width = 25;
A = A(j-width:j+width); % A is variable-size, if j is unknown at compile time
fsA = A(j+(-width:width)); % This makes A fixed-size, even if j is unknown at compile time
...
```

### 另请参阅

`coder.nullcopy` | `persistent`

### 详细信息

- “Eliminate Redundant Copies of Variables in Generated Code”
- “用于代码生成的结构体定义” (第 7-2 页)
- “在代码生成的矩阵索引操作方面与 MATLAB 的不兼容性” (第 6-12 页)
- “Avoid Data Copies of Function Inputs in Generated Code”

## 变量属性的重新赋值

对于 C/C++ 代码生成，有一些变量可以在初始赋值后用不同类、大小或复/实性重新赋值：

### 动态大小的变量

变量可以保留具有相同类和复/实性、但大小不同的值。如果初始赋值的大小不是常量，则在生成的代码中变量是动态调整大小的。有关详细信息，请参阅“可变大小数据”。

### 在代码中出于不同目的重用的变量

如果每次出现的变量只能有一种类型，则可以在初始赋值后重新赋值变量的类型（类、大小和复/实性）。在这种情况下，变量在生成的代码中会重命名，以创建多个自变量。有关详细信息，请参阅“通过不同属性重用同一变量”（第 4-8 页）。

通过不同属性重用同一变量

本节内容
“可通过不同属性重用同一变量的情形” （第 4-8 页）
“无法重用变量的情形” （第 4-8 页）
“变量重用的限制” （第 4-9 页）

对于 C/C++ 代码生成，有一些变量可以在初始赋值后用不同类、大小或复/实性重新赋值。变量可以保留具有相同类和复/实性、但大小不同的值。如果初始赋值的大小不是常量，则在生成的代码中变量是动态调整大小的。有关详细信息，请参阅“可变大小数据”。

如果每次出现的变量只能有一种类型，则可以在初始赋值后重新赋值变量的类型（类、大小和复/实性）。在这种情况下，变量在生成的代码中会重命名，以创建多个自变量。

可通过不同属性重用同一变量的情形

如果代码生成器能够在 C/C++ 代码生成过程中明确地确定一个输入、输出或局部变量的每 wh 实例的属性，则可以不同的类、大小或复/实性重用（或重新赋值）该变量。如果是这种情况，MATLAB 可在生成的代码中创建单独的唯一命名的局部变量。您可以在代码生成报告中查看这些重命名的变量。

变量重用的一个常见示例是在 if-elseif-else 或 switch-case 语句中。例如，以下函数 example1 首先在 if 语句中使用变量 t，其中它保留一个双精度标量，然后在 if 语句之外重用 t 来保留一个双精度向量。

```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
```

无法重用变量的情形

如果无法在代码生成过程中明确地确定一个变量的实例的类、大小和复/实性，则无法重用（重新赋值）该变量。在这种情况下，无法重命名变量，并且会出现编译错误。

例如，以下 example2 函数为 if 语句中的 x 赋予一个定点值，并重用 x 以在 else 子句中存储一个双精度矩阵。然后，它又在 if-else 语句之后使用了 x。此函数会生成编译错误，因为在 if-else 语句后，变量 x 可能有不同属性（具体取决于执行哪个 if-else 子句）。

```
function y = example2(use_fixpoint, data) %#codegen
if use_fixpoint
    % x is fixed-point
    x = fi(data, 1, 12, 3);
else
    % x is a matrix of doubles
    x = data;
end
% When x is reused here, it is not possible to determine its
% class, size, and complexity
t = sum(sum(x));
```



```
y = t > 0;
end
```

例 4.3. if 语句中的变量重用

要查看 MATLAB 如何重命名重用的变量 `t`，请执行下列步骤：

- 1 创建包含以下代码的 MATLAB 文件 `example1.m`。

```
function y = example1(u) %#codegen
if all(all(u>0))
    % First, t is used to hold a scalar double value
    t = mean(mean(u)) / numel(u);
    u = u - t;
end
% t is reused to hold a vector of doubles
t = find(u > 0);
y = sum(u(t(2:end-1)));
end
```

- 2 为 `example1` 生成一个 MEX 函数，并生成代码生成报告。

```
codegen -o example1x -report example1.m -args {ones(5,5)}
```

- 3 打开代码生成报告。

在**变量**选项卡上，您可以看到两个唯一命名的局部变量 `t>1` 和 `t>2`。

SUMMARY		ALL MESSAGES (0)		BUILD LOGS		COD
Name		Type	Size	Class		
y		Output	1 × 1	double		
u		Input	5 × 5	double		
t > 1		Local	1 × 1	double		
t > 2		Local	:25 × 1	double		

- 4 在变量列表中，点击 `t>1`。该报告突出显示变量 `t` 在 `if` 语句中的实例。`t` 的这些实例是双精度标量。
- 5 点击 `t>2`。代码生成报告突出显示 `t` 在 `if` 语句之外的实例。`t` 的这些实例是可变大小的列向量，其上界为 25。

变量重用的限制

无法在生成的代码中重命名以下变量：

- 持久变量。
- 全局变量。
- 使用 `coder.ref`、`coder.rref`、`coder.wref` 传递给 C 代码的变量。
- 使用 `coder.varsize` 设置其大小的变量。
- 使用 MATLAB Coder 生成代码时，其名称由 `coder.cstructname` 控制的变量。
- 在 `for` 循环的循环体中使用的索引变量。
- Simulink 模型中 MATLAB Function 模块的模块输出。

- Stateflow<sup>®</sup> 图中 MATLAB 函数的图自有变量。

## 支持的变量类型

您可以使用以下数据类型从 MATLAB 生成 C/C++ 代码：

类型	描述
<code>char</code>	字符数组
<code>complex</code>	复数数据。转换函数接受实部和虚部
<code>double</code>	双精度浮点
<code>int8</code> 、 <code>int16</code> 、 <code>int32</code> 、 <code>int64</code>	有符号整数
<code>logical</code>	布尔 <code>true</code> 或 <code>false</code>
<code>single</code>	单精度浮点
<code>struct</code>	结构体
<code>uint8</code> 、 <code>uint16</code> 、 <code>uint32</code> 、 <code>uint64</code>	无符号整数
定点	定点数据类型



# 为代码生成定义数据

---

- “代码生成的数据定义注意事项”（第 5-2 页）
- “复数数据的代码生成”（第 5-6 页）
- “代码生成中的字符编码”（第 5-10 页）
- “字符串的代码生成”（第 5-11 页）

## 代码生成的数据定义注意事项

要生成高效的独立代码，您必须定义以下不同于在 MATLAB 中运行代码时的数据类型和类。

数据	类型注意事项	更多信息
数组	元素的最大数量受限。	“Array Size Restrictions for Code Generation”
数值类型	在运算中使用数值类型变量或将它们作为输出返回之前，要先对它们赋值。	“为 C/C++ 代码生成定义变量的最佳做法”（第 4-3 页）
复数	<ul style="list-style-type: none"> <li>要在赋值和第一次使用之前设置变量的复/实性。</li> <li>包含复数或变量的表达式的计算结果为复数，即使结果的虚部为零也是如此。</li> </ul>	“复数数据的代码生成”（第 5-6 页）
字符和字符串	<ul style="list-style-type: none"> <li>字符精度限于 8 位。</li> <li>对于代码生成，字符串标量不支持全局变量、花括号索引、缺失值或使用函数 <code>coder.versize</code> 更改大小。</li> </ul>	<ul style="list-style-type: none"> <li>“代码生成中的字符编码”（第 5-10 页）</li> <li>“字符串的代码生成”（第 5-11 页）</li> </ul>
可变大小数据	在对变量进行初始固定大小赋值后，尝试增大变量可能导致编译错误。	<ul style="list-style-type: none"> <li>“可变大小数组的代码生成”（第 6-2 页）</li> <li>“为代码生成定义可变大小数据”（第 6-4 页）</li> </ul>
结构体	<ul style="list-style-type: none"> <li>在每个控制路径中均须以相同的顺序将字段分配给结构体。</li> <li>对结构体数组元素中的对应字段须指定相同的大小、类型和复/实性</li> </ul>	<ul style="list-style-type: none"> <li>“为代码生成定义标量结构体”（第 7-4 页）</li> <li>“为代码生成定义结构体数组”（第 7-6 页）</li> </ul>
元胞数组	<ul style="list-style-type: none"> <li>在将元胞数组传递给函数或从函数返回之前，须为所有元胞数组元素赋值。</li> <li>可变大小的元胞数组元素必须具有相同的大小、类型和复/实性。</li> </ul>	<ul style="list-style-type: none"> <li>“Code Generation for Cell Arrays”</li> <li>“代码生成的元胞数组限制”（第 9-2 页）</li> </ul>

数据	类型注意事项	更多信息
表	<ul style="list-style-type: none"> <li>基于输入数组创建表时，必须使用 '<b>VariableNames</b>' 名称-值参数指定变量名称。</li> <li>使用表函数和 '<b>Size</b>' 名称-值参数预分配表时，数据类型支持存在一定限制。</li> <li>指定变量的表索引必须为编译时常量。</li> <li>您无法通过赋值来更改表的大小。</li> <li>创建表后，无法更改表的 <b>VariableNames</b>、<b>RowNames</b>、<b>DimensionNames</b> 或 <b>UserData</b> 属性。</li> </ul> <p>适用于类的限制也适用于表。</p>	<ul style="list-style-type: none"> <li>“表的代码生成”（第 12-2 页）</li> <li>“Table Limitations for Code Generation”</li> </ul>
分类数组	<p>分类数组不支持以下输入和操作：</p> <ul style="list-style-type: none"> <li>MATLAB 对象数组</li> <li>稀疏矩阵</li> <li>重复的类别名称</li> <li>通过赋值实现增长</li> <li>添加类别</li> <li>删除元素</li> </ul> <p>适用于类的限制也适用于分类数组。</p>	<ul style="list-style-type: none"> <li>“Code Generation for Categorical Arrays”</li> <li>“Categorical Array Limitations for Code Generation”</li> </ul>
日期时间数组	<p><b>datetime</b> 数组不支持以下输入和操作：</p> <ul style="list-style-type: none"> <li>文本输入</li> <li>'<b>Format</b>' 名称-值参数</li> <li>'<b>TimeZone</b>' 名称-值参数和 '<b>TimeZone</b>' 属性</li> <li>设置时间分量属性</li> <li>通过赋值实现增长</li> <li>删除元素</li> </ul> <p>适用于类的限制也适用于 <b>datetime</b> 数组。</p>	<ul style="list-style-type: none"> <li>“Code Generation for Datetime Arrays”</li> <li>“代码生成的日期时间数组限制”（第 10-2 页）</li> </ul>

数据	类型注意事项	更多信息
持续时间数组	<p>持续时间数组不支持以下输入和操作：</p> <ul style="list-style-type: none"> <li>• 文本输入</li> <li>• 通过赋值实现增长</li> <li>• 删除元素</li> <li>• 使用 <code>char</code>、<code>cellstr</code> 或 <code>string</code> 函数将持续时间值转换为文本</li> </ul> <p>适用于类的限制也适用于持续时间数组。</p>	<ul style="list-style-type: none"> <li>• “持续时间数组的代码生成” (第 11-2 页)</li> <li>• “代码生成的持续时间数组限制” (第 11-6 页)</li> </ul>
时间表	<ul style="list-style-type: none"> <li>• 基于输入数组创建时间表时，必须使用 '<code>VariableNames</code>' 名称-值参数指定变量名称。</li> <li>• 使用时间表函数和 '<code>Size</code>' 名称-值参数预分配表时，数据类型支持存在一定限制。</li> <li>• 指定变量的时间表索引必须为编译时常量。</li> <li>• 您无法通过赋值来更改时间表的大小。</li> <li>• 创建时间表后，无法更改其 <code>VariableNames</code>、<code>DimensionNames</code> 或 <code>UserData</code> 属性。</li> <li>• 如果您创建规则时间表，并尝试设置不规则的行时间，则会生成错误。</li> <li>• 如果您创建不规则时间表，则即使您设置其采样率或时间步，它仍是不规则的。</li> </ul> <p>适用于类的限制也适用于时间表。</p>	<ul style="list-style-type: none"> <li>• “Code Generation for Timetables”</li> <li>• “Timetable Limitations for Code Generation”</li> </ul>
枚举数据：	仅支持基于整数的枚举类型。	“枚举”



数据	类型注意事项	更多信息
MATLAB 类	<ul style="list-style-type: none"> <li>在生成代码之前，最好通过在整个输入值范围内运行 MEX 函数来测试类属性验证。</li> <li>如果属性没有显式初始值，代码生成器将假定它未在构造函数的起始部分进行定义。代码生成器不会指定空矩阵作为默认值。</li> <li>类属性不支持 <code>coder.varsize</code> 函数。</li> <li>如果一个属性的初始值是对象，则该属性必须是常量。要使属性为常量，可在属性模块中声明常量特性。</li> </ul>	<ul style="list-style-type: none"> <li>“Generate C++ Classes for MATLAB Classes”</li> <li>“用于代码生成的 MATLAB 类定义”（第 15-2 页）</li> </ul>
函数句柄	<ul style="list-style-type: none"> <li>将不同函数句柄赋给同一变量会导致编译时错误。</li> <li>无法向入口函数或外部函数传递函数句柄，也无法从入口函数或外部函数传递函数句柄。</li> <li>无法从 MATLAB Function 模块调试器中查看函数句柄。</li> </ul>	“函数句柄”
深度学习数组	<p><b>dlarrays</b> 不支持以下输入和操作：</p> <ul style="list-style-type: none"> <li>数据格式参数必须为编译时常量</li> <li>在入口函数中定义 <b>dlarray</b> 变量。</li> <li><b>dlarray</b> 的输入必须为固定大小。</li> <li>代码生成不支持使用指定了上界大小和可变维度的 <b>coder.typeof</b> 函数创建 <b>dlarray</b> 类型对象。</li> </ul>	<ul style="list-style-type: none"> <li>“Code Generation for dlarray”</li> <li>“dlarray Limitations for Code Generation”</li> </ul>

上表并未详尽列出每种数据类型注意事项。请参阅“详细信息”列中的主题。

## 另请参阅

## 相关示例

- “为 C/C++ 代码生成定义变量的最佳做法”（第 4-3 页）
- “通过不同属性重用同一变量”（第 4-8 页）
- “Eliminate Redundant Copies of Variables in Generated Code”

# 复数数据的代码生成

本节内容
“定义复变量时的限制” （第 5-6 页）
“具有零值虚部的复数数据的代码生成” （第 5-6 页）
“含复数操作数的表达式的结果” （第 5-8 页）
“具有非有限值的复数乘法的结果” （第 5-9 页）

## 定义复变量时的限制

对于代码生成，您必须在赋值时设置变量的复/实性。使用复数常量为变量赋值或使用 `complex` 函数。例如：

```
x = 5 + 6i; % x is a complex number by assignment.
y = complex(5,6); % y is the complex number 5 + 6i.
```

赋值后，您不能更改该变量的复/实性。以下函数的代码生成失败，因为  $x(k) = 3 + 4i$  更改了  $x$  的复/实性。

```
function x = test1()
x = zeros(3,3); % x is real
for k = 1:numel(x)
    x(k) = 3 + 4i;
end
end
```

要解决此问题，请用复数常量为  $x$  赋值。

```
function x = test1()
x = zeros(3,3)+ 0i; %x is complex
for k = 1:numel(x)
    x(k) = 3 + 4i;
end
end
```

## 具有零值虚部的复数数据的代码生成

在代码生成中，虚部均为零值的复数数据仍为复数。该数据不会变为实数。这种行为意味着：

- 在某些情况下，按绝对值对复数数据排序的函数所生成的结果可能与 MATLAB 结果不同。请参阅“按绝对值对复数值排序的函数”（第 5-6 页）。
- 对于要求按绝对值对复数输入排序的函数，具有零值虚部的复数输入必须按绝对值排序。这些函数包括 `ismember`、`union`、`intersect`、`setdiff` 和 `setxor`。

### 按绝对值对复数值排序的函数

按绝对值对复数值排序的函数包括 `sort`、`issorted`、`sortrows`、`median`、`min` 和 `max`。这些函数按绝对值对复数排序，即使虚部为零也是如此。一般来说，按绝对值排序的结果不同于按实部排序的结果。因此，如果这些函数的输入在生成的代码中是具有零值虚部的复数，而在 MATLAB 中是实数，则生成的代码会产生与 MATLAB 不同的结果。在以下示例中，`sort` 的输入在 MATLAB 中是实数，而在生成的代码中是具有零值虚部的复数：

- 您将实数输入传递给生成的使用复数输入的函数

- 1 编写以下函数：

```
function myout = mysort(A)
myout = sort(A);
end
```

- 2 在 MATLAB 中调用 mysort。

```
A = -2:2;
mysort(A)

ans =

    -2    -1     0     1     2
```

- 3 生成一个使用复数输入的 MEX 函数。

```
A = -2:2;
codegen mysort -args {complex(A)} -report
```

- 4 使用实数输入调用该 MEX 函数。

```
mysort_mex(A)

ans =

     0     1    -1     2    -2
```

生成的 MEX 函数使用复数输入，因此，它将实数输入视为具有零值虚部的复数。它按复数的绝对值对这些数进行排序。由于虚部为零，MEX 函数会将结果以实数形式返回给 MATLAB 工作区。请参阅“生成的复数参数 MEX 函数的输入和输出”（第 5-8 页）。

- sort 的输入是在生成的代码中返回复数的函数的输出

- 1 编写以下函数：

```
function y = myfun(A)
x = eig(A);
y = sort(x,'descend');
```

eig 的输出是 sort 的输入。在生成的代码中，eig 返回复数结果。因此，在生成的代码中，x 是复数。

- 2 在 MATLAB 中调用 myfun。

```
A = [2 3 5;0 5 5;6 7 4];
myfun(A)

ans =

    12.5777
     2.0000
    -3.5777
```

eig 的结果是实数。因此，sort 的输入是实数。

- 3 生成一个使用复数输入的 MEX 函数。

```
codegen myfun -args {complex(A)}
```

- 4 调用该 MEX 函数。

```
myfun_mex(A)

ans =

    12.5777
   -3.5777i
    2.0000
```

在该 MEX 函数中，`eig` 返回复数结果。因此，`sort` 的输入是复数。MEX 函数按绝对值降序对输入进行排序。

生成的复数参数 MEX 函数的输入和输出

对于由 `codegen` 命令、`fiaccel` 命令或 MATLAB Coder 创建的 MEX 函数：

- 假设您生成使用复数输入的 MEX 函数。如果使用实数输入调用该 MEX 函数，MEX 函数会将实数输入转换为具有零值虚部的复数值。
- 如果 MEX 函数返回虚部均为零值的复数值，则 MEX 函数将这些值以实数值返回到 MATLAB 工作区。以如下函数为例：

```
function y = foo()
    y = 1 + 0i; % y is complex with imaginary part equal to zero
end
```

如果您为 `foo` 生成一个 MEX 函数并查看代码生成报告，会看到 `y` 是复数。

```
codegen foo -report
```

Name	Type	Size	Class
y	Output	1 × 1	complex double

如果运行该 MEX 函数，可以看到在 MATLAB 工作区中，`foo_mex` 的结果是实数值 1。

```
z = foo_mex

ans =

    1
```

含复数操作数的表达式的结果

一般来说，包含一个或多个复数操作数的表达式会在生成的代码中产生复数结果，即使结果的值为零也是如此。以如下代码行为例：

```
z = x + y;
```

假设在运行时，`x` 的值为  $2 + 3i$ ，`y` 的值为  $2 - 3i$ 。在 MATLAB 中，此代码产生实数结果  $z = 4$ 。在代码生成中，`x` 和 `y` 的类型是已知的，但其值是未知的。由于此表达式的两个操作数中至少有一个是复数，因此将 `z` 定义为需要存储实部和虚部的复变量。`z` 等于生成的代码中的复数结果  $4 + 0i$ ，而不是 MATLAB 代码中的结果 4。

这种行为的例外是：

- 当复数结果的虚部为零时，MEX 函数将结果以实数值返回到 MATLAB 工作区。请参阅“生成的复数参数 MEX 函数的输入和输出”（第 5-8 页）。
- 当参数的虚部为零时，外部函数的复数参数是实数。

```
function y = foo()
    coder.extrinsic('sqrt')
    x = 1 + 0i; % x is complex
    y = sqrt(x); % x is real, y is real
end
```

- 使用复数参数但产生实数结果的函数会返回实数值。

```
y = real(x); % y is the real part of the complex number x.
y = imag(x); % y is the real-valued imaginary part of x.
y = isreal(x); % y is false (0) for a complex number x.
```

- 接受实数参数但结果为复数的函数会返回复数值。

```
z = complex(x,y); % z is a complex number for a real x and y.
```

## 具有非有限值的复数乘法的结果

当复数乘法的操作数包含非有限值时，生成的代码产生的结果可能与 MATLAB 产生的结果不同。产生差异的原因在于代码生成定义复数乘法的方式。对于代码生成：

- 复数值乘以复数值  $(a + bi)(c + di)$  定义为  $(ac - bd) + (ad + bc)i$ 。即使实部或虚部为零，也会执行完整的计算。
- 实数值乘以复数值  $c(a + bi)$  定义为  $ca + cbi$ 。

## 代码生成中的字符编码

MATLAB 以 16 位 Unicode 表示字符。代码生成器以区域设置确定的 8 位代码集表示字符。MATLAB 和代码生成之间的字符编码差异会导致以下后果：

- 对编码数值大于 255 的字符，代码生成会出现错误。
- 对于 128-255 范围内的某些字符，可能无法以区域设置的代码集表示字符，或无法将该字符转换为等效的 16 位 Unicode 字符。在 MATLAB 和生成的代码之间传递此范围内的字符可能导致错误或产生不同的结果。
- 对于代码生成，某些工具箱函数只接受 7 位 ASCII 字符。
- 如果将不在 7 位 ASCII 代码集中的字符转换为数值类型（例如双精度类型），则在生成的代码中得到的结果可能与在 MATLAB 中得到的结果有所不同。对于代码生成，最好避免对字符执行算术运算。

### 另请参阅

#### 详细信息

- “国际化的区域设置概念”
- “生成的代码和 MATLAB 代码之间的差异”（第 2-6 页）

## 字符串的代码生成

代码生成支持 1×1 MATLAB 字符串数组。代码生成不支持包含多个元素的字符串数组。

1×1 字符串数组（称为字符串标量）包含一段文本，表示为 1×n 字符向量。字符串标量的一个示例是 "Hello, world"。有关字符串的详细信息，请参阅“字符串数组和字符数组中的文本”。

### 限制

对于字符串标量，代码生成不支持：

- 全局变量
- 用花括号 {} 进行索引
- 缺失值
- 使用 `codegen` 命令、`fiaccel` 命令或 MATLAB Coder 生成代码时，通过对 `assert` 语句使用预条件以编程方式定义输入类型
- 使用 `codegen` 命令、`fiaccel` 命令或 MATLAB Coder 生成代码时与 `coder.varsize` 结合使用
- 它们用作 Simulink 信号、参数或数据存储内存

对于代码生成，适用于类的限制也适用于字符串。请参阅“用于代码生成的 MATLAB 类定义”（第 15-2 页）。

### 生成的代码和 MATLAB 代码之间的差异

- 如果将包含多个一元运算符的字符串转换为 `double`，则在 MATLAB 和生成的代码之间可能产生不同结果。以如下函数为例：

```
function out = foo(op)
out = double(op + 1);
end
```

对于输入值 "1"，该函数将字符串 "1" 转换为 `double`。在 MATLAB 中，结果为 NaN。在生成的代码中，结果为 1。

- 对具有错位逗号（逗号不用作千位分隔符）的字符串进行双精度值转换可能会产生与 MATLAB 不同的结果。

### 另请参阅

### 详细信息

- “Define String Scalar Inputs”





# 可变大小数据的代码生成

---

- “可变大小数组的代码生成” (第 6-2 页)
- “为代码生成定义可变大小数据” (第 6-4 页)
- “在代码生成的可变大小支持方面与 MATLAB 的不兼容性” (第 6-8 页)

## 可变大小数组的代码生成

对于代码生成，数组维度为固定大小或可变大小。如果代码生成器可以确定维度的大小并且维度的大小不会更改，则维度为固定大小。当数组的所有维度都是固定大小时，该数组是固定大小数组。在以下示例中，`Z` 是固定大小数组。

```
function Z = myfcn()
Z = zeros(1,4);
end
```

第一个维度的大小为 1，第二个维度的大小为 4。

如果代码生成器不能确定维度的大小，或者代码生成器确定大小发生了更改，则维度为可变大小。如果数组至少有一个维度的大小是可变的，则数组是可变大小数组。

可变大小维度可以有界，也可以无界。有界维度具有固定的大小上限。无界维度没有固定的大小上限。

在以下示例中，`Z` 的第二个维度为有界可变大小。它的上界为 16。

```
function s = myfcn(n)
if (n > 0)
    Z = zeros(1,4);
else
    Z = zeros(1,16);
end
s = length(Z);
```

在以下示例中，如果 `n` 的值在编译时未知，则 `Z` 的第二个维度为无界。

```
function s = myfcn(n)
Z = rand(1,n);
s = sum(Z);
end
```

您可以通过以下方式定义可变大小数组：

- 使用具有非常量维度的构造函数（例如 `zeros`）
- 在使用之前为同一变量分配多个常量大小
- 使用 `coder.varsize` 将变量的所有实例声明为可变大小

有关详细信息，请参阅“为代码生成定义可变大小数据”（第 6-4 页）。

您可以控制是否允许使用可变大小数组进行代码生成。请参阅“启用和禁用对可变大小数组的支持”（第 6-3 页）。

## 可变大小数组的内存分配

对于其大小小于阈值的固定大小数组和可变大小数组，代码生成器在堆栈上静态分配内存。对于无界可变大小数组和其大小大于或等于阈值的可变大小数组，代码生成器在堆上动态分配内存。

您可以控制是否允许动态内存分配，以及何时在代码生成中使用动态内存分配。请参阅“Control Memory Allocation for Variable-Size Arrays”。

代码生成器将动态分配的数据表示为一种称为 `emxArray` 的结构体类型。代码生成器生成工具函数，这些函数用于创建 `emxArray` 并与之交互。如果您使用 Embedded Coder，则可以自定义 `emxArray` 类型和工具函数的生成标识符。请参阅“Identifier Format Control” (Embedded Coder)。

启用和禁用对可变大小数组的支持

默认情况下，对可变大小数组的支持处于启用状态。要修改此支持，请执行下列操作：

- 在代码配置对象中，将 `EnableVariableSizing` 参数设置为 `true` 或 `false`。
- 在 MATLAB Coder App 中，在内存设置中，选中或清除启用可变大小复选框。

代码生成报告中的可变大小数组

通过查看代码生成报告中 **Variables** 选项卡的 **Size** 列，可以判断数组是固定大小还是可变大小。

Name	Type	Size	Class
y	Output	1 × 1	double
A	Input	1 × :16	char
n	Input	1 × 1	double
X	Local	1 × :?	double

冒号 (:) 表示维度为可变大小。问号 (?) 表示大小为无界。例如，大小为 1×:? 表示第一个维度的大小是固定大小 1，第二个维度的大小是无界可变大小。斜体表示代码生成器生成的是可变大小数组，但数组的大小在执行期间不会更改。

Name	Type	Size	Class
y	Output	1 × :5	double
n	Input	1 × 1	double
Z	Local	<i>1 × 4</i>	double

另请参阅

详细信息

- “Control Memory Allocation for Variable-Size Arrays”
- “Specify Upper Bounds for Variable-Size Arrays”
- “为代码生成定义可变大小数据” (第 6-4 页)

# 为代码生成定义可变大小数据

对于代码生成，在运算中使用变量或以输出形式返回变量之前，必须为它们分配特定的类、大小和复/实性。通常，在初始分配后，您便无法再重新分配变量属性。因此，在为变量或结构体字段分配固定大小后，尝试增大变量或结构体字段可能会导致编译错误。在这些情况下，您必须使用以下方法之一将数据显式定义为可变大小。

方法	参阅
从可变大小矩阵构造函数中分配数据，例如： <ul style="list-style-type: none"><li>• <code>ones</code></li><li>• <code>zeros</code></li><li>• <code>repmat</code></li></ul>	“使用具有非常量维度的矩阵构造函数”（第 6-4 页）
在使用（读取）变量之前为同一变量分配多个常量大小。	“为同一个变量分配多个大小”（第 6-4 页）
将变量的所有实例定义为可变大小。	“使用 <code>coder.varsize</code> 显式定义可变大小数据”（第 6-5 页）

## 使用具有非常量维度的矩阵构造函数

您可以通过使用具有非常量维度的构造函数来定义可变大小矩阵。例如：

```
function s = var_by_assign(u) %#codegen
y = ones(3,u);
s = numel(y);
```

如果不使用动态内存分配，还必须添加 `assert` 语句来提供维度的上界。例如：

```
function s = var_by_assign(u) %#codegen
assert (u < 20);
y = ones(3,u);
s = numel(y);
```

## 为同一个变量分配多个大小

在代码中使用（读取）变量之前，可以通过为其分配多个恒定大小来使其大小可变。当代码生成器在堆栈上使用静态分配时，它会根据为每个维度指定的最大大小来推断上界。当您在所有分配中为给定的维度分配相同的大小时，代码生成器假定该维度固定为该大小。通过这些分配可以指定不同的形状和大小。

当代码生成器使用动态内存分配时，它不检查上界。它假设可变大小数据是无界的。

### 从多个不同形状的定义推断上界

```
function s = var_by_multiassign(u) %#codegen
if (u > 0)
    y = ones(3,4,5);
else
    y = zeros(3,1);
end
s = numel(y);
```

当代码生成器使用静态分配时，它推断 `y` 是三维矩阵：

- 第一个维度的大小固定为 3
- 第二个维度为可变大小，上界为 4
- 第三个维度为可变大小，上界为 5

当代码生成器使用动态分配时，它会以不同方式分析 `y` 的维度：

- 第一个维度的大小固定为 3。
- 第二个和第三个维度是无界的。

## 使用 `coder.varsize` 显式定义可变大小数据

要显式定义可变大小数据，请使用函数 `coder.varsize`。您也可以指定哪些维度大小可变及其上界。例如：

- 将 `B` 定义为可变大小的二维数组，其中每个维度的上界为 64。

```
coder.varsize('B', [64 64]);
```

- 将 `B` 定义为可变大小数组：

```
coder.varsize('B');
```

如果您仅提供第一个参数，`coder.varsize` 将假设 `B` 的所有维度都是可变的，并且上界为 `size(B)`。

### 指定哪些维度可变

您可以使用函数 `coder.varsize` 来指定哪些维度是可变的。例如，以下语句将 `B` 定义为数组，其第一个维度固定为 2，但其第二个维度可以增长到 16：

```
coder.varsize('B',[2, 16],[0 1])
```

.

第三个参数指定哪些维度是可变的。此参数必须为逻辑向量或只包含 0 和 1 的双精度向量。对应于零或 `false` 的维度具有固定大小。对应于 1 或 `true` 的维度的大小是可变的。`coder.varsize` 通常将大小为 1 的维度视为固定大小。请参阅“用单一维度定义可变大小矩阵”（第 6-6 页）。

### 定义固定维度后允许增大变量大小

函数 `var_by_if` 定义矩阵 `Y`，在第一次使用前具有固定的  $2 \times 2$  维度（其中语句 `Y = Y + u` 从 `Y` 读取）。但是，`coder.varsize` 将 `Y` 定义为可变大小矩阵，允许它根据 `else` 子句中的决策逻辑来更改大小：

```
function Y = var_by_if(u) %#codegen
if (u > 0)
    Y = zeros(2,2);
    coder.varsize('Y');
    if (u < 10)
        Y = Y + u;
    end
else
    Y = zeros(5,5);
end
```

如果没有 `coder.varsize`，代码生成器会推断 `Y` 是固定大小的  $2 \times 2$  矩阵。它会生成大小不匹配的错误。

### 用单一维度定义可变大小矩阵

单一维度是 `size(A,dim) = 1` 的维度。在下列情况下，单一维度的大小是固定的：

- 您在 `coder.varsize` 表达式中指定上界为 1 的维度。

例如，在此函数中，`Y` 的行为类似于具有一个可变大小维度的向量：

```
function Y = dim_singleton(u) %#codegen
Y = [1 2];
coder.varsize('Y', [1 10]);
if (u > 0)
    Y = [Y 3];
else
    Y = [Y u];
end
```

- 您使用矩阵构造函数表达式或矩阵函数初始化具有单一维度的可变大小数据。

例如，在此函数中，`X` 和 `Y` 的行为类似于向量，其中只有其第二个维度的大小是可变的。

```
function [X,Y] = dim_singleton_vects(u) %#codegen
Y = ones(1,3);
X = [1 4];
coder.varsize('Y','X');
if (u > 0)
    Y = [Y u];
else
    X = [X u];
end
```

您可以通过使用 `coder.varsize` 显式指定单一维度可变来覆盖此行为。例如：

```
function Y = dim_singleton_vary(u) %#codegen
Y = [1 2];
coder.varsize('Y', [1 10], [1 1]);
if (u > 0)
    Y = [Y Y+u];
else
    Y = [Y Y*u];
end
```

在此示例中，`coder.varsize` 的第三个参数是由 1 组成的向量，表示 `Y` 的每个维度的大小可变。

### 定义可变大小结构体字段

要将结构体字段定义为可变大小数组，请使用冒号 (:) 作为索引表达式。冒号 (:) 表示数组的所有元素的大小都是可变的。例如：

```
function y=struct_example() %#codegen

d = struct('values', zeros(1,0), 'color', 0);
data = repmat(d, [3 3]);
coder.varsize('data(:).values');

for i = 1:numel(data)
    data(i).color = rand-0.5;
    data(i).values = 1:i;
end
```

```
y = 0;
for i = 1:numel(data)
    if data(i).color > 0
        y = y + sum(data(i).values);
    end
end
```

表达式 `coder.varsize('data(:).values')` 将矩阵 `data` 的每个元素中的字段 `values` 定义为可变大小。

以下是其他示例：

- `coder.varsize('data.A(:).B')`

在此示例中，`data` 是包含矩阵 `A` 的标量变量。矩阵 `A` 的每个元素都包含大小可变的字段 `B`。

- `coder.varsize('data(:).A(:).B')`

此表达式将矩阵 `A` 和 `data` 的每个元素内的字段 `B` 定义为可变大小。

## 另请参阅

`coder.varsize` | `coder.typeof`

## 详细信息

- “可变大小数组的代码生成”（第 6-2 页）
- “Specify Upper Bounds for Variable-Size Arrays”

# 在代码生成的可变量大小支持方面与 MATLAB 的不兼容性

本节内容
“在标量扩展方面与 MATLAB 的不兼容性” (第 6-8 页)
“在确定可变量大小 N 维数组的大小方面与 MATLAB 的不兼容性” (第 6-9 页)
“在确定空数组的大小方面与 MATLAB 的不兼容性” (第 6-9 页)
“在确定空数组的类方面与 MATLAB 的不兼容性” (第 6-10 页)
“在矩阵-矩阵索引方面与 MATLAB 的不兼容性” (第 6-11 页)
“在建立向量-向量索引方面与 MATLAB 的不兼容性” (第 6-11 页)
“在代码生成的矩阵索引操作方面与 MATLAB 的不兼容性” (第 6-12 页)
“在串联可变量大小矩阵方面与 MATLAB 的不兼容性” (第 6-12 页)
“当串联内可变量大小元胞数组的花括号索引不返回任何元素时的差异” (第 6-12 页)

## 在标量扩展方面与 MATLAB 的不兼容性

标量扩展是对标量数据进行转换以便与向量或矩阵数据的维度相匹配的一种方法。如果一个操作数是标量，而另一个不是，则标量扩展会将标量应用于另一个操作数中的每个元素。

在代码生成过程中，标量扩展规则同样适用，除非处理的是两个可变量大小表达式。在这种情况下，两个操作数必须具有相同的大小。否则，即使其中一个可变量大小表达式在运行时变为标量，生成的代码也不会执行标量扩展。因此，如果启用了运行时错误检查，可能会发生运行时错误。

以如下函数为例：

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z;
```

当您为此函数生成代码时，代码生成器会将 `z` 确定为一个上界为 3 的可变量大小。

Summary	All Messages (0)	Build Logs	Code Insights (1)	Variables
Name	Type	Size	Class	
y	Output	3 × 3	double	
u	Input	1 × 1	double	
z	Local	3 × 3	double	

如果您运行 `u` 等于 0 或 1 的 MEX 函数，即使 `z` 在运行时为标量，生成的代码也不会执行标量扩展。因此，如果启用了运行时错误检查，可能会发生运行时错误。

```
scalar_exp_test_err1_mex(0)
Subscripted assignment dimension mismatch: [9] ~= [1].
```



```
Error in scalar_exp_test_err1 (line 11)
y(:) = z;
```

为了避免此问题，请使用索引强制  $z$  为标量值：

```
function y = scalar_exp_test_err1(u) %#codegen
y = ones(3);
switch u
    case 0
        z = 0;
    case 1
        z = 1;
    otherwise
        z = zeros(3);
end
y(:) = z(1);
```

## 在确定可变大小 N 维数组的大小方面与 MATLAB 的不兼容性

对于可变大小 N 维数组，`size` 函数可能在生成的代码中返回与 MATLAB 不同的结果。在生成的代码中，`size(A)` 返回固定长度输出，因为它不会舍弃可变大小 N 维数组的尾部单一维度。与之相比，MATLAB 中的 `size(A)` 则返回可变长度输出，因为它舍弃了尾部单一维度。

例如，如果数组  $A$  的形状为  $:?x:?x:?x$  和 `size(A,3)==1`，则 `size(A)`：

- 在生成的代码中返回三元素向量
- 在 MATLAB 代码中返回二元素向量

### 解决办法

如果您的应用程序要求生成的代码返回与 MATLAB 代码大小相同的可变大小 N 维数组，请考虑以下解决办法之一：

- 使用具有两个参数的 `size`。

例如，`size(A,n)` 在生成的代码和 MATLAB 代码中返回相同的答案。

- 重写 `size(A)`：

```
B = size(A);
X = B(1:ndims(A));
```

此版本返回具有可变长度输出的  $X$ 。但是，不能将可变大小  $X$  传递给需要固定大小参数的矩阵构造函数，例如 `zeros`。

## 在确定空数组的大小方面与 MATLAB 的不兼容性

空数组的大小在生成的代码中可能与 MATLAB 源代码中不同。此大小在生成的代码中可能为  $1 \times 0$  或  $0 \times 1$ ，但在 MATLAB 中可能为  $0 \times 0$ 。因此，不应编写依赖特定大小的空矩阵的代码。

以如下代码为例：

```
function y = foo(n) %#codegen
x = [];
i = 0;
```

```

while (i < 10)
    x = [5 x];
    i = i + 1;
end
if n > 0
    x = [];
end
y = size(x);
end

```

串联要求其操作数与未串联的维度的大小相匹配。在前面的串联中，标量值的大小为  $1 \times 1$ ， $x$  的大小为  $0 \times 0$ 。为了支持此用例，代码生成器将  $x$  的大小确定为  $[1 \times ?]$ 。因为串联之后还有一次赋值  $x = []$ ，所以在生成的代码中， $x$  的大小为  $1 \times 0$  而不是  $0 \times 0$ 。

在确定表示为 " 的空字符向量的大小时，此行为会持续存在。以如下代码为例：

```

function out = string_size
out = size("");
end

```

此处，`out` 的值在生成代码中可能是  $1 \times 0$  或  $0 \times 1$ ，但在 MATLAB 中是  $0 \times 0$ 。

要了解在确定因删除数组元素而产生的空数组的大小时与 MATLAB 的不兼容性，请参阅“删除数组元素所生成的空数组的大小”（第 2-13 页）。

## 解决方法

如果您的应用程序会检查矩阵是否为空，请使用以下解决办法之一：

- 重写代码以使用 `isempty` 函数，而不是 `size` 函数。
- 不要使用 `x=[]` 创建空数组，而应使用 `zeros` 创建具有特定大小的空数组。例如：

```

function y = test_empty(n) %#codegen
x = zeros(1,0);
i=0;
while (i < 10)
    x = [5 x];
    i = i + 1;
end
if n > 0
    x = zeros(1,0);
end
y=size(x);
end

```

## 在确定空数组的类方面与 MATLAB 的不兼容性

空数组的类在生成的代码中可能与 MATLAB 源代码中不同。因此，不应编写依赖空矩阵的类的代码。

以如下代码为例：

```

function y = fun(n)
x = [];
if n > 1
    x = ['a' x];
end

```

```
y=class(x);
end
```

`fun(0)` 在 MATLAB 中返回 `double`，但在生成的代码中返回 `char`。当语句 `n > 1` 为 `false` 时，MATLAB 不会执行 `x = ['a' x]`。`x` 的类为 `double`，即空数组的类。但是，代码生成器将考虑所有执行路径。它基于语句 `x = ['a' x]` 确定 `x` 的类为 `char`。

### 解决方法

不要使用 `x=[]` 创建空数组，而应创建特定类的空数组。例如，使用 `blanks(0)` 创建空字符数组。

```
function y = fun(n)
x = blanks(0);
if n > 1
    x = ['a' x];
end
y=class(x);
end
```

## 在矩阵-矩阵索引方面与 MATLAB 的不兼容性

在矩阵-矩阵索引中，您可以使用一个矩阵来索引另一个矩阵。在 MATLAB 中，矩阵-矩阵索引的一般规则是，结果的大小和方向与索引矩阵的大小和方向保持一致。例如，如果 `A` 和 `B` 是矩阵，则 `size(A(B))` 等于 `size(B)`。当 `A` 和 `B` 是向量时，MATLAB 将应用特殊规则。特殊的向量-向量索引规则是，结果的方向等于数据矩阵的方向。例如，如果 `A` 是  $1 \times 5$ ，`B` 是  $3 \times 1$ ，则 `A(B)` 是  $1 \times 3$ 。

代码生成器应用与 MATLAB 相同的矩阵-矩阵索引规则。如果 `A` 和 `B` 是可变大小矩阵，为了应用矩阵-矩阵索引规则，代码生成器会假设 `size(A(B))` 等于 `size(B)`。如果 `A` 和 `B` 在运行时变成向量且方向不同，则该假设不正确。因此，如果启用了运行时错误检查，可能会发生错误。

为了避免此问题，请通过使用冒号操作符进行索引，强制您的数据为向量。例如，假设您的代码在运行时有意要在向量和常规矩阵之间进行切换。您可以对向量-向量索引执行显式检查：

```
...
if isvector(A) && isvector(B)
    C = A(:);
    D = C(B(:));
else
    D = A(B);
end
...
```

第一个分支中的索引指定 `C` 和 `B(:)` 是编译时向量。因此，代码生成器将应用使用一个向量索引另一个向量的索引规则。结果的方向等于数据向量 `C` 的方向。

## 在建立向量-向量索引方面与 MATLAB 的不兼容性

在 MATLAB 中，特殊的向量-向量索引规则是，结果的方向等于数据向量的方向。例如，如果 `A` 是  $1 \times 5$ ，`B` 是  $3 \times 1$ ，则 `A(B)` 是  $1 \times 3$ 。但是，如果数据向量 `A` 是标量，则 `A(B)` 的方向等于索引向量 `B` 的方向。

代码生成器应用与 MATLAB 相同的向量-向量索引规则。如果 `A` 和 `B` 是可变大小向量，为了应用索引规则，代码生成器会假设 `B` 的方向与 `A` 的方向一致。如果 `A` 在运行时变成标量，并且 `A` 和 `B` 的方向不一致，则该假设不正确。因此，如果启用了运行时错误检查，可能会发生运行时错误。

为了避免出现此问题，应使向量的方向保持一致。或者通过指定行和列来索引单个元素。例如，`A(row, column)`。

## 在代码生成的矩阵索引操作方面与 MATLAB 的不兼容性

以下限制适用于代码生成时的矩阵索引操作：

- 初始化以下样式：

```
for i = 1:10
    M(i) = 5;
end
```

在本例中，执行循环时 `M` 的大小会改变。代码生成不支持数组大小随着时间改变而增加。

对于代码生成，需要预分配 `M`。

```
M = zeros(1,10);
for i = 1:10
    M(i) = 5;
end
```

以下限制适用于在禁用动态内存分配的情况下进行代码生成时的矩阵索引操作：

- `M(i:j)`，其中 `i` 和 `j` 在循环中改变

在代码生成过程中，不会为大小随着程序的执行而改变的表达式动态分配内存。要实现这种行为，请使用 `for`- 循环，如下所示：

```
...
M = ones(10,10);
for i=1:10
    for j = i:10
        M(i,j) = 2*M(i,j);
    end
end
...
```

---

**注意** 矩阵 `M` 必须在进入循环之前定义。

---

## 在串联可变大小矩阵方面与 MATLAB 的不兼容性

要进行代码生成，当您串联可变大小数组时，未串联的维度必须完全匹配。

## 当串联内可变大小元胞数组的花括号索引不返回任何元素时的差异

假设：

- `c` 是一个可变大小元胞数组。
- 可以使用花括号访问 `c` 的内容。例如，`c{2:4}`。
- 将结果包含在串联中。例如，`[a c{2:4} b]`。

- $c\{I\}$  不返回任何元素。 $c$  为空或者花括号内的索引生成空结果。

基于这些条件，MATLAB 将忽略串联中的  $c\{I\}$ 。例如， $[a\ c\{I\}\ b]$  变为  $[a\ b]$ 。代码生成器将  $c\{I\}$  视为空数组  $[c\{I\}]$ 。串联变为  $[...[c\{I\}]...]$ 。此串联将忽略  $[c\{I\}]$  数组。因此， $[c\{I\}]$  的属性与串联  $[...[c\{I\}]...]$  兼容，代码生成器将根据以下规则指定  $[c\{I\}]$  的类、大小和复/实性：

- 类和复/实性与元胞数组的基类型相同。
- 第二个维度的大小始终为 0。
- 对于其余的维度， $N_i$  的大小取决于基类型中对应的维度是固定大小还是可变大小。
  - 如果基类型中对应的维度为可变大小，则结果中的维度大小为 0。
  - 如果基类型中对应的维度为固定大小，则结果中的维度也具有该大小。

假设  $c$  的基类型的类为 `int8`，大小为 `:10x7x8x?:`。在生成的代码中， $[c\{I\}]$  的类为 `int8`。 $[c\{I\}]$  的大小为 `0x0x8x0`。第二个维度为 0。第一个和最后一个维度为 0，因为这两个维度在基类型中为可变大小。第三个维度为 8，因为基类型的第三个维度的大小固定为 8。

在串联内部，如果可变大小元胞数组的花括号索引不返回任何元素，生成的代码与 MATLAB 可能存在以下差异：

- 在生成的代码中， $[...c\{i\}...]$  的类可能不同于 MATLAB 中的类。

当  $c\{I\}$  不返回任何元素时，MATLAB 将从串联中删除  $c\{I\}$ 。因此， $c\{I\}$  不影响结果的类。MATLAB 基于其余的数组并根据类的优先级确定结果的类。请参阅“不同类的有效合并”。在生成的代码中，由于代码生成器将  $c\{I\}$  视为  $[c\{I\}]$ ，因此  $[c\{I\}]$  的类将影响整体串联  $[...[c\{I\}]...]$  的结果的类。根据前面所述的规则确定  $[c\{I\}]$  的类。

- 在生成的代码中， $[c\{I\}]$  的大小可能不同于 MATLAB 中的大小。

在 MATLAB 中，串联  $[c\{I\}]$  为 `0x0` 双精度值。在生成的代码中，根据前面所述的规则确定  $[c\{I\}]$  的大小。



# MATLAB 结构体的代码生成

---

- “用于代码生成的结构体定义” (第 7-2 页)
- “代码生成允许的结构体操作” (第 7-3 页)
- “为代码生成定义标量结构体” (第 7-4 页)
- “为代码生成定义结构体数组” (第 7-6 页)

## 用于代码生成的结构体定义

要为结构体生成高效的独立代码，必须按照与通常在 MATLAB 环境中运行代码时不同的方式来定义和使用结构体：

不同之处	更多信息
使用有限的操作集。	“代码生成允许的结构体操作”（第 7-3 页）
遵循对标量结构体的属性和值的限制。	“为代码生成定义标量结构体”（第 7-4 页）
使结构体在数组中统一。	“为代码生成定义结构体数组”（第 7-6 页）
在索引期间逐个引用结构体字段。	“Index Substructures and Fields”
为结构体和字段分配值时避免类型不匹配。	“Assign Values to Structures and Fields”



## 代码生成允许的结构体操作

要为 MATLAB 结构体生成高效的独立代码，您的操作需遵循以下限制：

- 使用圆点表示法索引结构体字段
- 将主要函数输入定义为结构体
- 将结构体传递给局部函数

# 为代码生成定义标量结构体

本节内容
“通过赋值定义标量结构体时的限制” （第 7-4 页）
“在每个控制流路径中以一致的顺序添加字段” （第 7-4 页）
“首次使用后添加新字段的限制” （第 7-4 页）

## 通过赋值定义标量结构体时的限制

当通过将一个变量赋给预先存在的结构体来定义标量结构体时，不需要在赋值之前定义该变量。但是，如果您已定义该变量，它必须具有与您赋给它的结构体相同的类、大小和复/实性。在以下示例中，**p** 定义为与预定义的结构体 **S** 具有相同属性的结构体：

```
...
S = struct('a', 0, 'b', 1, 'c', 2);
p = S;
...
```

## 在每个控制流路径中以一致的顺序添加字段

在创建结构体时，必须在每个控制流路径中以相同的顺序添加字段。例如，以下代码生成编译器错误，因为它在每个 **if** 语句子句中以不同顺序添加结构体 **x** 的字段：

```
function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.b = 30; % Generates an error (on variable x)
    x.a = 40;
end
y = x.a + x.b;
```

在此示例中，在第一个 **if** 语句子句中对 **x.a** 的赋值位于 **x.b** 之前，但在 **else** 子句中赋值的顺序相反。以下是更正后的代码：

```
function y = fcn(u) %#codegen
if u > 0
    x.a = 10;
    x.b = 20;
else
    x.a = 40;
    x.b = 30;
end
y = x.a + x.b;
```

## 首次使用后添加新字段的限制

对结构体执行以下操作后，无法向该结构体中添加字段：

- 从结构体中读取
- 对结构体数组进行索引

- 将结构体传递给函数

以如下代码为例：

```
...  
x.c = 10; % Defines structure and creates field c  
y = x; % Reads from structure  
x.d = 20; % Generates an error  
...
```

在此示例中，尝试在读取结构体 **x** 后添加新字段 **d** 会生成错误。

这种限制适用于整个结构体层次结构。例如，在对结构体的字段或嵌套结构体之一进行操作后，无法将字段添加到该结构体中，如此示例中所示：

```
function y = fcn(u) %#codegen
```

```
x.c = 10;  
y = x.c;  
x.d = 20; % Generates an error
```

在此示例中，在读取结构体 **x** 的字段 **c** 后，尝试向该结构体添加新字段 **d** 会生成错误。

## 为代码生成定义结构体数组

本节内容
“确保字段的一致性” （第 7-6 页）
“使用 repmat 定义具有一致字段属性的结构体数组” （第 7-6 页）
“使用 struct 定义结构体数组” （第 7-6 页）
“使用串联定义结构体数组” （第 7-7 页）

### 确保字段的一致性

对于代码生成，当您创建 MATLAB 结构体数组时，数组元素中的对应字段必须具有相同的大小、类型和复/实性。

创建结构体数组后，可以使用 `coder.varsize` 使结构体字段具有可变大小。请参阅“声明可变大小结构体字段”。

### 使用 repmat 定义具有一致字段属性的结构体数组

您可以使用 MATLAB `repmat` 函数从标量结构体创建结构体数组，该函数将复制并平铺现有标量结构体：

- 1 创建一个标量结构体，如“为代码生成定义标量结构体”（第 7-4 页）中所述。
- 2 调用 `repmat`，传递标量结构体和数组的维度。
- 3 使用标准数组索引和结构体圆点表示法为每个结构体赋值。

例如，以下代码会创建一个 1×3 标量结构体数组 `X`。数组中的每个元素都由结构体 `s` 定义，该结构体有两个字段，`a` 和 `b`：

```
...
s.a = 0;
s.b = 0;
X = repmat(s,1,3);
X(1).a = 1;
X(2).a = 2;
X(3).a = 3;
X(1).b = 4;
X(2).b = 5;
X(3).b = 6;
...
```

### 使用 struct 定义结构体数组

要使用 `struct` 函数创建结构体数组，请将字段值参数指定为元胞数组。每个元胞数组元素都是对应结构体数组元素中字段的值。对于代码生成，结构体中的对应字段必须具有相同的类型。因此，字段值的元胞数组中的元素必须具有相同的类型。

例如，以下代码会创建一个 1×3 结构体数组。对于结构体数组中的每个结构体，`a` 的类型为 `double`，`b` 的类型为 `char`。

```
s = struct('a', {1 2 3}, 'b', {'a' 'b' 'c'});
```

## 使用串联定义结构体数组

要创建小型结构体数组，可以使用方括号（`[]`）作为串联运算符将一个或多个结构体联接到一个数组中。请参阅“创建、串联和扩展矩阵”。对于代码生成，您串联的结构体必须具有相同的大小、类和复/实性。

例如，以下代码使用串联和局部函数来创建  $1 \times 3$  结构体数组的元素：

```
...  
W = [ sab(1,2) sab(2,3) sab(4,5) ];  
  
function s = sab(a,b)  
    s.a = a;  
    s.b = b;  
...
```

## 另请参阅

MATLAB Function

## 相关示例

- “为代码生成定义标量结构体”（第 7-4 页）
- “Define and Use Structure Parameters” (Simulink)
- “在 MATLAB Function 模块中创建结构体” (Simulink)



## 分类数组的代码生成

---





## 元胞数组的代码生成

---

## 代码生成的元胞数组限制

当您在用于代码生成的 MATLAB 代码中使用元胞数组时，必须遵守以下限制：

- “元胞数组元素赋值” (第 9-2 页)
- “可变大小元胞数组” (第 9-3 页)
- “使用 cell 定义可变大小元胞数组” (第 9-3 页)
- “元胞数组索引” (第 9-6 页)
- “使用 {end + 1} 增大元胞数组” (第 9-6 页)
- “元胞数组内容” (第 9-7 页)
- “将元胞数组传递给外部 C/C++ 函数” (第 9-7 页)

### 元胞数组元素赋值

在使用某元胞数组元素之前，必须在所有执行路径上对该元素赋值。例如：

```
function z = foo(n)
%#codegen
c = cell(1,3);
if n < 1
    c{2} = 1;

else
    c{2} = n;
end
z = c{2};
end
```

将元胞数组传递给函数或从函数返回元胞数组时，代码生成器会将其视为使用元胞数组的所有元素。因此，在将元胞数组传递给函数或从函数返回该元胞数组之前，必须为其所有元素赋值。例如，不允许使用以下代码，因为它没有为 c{2} 赋值，而 c 是函数输出。

```
function c = foo()
%#codegen
c = cell(1,3);
c{1} = 1;
c{3} = 3;
end
```

对元素的赋值必须在所有执行路径上保持一致。不允许使用以下代码，因为 y{2} 在一条执行路径上为 double 类型，而在另一条执行路径上为 char 类型。

```
function y = foo(n)
y = cell(1,3)
if n > 1;
    y{1} = 1
    y{2} = 2;
    y{3} = 3;
else
    y{1} = 10;
    y{2} = 'a';
    y{3} = 30;
end
```

## 可变大小元胞数组

- 异构元胞数组不支持 `coder.varsize`。
- 如果使用 `cell` 函数定义固定大小元胞数组，则无法使用 `coder.varsize` 指定元胞数组具有可变大小。例如，以下代码会导致代码生成错误，因为 `x = cell(1,3)` 使 `x` 成为固定大小的  $1 \times 3$  元胞数组。

```
...
x = cell(1,3);
coder.varsize('x',[1 5])
...
```

您可以将 `coder.varsize` 与使用花括号定义的元胞数组结合使用。例如：

```
...
x = {1 2 3};
coder.varsize('x',[1 5])
...
```

- 要使用 `cell` 函数创建可变大小的元胞数组，请使用以下代码模式：

```
function mycell(n)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
end
```

请参阅“使用 `cell` 定义可变大小元胞数组”（第 9-3 页）。

要指定元胞数组的上界，请使用 `coder.varsize`。

```
function mycell(n)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
coder.varsize('x',[1,20]);
end
end
```

## 使用 `cell` 定义可变大小元胞数组

对于代码生成，在使用某元胞数组元素之前，必须为其赋值。当您使用 `cell` 创建可变大小元胞数组（例如 `cell(1,n)`）时，MATLAB 会用一个空矩阵对每个元素赋值。然而，代码生成并不会对元素进行赋值。对于代码生成，在您使用 `cell` 创建可变大小元胞数组后，必须对该元胞数组的所有元素赋值，然后才能使用该元胞数组。例如：

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

在首次使用元胞数组之前，代码生成器会分析您的代码，以确定是否所有元素均已赋值。如果代码生成器检测到某些元素未赋值，代码生成将失败，并显示错误消息。例如，将 `for` 循环的上界修改为 `j`。

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:j %<- Modified here
    x{i} = i;
end
z = x{j};
end
```

如果进行了此修改且输入 `j` 小于 `n`，则该函数不会为所有元胞数组元素赋值。代码生成会产生错误：

Unable to determine that every element of 'x{:}' is assigned before this line.

有时，尽管您的代码对元胞数组的所有元素进行了赋值，但代码生成器仍报告此消息，这是因为分析未检测到所有元素均已赋值。请参阅“无法确定元胞数组的每个元素都已赋值”（第 37-2 页）。

为避免此错误，请遵循以下准则：

- 当您使用 `cell` 定义可变大元胞数组时，请按照以下模式编写代码：

```
function z = mycell(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

以下是针对多维元胞数组的模式：

```
function z = mycell(m,n,p)
%#codegen
x = cell(m,n,p);
for i = 1:m
    for j = 1:n
        for k = 1:p
            x{i,j,k} = i+j+k;
        end
    end
end
z = x{m,n,p};
end
```

- 循环计数器的递增或递减量为 1。
- 在一个循环或一组嵌套循环内定义元胞数组。例如，不允许使用以下代码：

```
function z = mycell(n, j)
x = cell(1,n);
for i = 1:5
    x{i} = 5;
end
for i = 6:n
    x{i} = 5;
end
```

```
end
z = x{};
end
```

- 对元胞维度、循环初始值和结束值使用相同的变量。例如，以下代码的代码生成就会失败，因为元胞创建使用的是 `n`，而循环结束值使用的是 `m`：

```
function z = mycell(n, j)
x = cell(1,n);
m = n;
for i = 1:m
    x{i} = 2;
end
z = x{};
end
```

重写代码以对元胞创建和循环结束值使用 `n`：

```
function z = mycell(n, j)
x = cell(1,n);
for i = 1:n
    x{i} = 2;
end
z = x{};
end
```

- 按照以下模式创建元胞数组：

```
x = cell(1,n)
```

通过使用所需的元胞初始化临时变量，将元胞数组赋给结构体的字段或对象的属性。例如：

```
t = cell(1,n)
for i = 1:n
    t{i} = i+1;
end
myObj.prop = t;
```

不要将元胞数组直接赋给结构体的字段或对象的属性。例如，不允许使用以下代码：

```
myObj.prop = cell(1,n);
for i = 1:n
    myObj.prop{i} = i+1;
end
```

不要在元胞数组构造函数 `{}` 内使用 `cell` 函数。例如，不允许使用以下代码：

```
x = {cell(1,n)};
```

- 为元胞数组元素赋值的元胞数组创建语句和循环赋值语句必须位于同一个唯一执行路径中。例如，不允许使用以下代码。

```
function z = mycell(n)
if n > 3
    c = cell(1,n);
else
    c = cell(n,1);
end
for i = 1:n
    c{i} = i;
```

```
end
z = c{n};
end
```

要修复此代码，请将赋值循环移入创建元胞数组的代码块内。

```
function z = cellerr(n)
if n > 3
    c = cell(1,n);
    for i = 1:n
        c{i} = i;
    end
else
    c = cell(n,1);
    for i = 1:n
        c{i} = i;
    end
end
z = c{n};
end
```

## 元胞数组索引

- 您不能使用圆括号 () 对元胞数组进行索引。请考虑使用花括号 {} 对元胞数组进行索引来访问元胞的内容。
- 您必须使用常量索引或使用具有常量边界的 for 循环对异构元胞数组进行索引。

例如，不允许使用以下代码。

```
x = {1, 'mytext'};
disp(x{randi});
```

您可以在具有常量边界的 for 循环中对异构元胞数组进行索引，因为代码生成器会展开循环。展开会为每次循环迭代创建循环体的一个单独副本，这使得每次循环迭代中的索引不变。然而，如果 for 循环体很大或者有很多迭代，则展开可能增加编译时间和生成无效代码。

如果 A 和 B 是常数，则下列代码展示了如何在具有常量边界的 for 循环中对异构元胞数组进行索引。

```
x = {1, 'mytext'};
for i = A:B
    disp(x{i});
end
```

## 使用 {end + 1} 增大元胞数组

要增大元胞数组 X，可以使用 X{end + 1}。例如：

```
...
X = {1 2};
X{end + 1} = 'a';
...
```

当您使用 {end + 1} 增大元胞数组时，请遵循以下限制：

- 仅使用 {end + 1}。不要使用 {end + 2}、{end + 3} 等。

- 仅对向量使用 `{end + 1}`。例如，不允许使用以下代码，因为 `X` 是矩阵，而不是向量：

```
...
X = {1 2; 3 4};
X{end + 1} = 5;
```

- 仅对变量使用 `{end + 1}`。在以下代码中，`{end + 1}` 不会使 `{1 2 3}` 增大。在本例中，代码生成器将 `{end + 1}` 视为超出 `X{2}` 范围的索引。

```
...
X = {'a' { 1 2 3 }};
X{2}{end + 1} = 4;
```

- 当 `{end + 1}` 在循环中增大元胞数组时，该元胞数组必须是可变大小的。因此，元胞数组必须是同构元胞数组。

允许使用以下代码，因为 `X` 是同构元胞数组。

```
...
X = {1 2};
for i=1:n
    X{end + 1} = 3;
end
...
```

不允许使用以下代码，因为 `X` 是异构元胞数组。

```
...
X = {1 'a' 2 'b'};
for i=1:n
    X{end + 1} = 3;
end
...
```

## 元胞数组内容

元胞数组不能包含 `mxarrays`。在元胞数组中，不能存储外部函数返回的值。

## 将元胞数组传递给外部 C/C++ 函数

您不能将元胞数组传递给 `coder.ceval`。如果某变量是 `coder.ceval` 的输入参数，则应将该变量定义为数组或结构体，而不能是元胞数组。

## 另请参阅

## 详细信息

- “Code Generation for Cell Arrays”
- “生成的代码和 MATLAB 代码之间的差异”（第 2-6 页）





## 日期时间数组的代码生成

---

## 代码生成的日期时间数组限制

当您在打算用于代码生成的 MATLAB 代码中创建 `datetime` 数组时，必须使用 `datetime` 函数指定值。请参阅“日期时间”。

对于 `datetime` 数组，代码生成不支持以下输入和操作：

- 文本输入。例如，将字符向量指定为输入参数会生成错误。

```
function d = foo() %#codegen
    d = datetime('2019-12-01');
end
```

- 'Format' 名称-值对组参数。无法通过使用 `datetime` 函数或通过设置 `datetime` 数组的 `Format` 属性来指定显示格式。要使用某特定显示格式，请在 MATLAB 中创建一个 `datetime` 数组，然后将其作为输入参数传递给用于代码生成的函数。
- 'TimeZone' 名称-值对组参数和 `TimeZone` 属性。当您在用于代码生成的代码中使用 `datetime` 数组时，这些数组必须未设置时区。
- 设置时间分量属性。例如，在以下代码中设置 `Hour` 属性会生成错误：

```
d = datetime;
d.Hour = 2;
```

- 通过赋值实现增长。例如，在数组末尾以外赋值会生成错误。

```
function d = foo() %#codegen
    d = datetime(2019,1:12,1,12,0,0);
    d(13) = datetime(2020,1,1,12,0,0);
end
```

- 删除元素。例如，将空数组赋给元素会生成错误。

```
function d = foo() %#codegen
    d = datetime(2019,1:12,1,12,0,0);
    d(1) = [];
end
```

- 使用 `char`、`cellstr` 或 `string` 函数将 `datetime` 值转换为文本。

适用于类的限制也适用于 `datetime` 数组。有关详细信息，请参阅“用于代码生成的 MATLAB 类定义”（第 15-2 页）。

### 另请参阅

`datetime` | `NaT`

### 详细信息

- “Code Generation for Datetime Arrays”
- “Define Datetime Array Inputs”

# 持续时间数组的代码生成

---

- “持续时间数组的代码生成” (第 11-2 页)
- “代码生成的持续时间数组限制” (第 11-6 页)

持续时间数组的代码生成

本节内容
“为代码生成定义持续时间数组” （第 11-2 页）
“允许对持续时间数组执行的操作” （第 11-2 页）
“支持持续时间数组的 MATLAB 工具箱函数” （第 11-3 页）

持续时间数组中的值以固定长度的单位（如小时、分钟和秒）来表示经过的时间。您可以创建以固定长度（24 小时）的天数和固定长度（365.2425 天）的年数为单位的已用时间。

您可以对持续时间数组执行加法、减法、排序、比较、串联和绘图等操作。

在代码生成中使用持续时间数组时，请遵守这些限制。

为代码生成定义持续时间数组

对于代码生成，请使用 `duration` 函数创建持续时间数组。例如，假设您的 MATLAB 函数的输入参数是三个任意大小的数值数组，其元素将时间长度指定为小时、分钟和秒。您可以从这三个输入数组创建一个持续时间数组。

```
function d = foo(h,m,s) %#codegen
    d = duration(h,m,s);
end
```

您可以使用 `years`、`days`、`hours`、`minutes`、`seconds` 和 `milliseconds` 函数以年、日、小时、分钟或秒为单位创建持续时间数组。例如，您可以从输入数值数组创建一个由小时数组成的数组。

```
function d = foo(h) %#codegen
    d = hours(h);
end
```

允许对持续时间数组执行的操作

对于代码生成，您只能对持续时间数组执行下表中列出的操作。

运算	示例	附注
赋值运算符：=	<code>d = duration(1:3,0,0);</code> <code>d(1) = hours(5);</code>  <code>d = duration(1:3,0,0);</code> <code>d(1) = hours(5);</code>	代码生成不支持使用赋值运算符 = 进行以下操作： <ul style="list-style-type: none"><li>删除元素。</li><li>扩展持续时间数组的大小。</li></ul>
关系运算符：< > <= >= == ~=	<code>d = duration(1:3,0,0);</code> <code>tf = d(1) &lt; d(2);</code>  <code>d = duration(1:3,0,0);</code> <code>tf = d(1) &lt; d(2);</code>	代码生成支持关系运算符。

运算	示例	附注
索引操作	<pre>d = duration(1:3,0,0); idx = [1 2]; d(idx); idx = logical([1 1 0]); d(idx);  d = duration(1:3,0,0); idx = [1 2]; d(idx); idx = logical([1 1 0]); d(idx);</pre>	代码生成支持按位置索引、线性索引和逻辑索引。
串联	<pre>d1 = duration(1:3,0,0); d2 = duration(4,30,0); d = [d1 d2];  d1 = duration(1:3,0,0); d2 = duration(4,30,0); d = [d1 d2];</pre>	代码生成支持持续时间数组的串联。

支持持续时间数组的 MATLAB 工具箱函数

对于代码生成，您可以将持续时间数组与以下 MATLAB 工具箱函数结合使用：

- `abs`
- `cat`
- `ceil`
- `colon`
- `cummax`
- `cummin`
- `cumsum`
- `ctranspose`
- `datevec`
- `days`
- `diff`
- `duration`
- `eps`
- `eq`
- `floor`
- `ge`
- `gt`
- `hms`
- `horzcat`
- `hours`
- `interp1`
- `intersect`
- `iscolumn`

- isempty
- isequal
- isequaln
- isfinite
- isinf
- ismatrix
- ismember
- isnan
- isreal
- isrow
- isscalar
- issorted
- issortedrows
- isvector
- ldivide
- le
- length
- linspace
- lt
- max
- mean
- median
- milliseconds
- min
- minus
- minutes
- mldivide
- mode
- mrdivide
- mod
- mtimes
- ndims
- ne
- nnz
- numel
- permute
- plus
- repmat
- rdivide

- `rem`
- `reshape`
- `seconds`
- `setdiff`
- `setxor`
- `sign`
- `size`
- `sort`
- `sortrows`
- `std`
- `sum`
- `times`
- `transpose`
- `uminus`
- `union`
- `unique`
- `uplus`
- `vertcat`
- `years`

## 另请参阅

## 详细信息

- “Define Duration Array Inputs”
- “代码生成的持续时间数组限制” (第 11-6 页)

## 代码生成的持续时间数组限制

当您在打算用于代码生成的 MATLAB 代码中创建持续时间数组时，必须使用 `duration`、`years`、`days`、`hours`、`minutes`、`seconds` 或 `milliseconds` 函数指定持续时间。请参阅“日期时间”。

对于持续时间数组，代码生成不支持以下输入和操作：

- 文本输入。例如，将字符向量指定为输入参数会生成错误。

```
function d = foo() %#codegen
    d = duration('01:30:00');
end
```

- 通过赋值实现增长。例如，在数组末尾以外赋值会生成错误。

```
function d = foo() %#codegen
    d = duration(1:3,0,0);
    d(4) = hours(4);
end
```

- 删除元素。例如，将空数组赋给元素会生成错误。

```
function d = foo() %#codegen
    d = duration(1:3,0,0);
    d(1) = [];
end
```

- 使用 `char`、`cellstr` 或 `string` 函数将持续时间值转换为文本。

适用于类的限制也适用于持续时间数组。有关详细信息，请参阅“用于代码生成的 MATLAB 类定义”（第 15-2 页）。

### 另请参阅

`duration` | `years` | `days` | `hours` | `minutes` | `seconds` | `milliseconds`

### 详细信息

- “持续时间数组的代码生成”（第 11-2 页）
- “Define Duration Array Inputs”



## 表的代码生成

---

表的代码生成

本节内容
“为代码生成定义表” （第 12-2 页）
“允许对表执行的操作” （第 12-2 页）
“支持表的 MATLAB 工具箱函数” （第 12-3 页）

**table** 数据类型是一种适用于列向数据或表数据的数据类型，这些数据通常以列的形式存储在文本文件或电子表格中。表由若干行向变量和若干列向变量组成。表中的每个变量可以具有不同的数据类型和大小，但有一个限制条件是每个变量的行数必须相同。有关详细信息，请参阅“表”。

在代码生成中使用表时，请遵守这些限制。

为代码生成定义表

对于代码生成，请使用 **table** 函数。例如，假设您的 MATLAB 函数的输入参数是三个具有相同行数的数组和一个具有变量名称的元胞数组。您可以创建一个包含这些数组作为表变量的表。

```
function T = foo(A,B,C,vnames) %#codegen
    T = table(A,B,C,'VariableNames',vnames);
end
```

您可以使用 **array2table**、**cell2table** 和 **struct2table** 函数将数组、元胞数组和结构体转换为表。例如，您可以将输入元胞数组转换为表。

```
function T = foo(C,vnames) %#codegen
    T = cell2table(C,'VariableNames',vnames);
end
```

对于代码生成，您必须在创建表时提供表变量名称。表变量名称不必是有效的 MATLAB 标识符。名称必须由 ASCII 字符组成，但可以包括任何 ASCII 字符（如逗号、破折号和空白字符）。

允许对表执行的操作

对于代码生成，您只能对表执行下面列出的操作。

运算	示例	附注
赋值运算符：=	T = table(A,B,C,'VariableNames',vnames); T{:,1} = D;	代码生成不支持使用赋值运算符 = 进行以下操作： <ul style="list-style-type: none"><li>删除变量或行。</li><li>添加变量或行。</li></ul>
索引操作	T = table(A,B,C,'VariableNames',vnames); T(1:5,1:3);	代码生成支持按位置、变量或行名称进行索引，以及逻辑索引。  代码生成支持： <ul style="list-style-type: none"><li>带圆括号 () 的表索引。</li><li>带花括号 {} 的内容索引。</li><li>访问表变量的圆点表示法。</li></ul>

运算	示例	附注
串联	<pre>T1 = table(A,B,C,'VariableNames',vnames); T2 = table(D,E,F,'VariableNames',vnames); T = [T1 ; T2];</pre>	<p>代码生成支持表串联。</p> <ul style="list-style-type: none"><li>• 对于垂直串联，表中的变量必须具有相同的名称，其顺序也需相同。</li><li>• 对于水平串联，表必须具有相同的行数。如果表具有行名称，则它们必须具有相同的行名称，其顺序也需相同。</li></ul>

支持表的 MATLAB 工具箱函数

对于代码生成，您可以将表与以下 MATLAB 工具箱函数结合使用：

- addvars
- array2table
- cat
- cell2table
- convertvars
- height
- horzcat
- innerjoin
- intersect
- isempty
- ismember
- issortedrows
- join
- mergevars
- movevars
- ndims
- numel
- outerjoin
- removevars
- renamevars
- rows2vars
- setdiff
- setxor
- size
- sortrows
- splitvars
- stack

- `struct2table`
- `table`
- `table2array`
- `table2cell`
- `table2struct`
- `union`
- `unique`
- `unstack`
- `varfun`
- `vertcat`
- `width`

### 另请参阅

### 详细信息

- “Define Table Inputs”
- “Table Limitations for Code Generation”

## 时间表的代码生成

---



## 枚举数据的代码生成

---

## 在生成的代码中自定义枚举类型

对于代码生成，要自定义枚举，请在类定义的静态方法部分中包含下表中列出的方法的自定义版本。

方法	描述	返回或指定的默认值	何时使用
<code>getDefaultValue</code>	返回默认枚举值。	枚举类定义中的第一个值。	对于不同于第一个枚举值的默认值，请提供返回所需默认值的 <code>getDefaultValue</code> 方法。请参阅“指定默认枚举值”（第 14-2 页）。
<code>getHeaderFile</code>	指定定义外部定义的枚举类型的文件。	"	要使用外部定义的枚举类型，请提供 <code>getHeaderFile</code> 方法，该方法返回定义该类型的头文件的路径。在这种情况下，代码生成器不会生成类定义。请参阅“指定头文件”（第 14-3 页）。
<code>addClassNameToEnumNames</code>	指定类名在生成的代码中是否变成前缀。	<code>false</code> - 不使用前缀。	如果希望类名称成为生成代码中的前缀，请将 <code>addClassNameToEnumNames</code> 方法的返回值设置为 <code>true</code> 。请参阅“在生成的枚举类型值名称中包含类名称前缀”（第 14-3 页）。  <b>注意</b> 在生成 C++11 枚举类时，代码生成器会忽略此静态方法。
<code>generateEnumClass</code>	指定是否生成 C++11 枚举类	<code>true</code> - 枚举类以 C++11 代码生成	在生成 C++11 代码时，要指示代码生成器为特定的 MATLAB 枚举生成普通 C 枚举，请将 <code>generateEnumClass</code> 方法的返回值设置为 <code>false</code> 。请参阅“生成包含普通 C 枚举的 C++11 代码”（第 14-4 页）。

### 指定默认枚举值

如果转换为枚举类型的变量的值与枚举类型值之一不匹配，则：

- 生成的 MEX 会报告错误。



- 生成的 C/C++ 代码会将变量的值替换为枚举类型的默认值。

除非您另有指定，否则枚举类型的默认值是枚举类定义中的第一个值。要指定不同的默认值，请在方法部分添加您自己的 `getDefaultValue` 方法。在此示例中，第一个枚举成员值是 `LEDcolor.GREEN`，但 `getDefaultValue` 方法返回 `LEDcolor.RED`：

```
classdef LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end

    methods (Static)
        function y = getDefaultValue()
            y = LEDcolor.RED;
        end
    end
end
```

## 指定头文件

要指定在外部文件中定义枚举类型，请提供自定义的 `getHeaderFile` 方法。此示例指定在外部文件 `my_LEDcolor.h` 中定义 `LEDcolor`。

```
classdef LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end

    methods(Static)
        function y=getHeaderFile()
            y='my_LEDcolor.h';
        end
    end
end
```

您必须提供 `my_LEDcolor.h`。例如：

```
enum LEDcolor
{
    GREEN = 1,
    RED
};
typedef enum LEDcolor LEDcolor;
```

如果将 MATLAB 枚举 `LEDcolor` 放在包 `pkg` 中并生成 C++ 代码，代码生成会保留此枚举的名称，并将其放在生成代码中的命名空间 `pkg` 内。因此，在您提供的头文件中，您必须在命名空间 `pkg` 内定义此枚举。

## 在生成的枚举类型值名称中包含类名称前缀

默认情况下，生成的枚举类型值名称不包括类名称前缀。例如：

```
enum LEDcolor
{
```

```

    GREEN = 1,
    RED
};

```

```
typedef enum LEDcolor LEDcolor;
```

要包含类名称前缀，请提供返回 `true` 的 `addClassNameToEnumNames` 方法。例如：

```

classdef LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2)
    end

    methods(Static)
        function y = addClassNameToEnumNames()
            y=true;
        end
    end
end

```

在生成的类型定义中，枚举值名称包括类前缀 `LEDcolor`。

```

enum LEDcolor
{
    LEDcolor_GREEN = 1,
    LEDcolor_RED
};

```

```
typedef enum LEDcolor LEDcolor;
```

## 生成包含普通 C 枚举的 C++11 代码

当您生成 C++11 代码时，您的 MATLAB 枚举类转换为 C++11 枚举类。例如：

```

enum class MyEnumClass16 : short
{
    Orange = 0, // Default value
    Yellow,
    Pink
};

```

要生成普通 C 枚举，请提供返回 `false` 的 `generateEnumClass` 方法。例如：

```

classdef MyEnumClass16 < int16
    enumeration
        Orange(0),
        Yellow(1),
        Pink(2)
    end

    % particular enum opting out
    methods(Static)
        function y = generateEnumClass()
            y = false;
        end
    end
end

```

现在生成的 C++11 代码包含普通 C 枚举。

```
enum MyEnumClass16 : short
{
    Orange = 0, // Default value
    Yellow,
    Pink
};
```

## 另请参阅

## 详细信息

- 修改超类方法和属性
- “Code Generation for Enumerations”



# MATLAB 类的代码生成

---

- “用于代码生成的 MATLAB 类定义” （第 15-2 页）
- “为 MATLAB 句柄类和 System object 生成代码” （第 15-6 页）
- “MATLAB 代码生成中的 System object” （第 15-9 页）

## 用于代码生成的 MATLAB 类定义

要为 MATLAB 类生成高效的独立代码，必须按照与在 MATLAB 环境中运行代码时不同的方式来使用类。

### 语言限制

虽然为类的常用功能（例如属性和方法）提供了代码生成支持，但仍有一些高级功能不受支持，例如：

- 事件
- 侦听程序
- 对象数组
- 递归数据结构体
  - 链表
  - 树
  - 图
- 构造函数中的嵌套函数
- 可重载运算符 `subsref`、`subsassign` 和 `subsindex`

在 MATLAB 中，类可以定义自己的 `subsref`、`subsassign` 和 `subsindex` 方法版本。代码生成不支持拥有这些自定义方法的类。

- `empty` 方法

在 MATLAB 中，类具有内置的静态方法 `empty`，此方法将创建该类的空数组。代码生成不支持此方法。

- 以下 MATLAB 句柄类方法：

- `addlistener`
- `eq`
- `findobj`
- `findprop`

- `AbortSet` 属性特性

### 与类不兼容的代码生成功能

- 可以为使用类的入口点 MATLAB 函数生成代码，但不能直接为 MATLAB 类生成代码。

例如，如果 `ClassNameA` 是一个类定义，则不能通过执行以下命令来生成代码：

```
codegen ClassNameA
```

- 句柄类对象不能作为入口函数的输入或输出。
- 值类对象可以作为入口函数的输入或输出。但是，如果值类对象包含句柄类对象，则值类对象不能作为入口函数的输入或输出。句柄类对象不能作为入口函数的输入或输出。
- 代码生成不支持作为句柄类的全局变量。
- 代码生成不支持构造函数的多个输出。
- 代码生成不支持将值类对象指定给不可调属性。例如，如果 `prop` 是不可调属性，而 `v` 是基于值类的对象，则 `obj.prop=v`；无效。

- 不能使用 `coder.extrinsic` 将类或方法声明为外部类或方法。
- 不能将 MATLAB 类传递给 `coder.ceval` 函数。可以将类属性传递给 `coder.ceval`。
- 如果某属性具有 `get` 方法、`set` 方法或验证程序，或者是具有某些特性的 System object 属性，则您不能按引用将该属性传递给外部函数。请参阅“Passing By Reference Not Supported for Some Properties”。
- 如果对象具有重复的属性名称，并且代码生成器尝试对该对象进行常量折叠，则代码生成可能会失败。当某对象与 `coder.Constant` 或 `coder.const` 结合使用时，或者当该对象作为常量折叠的外部函数的输入或输出时，代码生成器会对该对象进行常量折叠。

在下列情况下，子类的对象中会出现重复的属性名称：

- 子类具有与超类的某个属性名称相同的属性。
- 子类派生于某一属性使用相同名称的多个超类。

在多个具有继承关系的类中，重复的属性名称必须为一致的常量或非常量。例如，如果具有常量属性 `aProp` 的对象从 `aProp` 定义为非常量的超类继承 `aProp`，则代码生成会生成错误。

有关 MATLAB 何时允许重复的属性名称的信息，请参阅“Subclassing Multiple Classes”。

## 为代码生成定义类属性

要进行代码生成，必须按照与在 MATLAB 环境中运行代码时不同的方式来定义类属性：

- MEX 函数报告属性验证产生的错误。独立的 C/C++ 代码仅在启用运行时错误报告时才报告这些错误。请参阅“Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”。在生成独立的 C/C++ 代码之前，最好通过在整个输入值范围内运行 MEX 函数来测试属性验证。
- 定义属性之后，不要为其指定不兼容的类型。在使用属性之前，需对其进行必要的配置。

定义类属性以进行代码生成时，需要考虑的因素与定义变量时相同。在 MATLAB 语言中，变量可在运行时动态更改其类、大小或复/实性，所以您可以使用同一个变量存储不同类、大小或复/实性的值。C 和 C++ 使用静态定型。在使用变量之前，为了确定变量的类型，代码生成器需要为每个变量完整赋值。同样，在使用属性之前，您必须显式定义属性的类、大小和复/实性。

- 初始值：
  - 如果属性没有显式初始值，代码生成器将假定它未在构造函数的起始部分进行定义。代码生成器不会指定空矩阵作为默认值。
  - 如果属性没有初始值，而且代码生成器无法确定该属性在第一次使用之前是否已赋值，软件将会生成编译错误。
  - 对于 System object，如果不可调属性是一个结构体，则必须为该结构体完整赋值。不能使用下标进行部分赋值。

例如，对于不可调属性，可以使用以下完整赋值语句：

```
mySystemObject.nonTunableProperty=struct('fieldA','a','fieldB','b');
```

但不能使用以下部分赋值语句：

```
mySystemObject.nonTunableProperty.fieldA = 'a';
mySystemObject.nonTunableProperty.fieldB = 'b';
```

- 类属性不支持 `coder.varsize`。

- 如果一个属性的初始值是对象，则该属性必须是常量。要使属性为常量，可在属性模块中声明 **Constant** 属性。例如：

```
classdef MyClass
    properties (Constant)
        p1 = MyClass2;
    end
end
```

代码生成不支持分配给包含 System object 的对象的常量属性。

- MATLAB 会在代码生成之前加载类时计算类的初始值。如果您在 MATLAB 类属性初始化中使用持久变量，则加载类时计算的持久变量的值属于 MATLAB；它不是在代码生成时使用的值。如果您在 MATLAB 类属性初始化中使用 `coder.target`，`coder.target('MATLAB')` 将返回 **true (1)**。
- 可变大小属性：
  - 代码生成支持值类和句柄类的有上界和无界的可变大小属性。
  - 要生成无界的可变大小类属性，请启用动态内存分配。
  - 要创建一个可变大小的类属性，需要对类属性先后进行两次赋值，先为标量值，然后是数组。

```
classdef varSizeProp1 < handle
    properties
        prop
        varProp
    end
end

function extFunc(n)
    obj = varSizeProp1;
    % Assign a scalar value to the property.
    obj.prop = 1;
    obj.varProp = 1;
    % Assign an array to the same property to make it variable-sized.
    obj.prop = 1:98;
    obj.varProp = 1:n;
end
```

在上述代码中，对 **prop** 和 **varProp** 的第一个赋值是标量，第二个赋值是具有相同基类型的数组。**prop** 的大小有上界 98，使其成为具有上界的可变大小属性。

如果 **n** 在编译时未知，则 **obj.varProp** 是无界的可变大小属性。如果它是已知的，它是具有上界的可变大小类属性。

- 如果类属性是用可变大小数组初始化的，则属性是可变大小的。

```
classdef varSizeProp2
    properties
        prop
    end
    methods
        function obj = varSizeProp2(inVar)
            % Assign incoming value to local variable
            locVar = inVar;

            % Declare the local variable to be a variable-sized column
            % vector with no size limit
            coder.varsize('locVar',[inf 1],[1 0]);

            % Assign value
            obj.prop = locVar;
        end
    end
end
```

在上述代码中，**inVar** 传递给类构造函数，并存储在 **locVar** 中。**locVar** 由 `coder.varsize` 修改为可变大小，并分配给类属性 **obj.prop**，这使得属性的大小可变。



- 如果函数调用 `varSizeProp2` 的输入是可变大小的，则不需要 `coder.varsize`。

```
function z = constructCall(n)
    z = varSizeProp2(1:n);
end
```

- 如果 `n` 的值在编译时未知并且没有指定的界限，则 `z.prop` 是无界的可变大小类属性。
- 如果 `n` 的值在编译时未知，并且具有指定的界限，则 `z.prop` 是具有上界的可变大小类属性。
- 如果属性是常量，其值是对象，则不能更改该对象的属性值。例如，假设：
  - `obj` 是 `myClass1` 的对象。
  - `myClass1` 有一个常量属性 `p1`，它是 `myClass2` 的对象。
  - `myClass2` 有一个 `p2` 属性。

代码生成不支持以下代码：

```
obj.p1.p2 = 1;
```

## 不支持从内置 MATLAB 类继承

不能为从内置 MATLAB 类继承的类生成代码。例如，不能为以下类生成代码：

```
classdef myclass < double
```

此规则的一个例外是 MATLAB 枚举类。您可以为从内置 MATLAB 类继承的枚举类生成代码。请参阅“Code Generation for Enumerations”。

## 另请参阅

`coder.target`

## 相关示例

- “Generate Standalone C/C++ Code That Detects and Reports Run-Time Errors”
- “Classes That Support Code Generation” (Simulink)

## 为 MATLAB 句柄类和 System object 生成代码

此示例说明如何为用户定义的 System object 生成代码，然后在代码生成报告中查看生成的代码。

- 1 在一个可写文件夹中，创建一个 System object `AddOne`，它是 `matlab.System` 的子类。将代码另存为 `AddOne.m`。

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

methods (Access=protected)
% stepImpl method is called by the step method
function y = stepImpl(~,x)
    y = x+1;
end
end
end
```

- 2 编写一个使用此 System object 的函数。

```
function y = testAddOne(x)
%#codegen
    p = AddOne();
    y = p.step(x);
end
```

- 3 为此代码生成一个 MEX 函数。

```
codegen -report testAddOne -args {0}
```

`-report` 选项指示 `codegen` 生成一份代码生成报告，即使没有出现错误或警告也是如此。`-args` 选项指定 `testAddOne` 函数接受一个标量双精度输入。

- 4 点击[查看报告](#)链接。
- 5 在 **MATLAB 源代码**窗格中，点击 `testAddOne`。要查看有关 `testAddOne` 中变量的信息，请点击 **变量**选项卡。

The screenshot shows the MATLAB IDE with the `testAddOne.m` file open. The code is as follows:

```
1 function y = test
2 %codegen
3 p = AddOne();
4 y = p.step(x);
5 end
```

A tooltip for the variable `p` is displayed, showing its properties:

- Size: 1 × 1
- Class: AddOne

The left sidebar shows the **MATLAB SOURCE** pane with the **Function List** and **Call Tree** tabs. The **Function List** shows the following functions:

- testAddOne.m
- fx testAddOne
- AddOne.m
- fx AddOne
- fx stepImpl

The **GENERATED CODE** pane shows the following files:

- Source Files
  - c\_mexapi\_version.c
  - rt\_nonfinite.h
  - rtwtypes.h
  - testAddOne.c
  - testAddOne.h
  - testAddOne\_data.c
  - testAddOne\_data.h
  - testAddOne\_initialize.c
  - testAddOne\_initialize.h
  - testAddOne\_terminate.c
  - testAddOne\_terminate.h
  - testAddOne\_types.h
- Interface Files
  - \_coder\_testAddOne\_api.c
  - \_coder\_testAddOne\_api.h
  - \_coder\_testAddOne\_info.c
  - \_coder\_testAddOne\_info.h
  - \_coder\_testAddOne\_mex.c
  - \_coder\_testAddOne\_mex.h

The bottom pane shows the **SUMMARY** table:

Name	Type	Size	Class
y	Output	1 × 1	double
x	Input	1 × 1	double
p	Local	1 × 1	AddOne

6 要查看 `addOne` 的类定义，请在 **MATLAB 源代码**窗格中，点击 `AddOne`。

The screenshot shows the MATLAB IDE with the `AddOne.m` file open. The code is as follows:

```
1 classdef AddOne < matlab.System
2 % ADDONE Compute an output value that increments the input by one
3
4 methods (Access=protected)
5 % stepImpl method is called by the step method
6 function y = stepImpl(~,x)
7     y = x+1;
8 end
9 end
10 end
```

The left sidebar shows the **MATLAB SOURCE** pane with the **Function List** and **Call Tree** tabs. The **Function List** shows the following functions:

- testAddOne.m
- fx testAddOne
- AddOne.m
- fx AddOne
- fx stepImpl

### 另请参阅

### 详细信息

- “Code Generation for Handle Class Destructors”

# MATLAB 代码生成中的 System object

本节内容
“代码生成中对 System object 的使用规则和限制” (第 15-9 页)
“通过 codegen 使用 System object” (第 15-11 页)
“通过 MATLAB Function 模块使用 System object” (第 15-11 页)
“通过 MATLAB System 模块使用 System object” (第 15-11 页)
“System object 和 MATLAB Compiler 软件” (第 15-11 页)

您可以在 MATLAB 中通过使用 MATLAB Coder 从包含 System object 的系统生成 C/C++ 代码。您可以生成高效紧凑的代码以部署在桌面和嵌入式系统中，并提高定点算法的执行速度。

## 代码生成中对 System object 的使用规则和限制

在 MATLAB 生成的代码中使用 System object 时需遵循以下使用规则和限制。

### 对象构造和初始化

- 如果对象存储在持久变量中，则只需将对象句柄嵌入调用 `isempty()` 函数的 if 语句中，执行一次 System object 初始化即可。
- 将 System object 构造函数的参数设置为编译时常量。
- 初始化 `releaseImpl` 在 `setupImpl` 结束之前使用的所有 System object 属性。
- 在代码生成中，您不能使用其他 MATLAB 类对象将 System object 属性初始化为其默认值。您必须在构造函数中初始化这些属性。

### 输入和输出

- System object 最多可以接受 1024 个输入。每个输入支持最多八个维度。
- 输入的数据类型不应更改。
- 输入的复/实性不应更改。
- 如果希望更改输入的大小，则需要确认已启用对可变大小数据的支持。要支持可变大小数据的代码生成，同样需要启用对可变大小数据的支持。默认情况下，在 MATLAB 中启用了可变大小数据的支持。
- 如果可变大小数据超出了 `DynamicMemoryAllocationThreshold` 的值，则软件中预定义的 System object 将不支持可变大小。
- 不要将 System object 设置为 MATLAB Function 模块的输出。
- 对于 MATLAB Function 模块中的任何 System object，不要使用“将仿真作为工作点保存和恢复”选项。
- 不要将 System object 作为示例输入参数传递给使用 `codegen` 编译的函数。
- 不要将 System object 传递给使用 `coder.extrinsic` 函数声明为外部函数（在解释模式下调用的函数）的函数。从外部函数返回的 System object 和自动变为外部对象的示波器 System object 可以用作另一个外部函数的输入。但是，这些函数不会生成代码。

### 属性

- 在 MATLAB System 模块中，不能对 System object 的离散状态属性使用可变大小。私有属性可以是可变大小的。

- 对象不能用作属性的默认值。
- 对不可调属性只能进行一次赋值，包括构造函数中的赋值。
- 不可调属性值必须是常量。
- 对于定点输入，如果可调属性具有从属数据类型属性，则只能在构造时或锁定对象后设置可调属性。
- 对于 `getNumInputsImpl` 和 `getNumOutputsImpl` 方法，如果您基于某个对象属性设置返回参数，则该对象属性必须具有 `Nontunable` 特性。

### 全局变量

- 允许在 System object 中使用全局变量，除非您正在 Simulink 中通过 MATLAB System 模块使用该 System object。请参阅“Generate Code for Global Data”。

### 方法

- 仅支持对以下 System object 方法进行代码生成：
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone`（仅适用于数据源）
  - `isLocked`
  - `release`
  - `reset`
  - `set`（适用于可调属性）
  - `step`
- 对于您定义的 System object，仅支持对以下方法进行代码生成：
  - `getDiscreteStateImpl`
  - `getNumInputsImpl`
  - `getNumOutputsImpl`
  - `infoImpl`
  - `isDoneImpl`
  - `isInputDirectFeedthroughImpl`
  - `outputImpl`
  - `processTunedPropertiesImpl`
  - `releaseImpl` - 对于此方法，不会自动生成代码。要释放对象，您必须在代码中显式调用 `release` 方法。
  - `resetImpl`
  - `setupImpl`
  - `stepImpl`
  - `updateImpl`
  - `validateInputsImpl`
  - `validatePropertiesImpl`

## 通过 codegen 使用 System object

您可以在 MATLAB 代码中包含 System object，就像包含任何其他元素一样。然后，您可以使用 **codegen** 命令从 MATLAB 代码编译 MEX 文件（如果您有 MATLAB Coder 许可证，则可以使用此功能）。此编译过程涉及到一些优化，对加速仿真非常有用。有关详细信息，请参阅“MATLAB Coder 快速入门”和“MATLAB 类”。

---

**注意** 大多数（但并非全部）System object 都支持代码生成。有关信息，请参阅特定对象的参考页。

---

## 通过 MATLAB Function 模块使用 System object

使用 MATLAB Function 模块，您可以在 Simulink 模型中包含任何 System object 和任何 MATLAB 语言函数。然后，即可基于该模型生成可嵌入代码。System object 为代码生成提供比大多数关联模块更高级别的算法。有关详细信息，请参阅“用 MATLAB Function 模块在 Simulink 中实现 MATLAB 函数”（Simulink）。

## 通过 MATLAB System 模块使用 System object

使用 MATLAB System 模块，您可以在 Simulink 模型中包含您使用类定义文件创建的各个 System object。然后，即可基于该模型生成可嵌入代码。有关详细信息，请参阅“MATLAB System 模块”（Simulink）。

## System object 和 MATLAB Compiler 软件

MATLAB Compiler 软件支持在 MATLAB 函数内使用 System object。编译器产品不支持在 MATLAB 脚本中使用 System object。

## 另请参阅

### 详细信息

- “Generate Code That Uses Row-Major Array Layout”





## 生成 C++ 类

---



## 函数句柄的代码生成

---

## 代码生成的函数句柄限制

当您在用于代码生成的 MATLAB 代码中使用函数句柄时，请遵循以下限制：

### 不要使用相同的约束变量引用不同函数句柄

在某些情况下，使用相同的约束变量引用不同函数句柄会导致编译时错误。例如，以下代码不执行编译：

```
function y = foo(p)
x = @plus;
if p
    x = @minus;
end
y = x(1, 2);
```

### 不要向 `coder.ceval` 传入函数句柄或从其传出函数句柄

您不能向 `coder.ceval` 传入函数句柄，也不能从其传出函数句柄。例如，假设 `f` 和 `str.f` 是函数句柄：

```
f = @sin;
str.x = pi;
str.f = f;
```

以下语句将导致编译错误：

```
coder.ceval('foo', @sin);
coder.ceval('foo', f);
coder.ceval('foo', str);
```

### 不要将函数句柄与外部函数相关联

您不能创建引用 MATLAB 外部函数的函数句柄。

### 不要向外部函数传入函数句柄或从其传出函数句柄

您不能向 `feval` 和其他 MATLAB 外部函数传入，或从其传出函数句柄。

### 不要向入口函数传入函数句柄或从其传出函数句柄

您不能向入口函数传入，也不能从其传出函数句柄。以如下函数为例：

```
function x = plotFcn(fhandle, data)

assert(isa(fhandle,'function_handle') && isa(data,'double'));

plot(data, fhandle(data));
x = fhandle(data);
```

在此示例中，函数 `plotFcn` 接收一个函数句柄及其数据作为输入。`plotFcn` 尝试使用输入 `data` 调用 `fhandle` 引用的函数，并绘制结果。但是，此代码会生成编译错误。该错误指示在 MATLAB 函数内执行调用以指定输入的属性时，函数 `isa` 不会将 `'function_handle'` 识别为类名。

## 另请参阅

## 详细信息

- “使用 coder.extrinsic 构造” (第 20-6 页)



## 深度学习数组的代码生成

---





## 为代码生成定义函数

---

## 匿名函数的代码生成

您可以在用于代码生成的 MATLAB 代码中使用匿名函数。例如，您可以为以下 MATLAB 代码生成代码，这些代码定义求数字平方的匿名函数。

```
sqr = @(x) x.^2;  
a = sqr(5);
```

若要创建传递给在某一值范围内计算表达式的 MATLAB 函数的函数句柄，匿名函数非常有用。例如，以下 MATLAB 代码使用匿名函数创建 `fzero` 函数的输入：

```
b = 2;  
c = 3.5;  
x = fzero(@(x) x^3 + b*x + c,0);
```

### 用于代码生成的匿名函数限制

匿名函数对于值类和元胞数组有代码生成限制。

### 另请参阅

### 详细信息

- “用于代码生成的 MATLAB 类定义” (第 15-2 页)
- “代码生成的元胞数组限制” (第 9-2 页)
- “参数化函数”

## 为代码生成调用函数

---

- “代码生成的函数调用解析” (第 20-2 页)
- “代码生成路径中文件类型的解析” (第 20-3 页)
- “编译指令 `%#codegen`” (第 20-4 页)
- “使用 MATLAB 引擎在生成的代码中执行函数调用” (第 20-5 页)

## 代码生成的函数调用解析

从 MATLAB 函数，您可以调用局部函数、支持的工具箱函数和其他 MATLAB 函数。MATLAB 按如下方式为代码生成解析函数名称：

### 函数调用解析的要点

此图说明了关于 MATLAB 如何为代码生成解析函数调用的要点：

- 搜索两条路径，即代码生成路径和 MATLAB 路径

请参阅“编译路径搜索顺序”（第 20-2 页）。

- 尝试编译函数，除非代码生成器确定不应编译它们或者您显式声明它们是外部函数。

如果代码生成不支持 MATLAB 函数，则可以使用构造 `coder.extrinsic` 将其声明为外部函数，如“使用 `coder.extrinsic` 构造”（第 20-6 页）中所述。在仿真过程中，代码生成器会为外部函数的调用生成代码，但不会为函数生成内部代码。因此，仿真只能在安装了 MATLAB 软件的平台上运行。在独立代码生成过程中，代码生成器会尝试确定外部函数是否影响调用时所在函数的输出 - 例如，通过将 `mxArrays` 返回给输出变量。如果输出不变，将继续进行代码生成，但会从生成的代码中排除外部函数。否则，将出现编译错误。

代码生成器可以检测到对多种常见可视化函数（如 `plot`、`disp` 和 `figure`）的调用。该软件将这些函数视为外部函数，但您不需要使用 `coder.extrinsic` 函数将它们声明为外部函数。

- 根据“代码生成路径中文件类型的解析”（第 20-3 页）中描述的优先级规则解析文件类型

### 编译路径搜索顺序

在代码生成过程中，会在两条路径上解析函数调用：

#### 1 代码生成路径

在代码生成过程中，MATLAB 会先搜索此路径。代码生成路径包含代码生成支持的工具箱函数。

#### 2 MATLAB 路径

如果函数不在代码生成路径上，MATLAB 将搜索此路径。

MATLAB 在搜索每条路径时应用相同的调度规则（请参阅“函数优先顺序”）。

### 何时使用代码生成路径

使用代码生成路径用自定义版本覆盖 MATLAB 函数。代码生成路径上的文件会隐藏 MATLAB 路径上的同名文件。

有关如何向代码生成路径添加其他文件夹的详细信息，请参阅“Paths and File Infrastructure Setup”。

## 代码生成路径中文件类型的解析

MATLAB 使用以下优先级规则进行代码生成：

## 编译指令 `##codegen`

将 `##codegen` 指令（或编译指令）添加到函数中的函数签名之后，以指示您要为 MATLAB 算法生成代码。添加此指令将指示 MATLAB 代码分析器帮助您诊断并修复在代码生成过程中会导致错误的违规情况。

```
function y = my_fcn(x) ##codegen
```

```
....
```

---

**注意** MATLAB Function 模块不需要 `##codegen` 指令。MATLAB Function 模块内的代码始终用于代码生成。不管有没有 `##codegen` 指令，都不会更改错误检查行为。

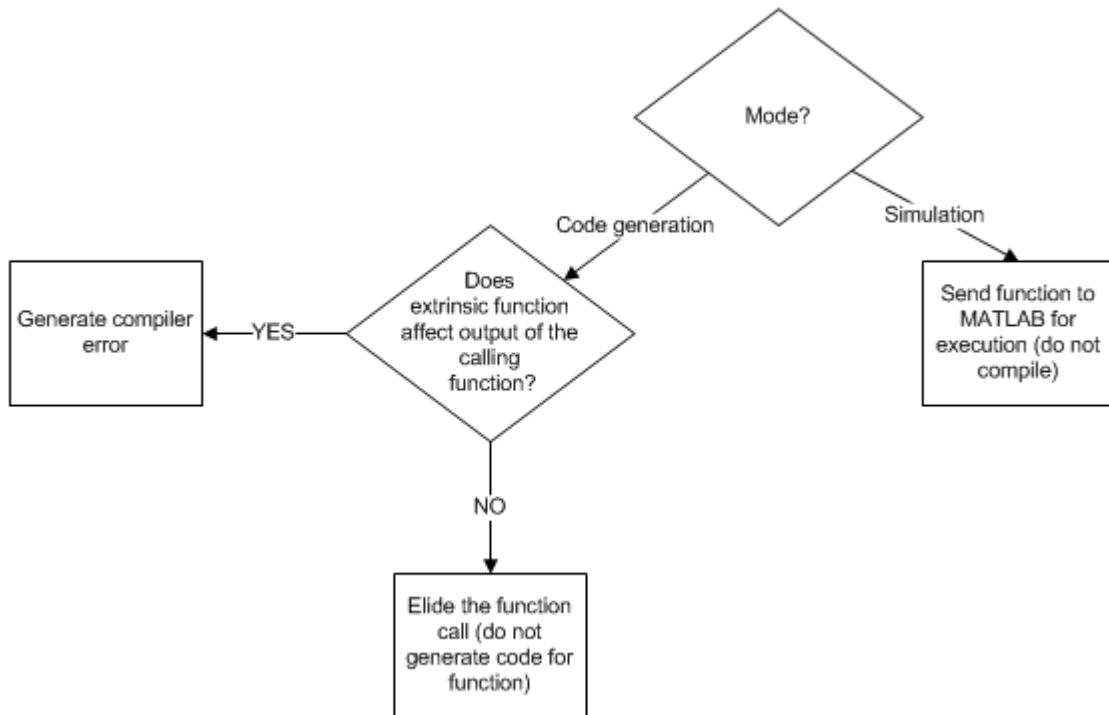
---

## 使用 MATLAB 引擎在生成的代码中执行函数调用

在 MATLAB 代码中处理对函数 `foo` 的调用时，代码生成器会找到 `foo` 的定义并为其函数体生成代码。在某些情况下，您可能希望绕过代码生成，而是使用 MATLAB 引擎来执行调用。使用 `coder.extrinsic('foo')` 声明对 `foo` 的调用不生成代码，而是使用 MATLAB 引擎执行。在此上下文中，`foo` 称为外部函数。在执行期间，仅当 MATLAB 引擎可用时，此功能才可用。这种情况的示例包括 MEX 函数的执行、Simulink 仿真或代码生成时的函数调用（也称为编译时）。

如果为调用 `foo` 并包含 `coder.extrinsic('foo')` 的函数生成独立代码，代码生成器将尝试确定 `foo` 是否影响输出。如果 `foo` 不影响输出，则代码生成器继续生成代码，但会从生成的代码中排除 `foo`。否则，代码生成器会产生编译错误。

将 `coder.extrinsic('foo')` 指令包含在某特定 MATLAB 函数中会将该 MATLAB 函数中对 `foo` 的所有调用都声明为外部调用。或者，您可能希望将外部声明的范围缩小到仅对 `foo` 的一次调用。请参阅“使用 `feval` 调用 MATLAB 函数”（第 20-8 页）。



### 将函数声明为外部函数的情形

以下是您可能会考虑将 MATLAB 函数声明为外部函数的一些常见情况：

- 函数执行显示或记录操作。此类函数主要在仿真期间有用，在嵌入式系统中不使用。
- 在您的 MEX 执行或 Simulink 仿真中，您要使用代码生成不支持的 MATLAB 函数。此工作流不适用于非仿真目标。
- 您使用 `coder.const` 指示代码生成器对函数调用进行常量折叠。在这种情况下，仅当 MATLAB 引擎可用于执行调用时，才会在代码生成期间调用该函数。

## 使用 coder.extrinsic 构造

要将函数 `foo` 声明为外部函数，请在您的 MATLAB 代码中包含以下语句。

```
coder.extrinsic('foo')
```

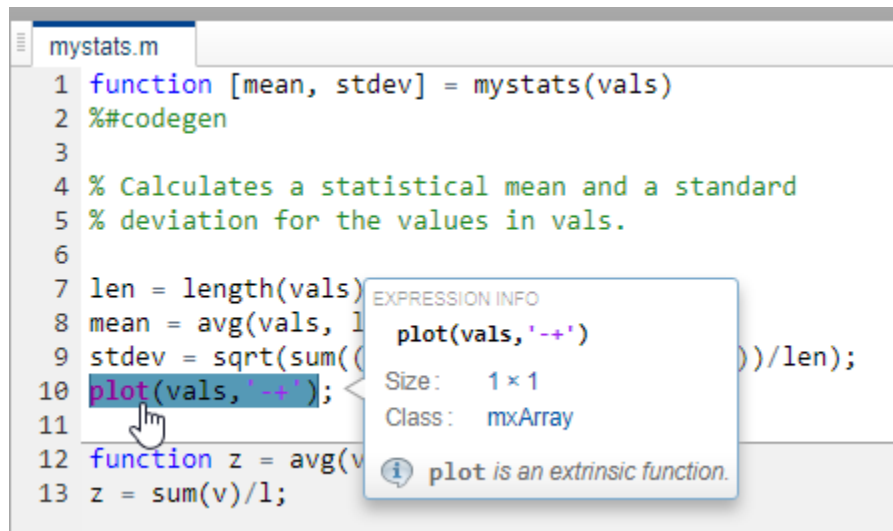
当将函数声明为代码生成的外部函数时，请遵守以下规则：

- 在调用函数之前将其声明为外部函数。
- 不要在条件语句中使用外部声明。
- 将外部函数的返回值赋给已知类型。请参阅“使用 `mxArray`”（第 20-8 页）。

有关其他信息和示例，请参阅 `coder.extrinsic`。

代码生成器自动将许多常见的 MATLAB 可视化函数（例如 `plot`、`disp` 和 `figure`）视为外部函数。您不必使用 `coder.extrinsic` 将它们显式声明为外部函数。例如，您可能希望通过调用 `plot` 来可视化 MATLAB 环境中的结果。如果您从调用 `plot` 的函数中生成一个 MEX 函数并运行该 MEX 函数，代码生成器会将 `plot` 函数的调用调度给 MATLAB 引擎。如果您生成一个库或可执行文件，生成的代码中将不包含对 `plot` 函数的调用。

如果您使用 MATLAB Code 生成 MEX 或独立 C/C++ 代码，代码生成报告将突出显示您的 MATLAB 代码对外部函数的调用。通过检查报告，您可以确定哪些函数仅在 MATLAB 环境中受支持。



### 外部函数声明的作用域

`coder.extrinsic` 构造具有函数作用域。以如下代码为例：

```
function y = foo %#codegen
coder.extrinsic('rat','min');
[N D] = rat(pi);
y = 0;
y = min(N, D);
```

在此示例中，`rat` 和 `min` 每次在主函数 `foo` 中调用时都被视为外部函数。有两种方式可以缩小外部函数声明在主函数内的作用域：



- 在局部函数中将 MATLAB 函数声明为外部函数，如以下示例中所示：

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = mymin(N, D);

function y = mymin(a,b)
coder.extrinsic('min');
y = min(a,b);
```

这里，函数 `rat` 每次在主函数 `foo` 内调用时都是外部函数，但函数 `min` 只有在局部函数 `mymin` 内调用时才是外部函数。

- 请使用 `feval` 调用 MATLAB 函数，而不是使用 `coder.extrinsic` 构造。下一节将介绍这种方法。

### 对非静态方法进行外部声明

假设您定义一个具有非静态方法 `foo` 的类 `myClass`，然后创建此类的实例 `obj`。如果要在用于代码生成的 MATLAB 代码中将方法 `obj.foo` 声明为外部方法，请遵循以下规则：

- 将对 `foo` 的调用编写为函数调用。不要使用圆点表示法来编写调用。
- 使用语法 `coder.extrinsic('foo')` 将 `foo` 声明为外部函数。

例如，将 `myClass` 定义为：

```
classdef myClass
    properties
        prop = 1
    end
    methods
        function y = foo(obj,x)
            y = obj.prop + x;
        end
    end
end
```

下面是将 `foo` 声明为外部函数的 MATLAB 函数示例。

```
function y = myFunction(x) %#codegen
coder.extrinsic('foo');
obj = myClass;
y = foo(obj,x);
end
```

非静态方法也称为普通方法。请参阅“方法语法”。

### 其他用途

使用 `coder.extrinsic` 构造可以：

- 调用在仿真过程中不会生成输出，从而不会生成不必要代码的 MATLAB 函数。
- 使您的代码具有自说明，从而更容易调试。您可以扫描 `coder.extrinsic` 语句的源代码以隔离对 MATLAB 函数的调用，它们可能会创建并传播 `mxArrays`。请参阅“使用 `mxArray`”（第 20-8 页）。

## 使用 feval 调用 MATLAB 函数

要将外部声明的作用域缩小到近一个函数调用，请使用函数 `feval`。`feval` 在代码生成过程中自动解释为外部函数。因此，您可以使用 `feval` 调用要在 MATLAB 环境中执行的函数，而不用编译为生成的代码。

请参考如下示例：

```
function y = foo
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0;
y = feval('min',N,D);
```

因为 `feval` 是外部函数，所以语句 `feval('min',N,D)` 由 MATLAB 计算（而不进行编译），这样产生的结果与专门为此调用将 `min` 声明为外部函数相同。相比之下，函数 `rat` 在整个函数 `foo` 中都是外部函数。

代码生成器不支持使用 `feval` 来调用局部函数或位于私有文件夹中的函数。

## 使用 mxArray

外部函数的运行时输出是 `mxArray`，也称为 MATLAB 数组。对 `mxArrays` 有效的操作只有下列几个：

- 在变量中存储 `mxArray`。
- 将 `mxArray` 传递给外部函数。
- 将函数中的 `mxArray` 返回到 MATLAB。
- 运行时将 `mxArray` 转换为已知类型。请将 `mxArray` 赋给变量，此变量的类型已由以前的赋值定义。请参阅以下示例。

要在其他操作中使用外部函数返回的 `mxArray`（例如，从 MATLAB Function 模块返回到 Simulink 执行），必须首先将其转换为已知类型。

如果函数的输入参数是 `mxArrays`，代码生成器会自动将该函数视为外部函数。

### 将 mxArray 转换为已知类型

要将 `mxArray` 转换为已知类型，请将 `mxArray` 指定给已定义类型的变量。在运行时，`mxArray` 将被转换为其指定给的变量的类型。如果 `mxArray` 中的数据与该变量的类型不一致，将会发生运行时错误。

以如下代码为例：

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = min(N,D);
```

这里，顶层函数 `foo` 调用外部 MATLAB 函数 `rat`，后者返回两个 `mxArrays`，分别代表 `pi` 的有理分式逼近式的分子 `N` 和分母 `D`。您可以将这些 `mxArrays` 传递给另一个 MATLAB 函数，在本例中为 `min`。由于传递给 `min` 的输入是 `mxArrays`，代码生成器会自动将 `min` 视为外部函数。因此，`min` 返回 `mxArray`。

使用 MATLAB Coder 生成 MEX 函数时，您可以将 `min` 返回的此 `mxArray` 直接赋给输出 `y`，因为 MEX 函数将其输出返回到 MATLAB。

```
codegen foo
```

Code generation successful.

但是，如果您将 `foo` 放在 Simulink 模型的 MATLAB Function 模块中，然后更新或运行该模型，则会出现以下错误：

Function output 'y' cannot be an mxArray in this context.  
Consider preinitializing the output variable with a known type.

发生此错误是因为不支持将 `mxArray` 返回到 Simulink。要解决此问题，请将 `y` 定义为您希望 `min` 返回的值的类型和大小，本例中为双精度标量值：

```
function y = foo %#codegen
coder.extrinsic('rat');
[N D] = rat(pi);
y = 0; % Define y as a scalar of type double
y = min(N,D);
```

在此示例中，外部函数 `min` 的输出会影响您为其生成代码的入口函数 `foo` 的输出 `y`。如果您尝试为 `foo` 生成独立代码（例如静态库），代码生成器将无法忽略外部函数调用，并产生代码生成错误。

`codegen -config:lib foo`

??? The extrinsic function 'min' is not available for standalone code generation. It must be eliminated for stand-alone code to be generated. It could not be eliminated because its outputs appear to influence the calling function. Fix this error by not using 'min' or by ensuring that its outputs are unused.

Error in ==> foo Line: 4 Column: 5  
Code generation failed: View Error Report

Error using codegen

## 使用外部函数的限制

代码生成过程中不支持完整的 MATLAB 运行时环境。因此，从外部调用 MATLAB 函数时存在以下限制：

- 用来检查调用方或读取/写入调用方工作区的 MATLAB 函数在代码生成过程中不起作用。此类函数包括：
  - `dbstack`
  - `evalin`
  - `assignin`
  - `save`
- 如果您的外部函数在运行时执行以下操作，则生成的代码中的函数可能会产生不可预知的结果：
  - 更改文件夹
  - 更改 MATLAB 路径
  - 删除或添加 MATLAB 文件
  - 更改警告状态
  - 更改 MATLAB 预设
  - 更改 Simulink 参数

- 代码生成器不支持使用 `coder.extrinsic` 来调用位于私有文件夹中的函数。
- 代码生成器不支持使用 `coder.extrinsic` 来调用局部函数。
- 您可以调用最多具有 64 个输入和 64 个输出的外部函数。

### 另请参阅

`coder.extrinsic` | `coder.const`

## 定点转换

---



## 使用编程 workflow 自动执行定点转换

---





## 单精度转换

---



## 设置 MATLAB Coder 工程

---

- “设置 MATLAB Coder 工程” (第 24-2 页)
- “使用 App 指定全局变量类型和初始值” (第 24-3 页)

## 设置 MATLAB Coder 工程

- 1 要打开该 App，请在 MATLAB 工具条的 **Apps** 选项卡上，点击 **Code Generation** 下的 MATLAB Coder App 图标。
- 2 创建一个工程或打开一个现有工程。请参阅“创建工程”（第 24-2 页）和“打开现有工程”（第 24-2 页）。
- 3 如果 App 在入口函数中检测到代码生成就绪方面的问题，请解决这些问题。
- 4 定义入口函数输入类型的属性。请参阅“Specify Properties of Entry-Point Function Inputs Using the App”。
- 5 检查是否存在运行时问题。提供代码或 App 可用于测试代码的测试文件。App 将生成一个 MEX 函数。它运行您的测试代码或测试文件，将对 MATLAB 函数的调用替换为对 MEX 函数的调用。此步骤是可选的。不过，建议最好执行此步骤。您可以检测并解决在生成的 C 代码中更难诊断出来的运行时错误。
- 6 配置编译设置。选择编译类型、语言和生产硬件。（可选）修改其他编译设置。请参阅“配置编译设置”（第 27-12 页）。

现在您便可以生成代码了。

### 创建工程

在 **Select Source Files** 页面上，指定要为其生成代码的 MATLAB 文件。入口函数是从 MATLAB 调用的函数。不要添加名称中包含空格的文件。

App 创建具有第一个入口函数名称的工程。

### 打开现有工程

- 1 在 App 工具栏上，点击  并选择 **Open existing project**。
- 2 键入或选择工程。

App 将关闭其他打开的工程。

MATLAB Online™ 不支持 MATLAB Coder App。

如果该工程是一个定点转换器工程，并且您具有 Fixed-Point Designer 许可证，则该工程将在定点转换器中打开。

# 使用 App 指定全局变量类型和初始值

本节内容
“为什么要为全局变量指定类型定义？” (第 24-3 页)
“指定全局变量类型” (第 24-3 页)
“通过示例定义全局变量” (第 24-3 页)
“定义或编辑全局变量类型” (第 24-4 页)
“定义全局变量初始值” (第 24-4 页)
“定义全局变量常量值” (第 24-5 页)
“删除全局变量” (第 24-5 页)

## 为什么要为全局变量指定类型定义？

如果在 MATLAB 算法中使用全局变量，则在构建工程之前，必须为每个全局变量添加全局类型定义和初始值。如果不初始化全局数据，则 App 将在 MATLAB 全局工作区中查找变量。如果变量不存在，App 会生成错误。

对于 MEX 函数，如果您使用全局数据，还必须指定是否在 MATLAB 和 MEX 函数之间同步这些数据。

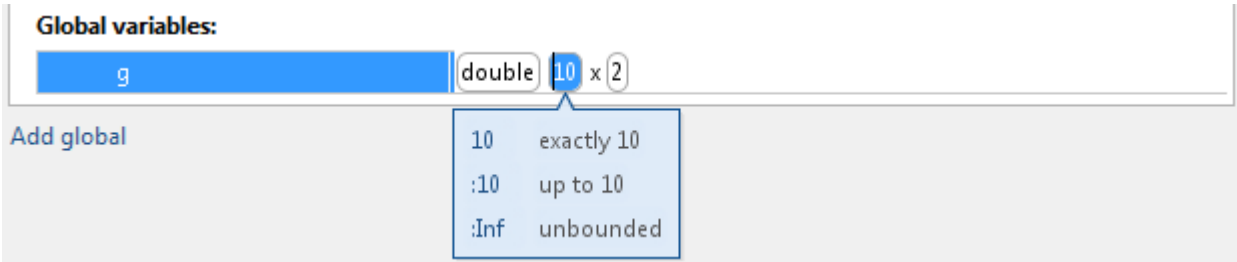
## 指定全局变量类型

- 使用以下方法之一指定每个全局变量的类型：
  - 通过示例定义 (第 24-3 页)
  - 定义类型 (第 24-4 页)
- 为每个全局变量定义初始值 (第 24-4 页)。

如果不为全局变量提供类型定义和初始值，请在 MATLAB 工作区中创建一个具有相同名称和适当的类、大小、复/实性和值的变量。

## 通过示例定义全局变量

- 点击要定义的全局变量右侧的字段。
- 选择“按示例定义”。
- 在全局名称右侧的字段中，输入具有所需的类、大小和复/实性的 MATLAB 表达式。MATLAB Coder 软件使用此表达式的值的类、大小和复/实性作为全局变量的类型。
- (可选) 更改全局变量的大小。点击要更改的维度，然后输入大小，例如 10。



您可以指定：

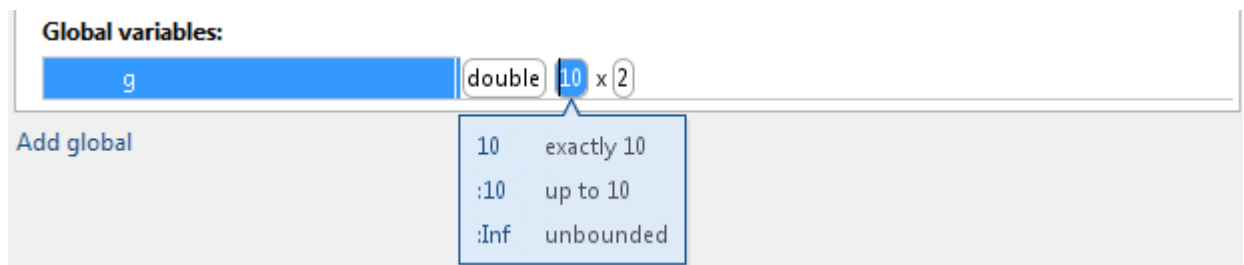
- 固定大小。在此示例中，选择 **10**。
- 可变大小，通过使用 **:** 前缀来指定，最高不超过指定的限制。在此示例中，要指定输入的大小不超过 **10**，请选择 **:10**。
- 无界可变大小，通过选择 **:Inf** 来指定。

## 定义或编辑全局变量类型

- 1 点击要定义的全局变量右侧的字段。
- 2 （可选）对于数值类型，请选择**复数**使参数成为复数类型。默认情况下，输入是实数。
- 3 选择全局变量的类型。例如，**"double"**。

默认情况下，全局变量是标量。

- 4 （可选）更改全局变量的大小。点击要更改的维度，然后输入大小，例如 **10**。



您可以指定：

- 固定大小。在此示例中，选择 **10**。
- 可变大小，通过使用 **:** 前缀来指定，最高不超过指定的限制。在此示例中，要指定输入的大小不超过 **10**，请选择 **:10**。
- 无界可变大小，通过选择 **:Inf** 来指定。

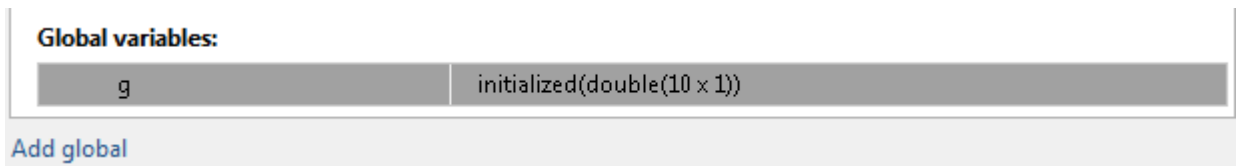
## 定义全局变量初始值

- “在定义类型之前定义初始值” （第 24-4 页）
- “在定义类型后定义初始值” （第 24-5 页）

### 在定义类型之前定义初始值

- 1 点击全局变量右侧的字段。
- 2 选择“定义初始值”。
- 3 输入 MATLAB 表达式。MATLAB Coder 软件使用指定的 MATLAB 表达式的值作为全局变量的值。由于您在定义全局变量的初始值之前未定义其类型，MATLAB Coder 将初始值类型用作全局变量类型。

工程显示全局变量已初始化。

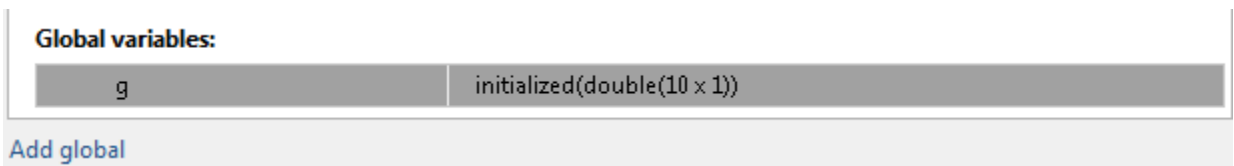


如果在定义全局变量的初始值后更改其类型，则必须重新定义初始值。

### 在定义类型后定义初始值

- 点击预定义的全局变量的类型字段。
- 选择“定义初始值”。
- 输入 MATLAB 表达式。MATLAB Coder 软件使用指定的 MATLAB 表达式的值作为全局变量的值。

工程显示全局变量已初始化。



### 定义全局变量常量值

- 1 点击全局变量右侧的字段。
- 2 选择“定义常量值”。
- 3 在全局变量右侧的字段中，输入 MATLAB 表达式。

### 删除全局变量

- 1 右键点击全局变量。
- 2 从菜单中选择删除全局变量。



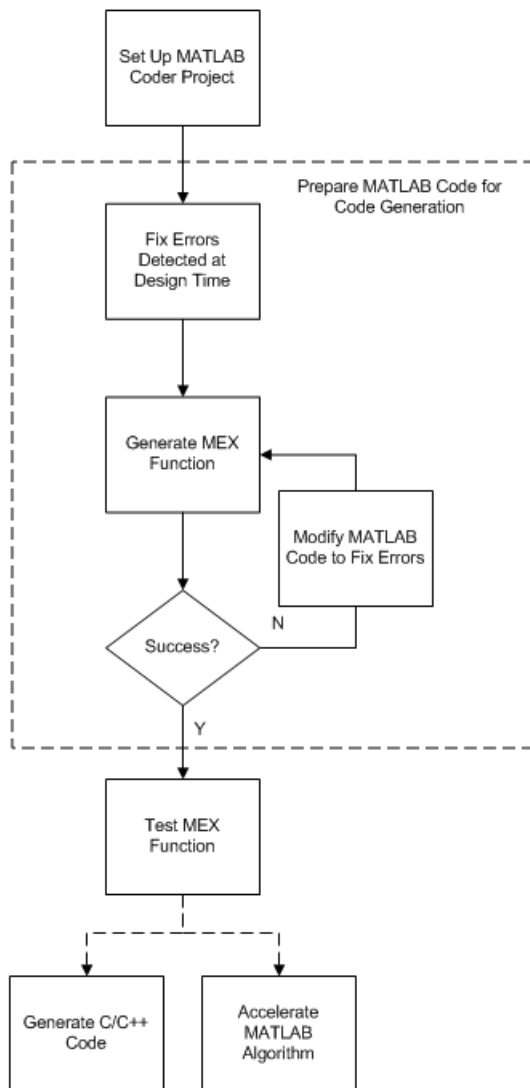


## 准备用于 C/C++ 代码生成的 MATLAB 代码

---

- “准备 MATLAB 代码以用于代码生成的工作流” (第 25-2 页)
- “代码生成就绪工具” (第 25-3 页)
- “使用 MATLAB Coder App 生成 MEX 函数” (第 25-5 页)
- “修复在代码生成时检测到的错误” (第 25-9 页)

## 准备 MATLAB 代码以用于代码生成的工作流



### 另请参阅

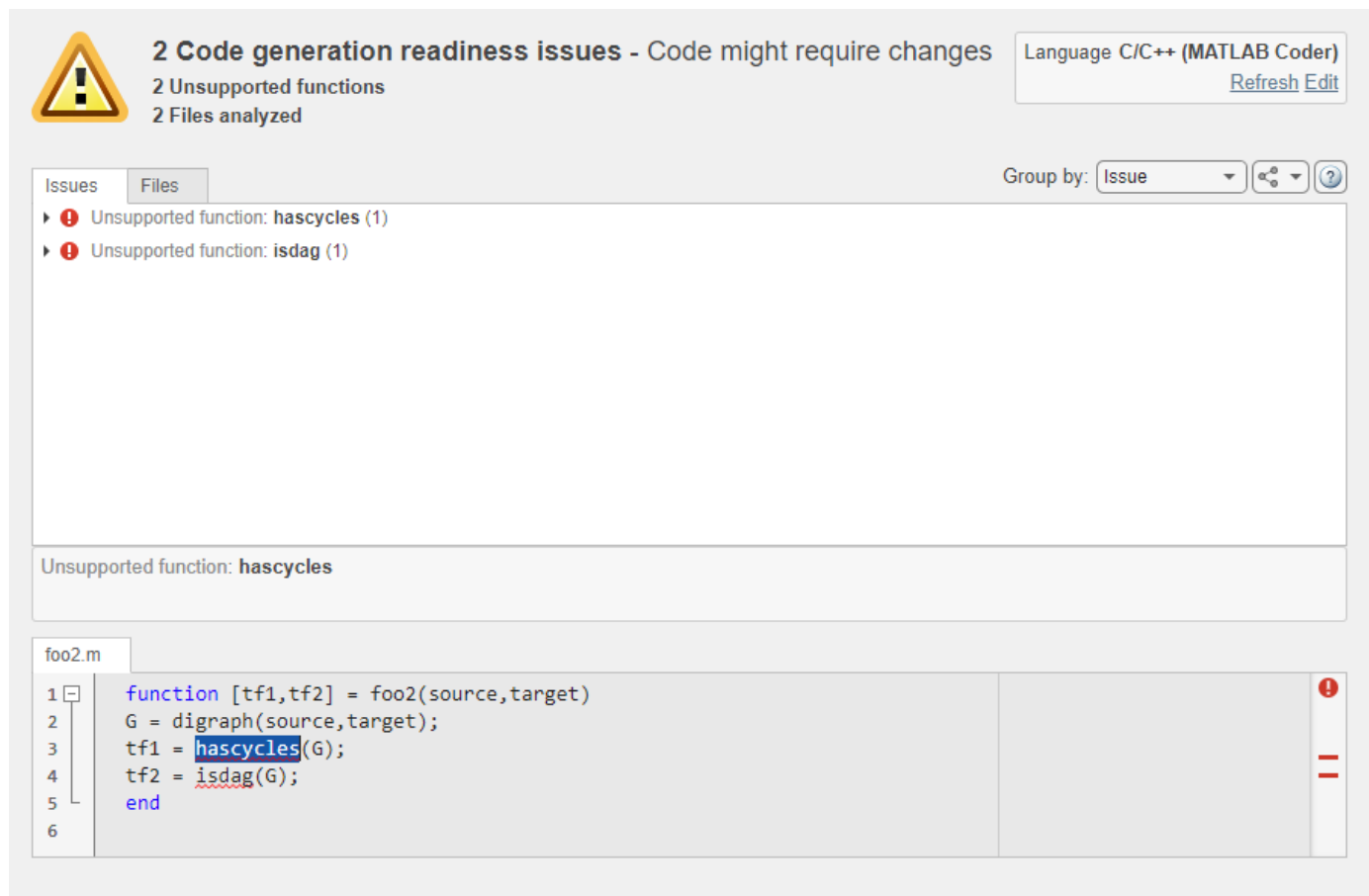
- “设置 MATLAB Coder 工程”（第 24-2 页）
- “Fixing Errors Detected at Design Time”
- “使用 MATLAB Coder App 生成 MEX 函数”（第 25-5 页）
- “修复在代码生成时检测到的错误”（第 25-9 页）
- “Workflow for Testing MEX Functions in MATLAB”
- “Accelerate MATLAB Algorithms”

## 代码生成就绪工具

代码生成就绪工具会筛查 MATLAB 代码中是否存在代码生成不支持的功能和函数。该工具提供的报告会列出包含不支持的功能和函数的源文件。该工具可能无法检测到所有代码生成问题。在某些情况下，该工具可能会报告伪错误。因此，在生成代码之前，请通过生成 MEX 函数来验证您的代码是否适合代码生成。

代码生成就绪工具不会报告代码生成器自动视为外部函数的函数。这些函数的示例有 **plot**、**disp** 和 **figure**。

### “问题”选项卡



**2 Code generation readiness issues - Code might require changes** Language C/C++ (MATLAB Coder)  
 2 Unsupported functions Refresh Edit  
 2 Files analyzed

Issues Files Group by: Issue

- Unsupported function: **hascycles** (1)
- Unsupported function: **isdag** (1)

Unsupported function: **hascycles**

foo2.m

```

1 function [tf1,tf2] = foo2(source,target)
2 G = digraph(source,target);
3 tf1 = hascycles(G);
4 tf2 = isdag(G);
5 end
6
```

在**问题**选项卡上，该工具显示与以下项相关的信息：

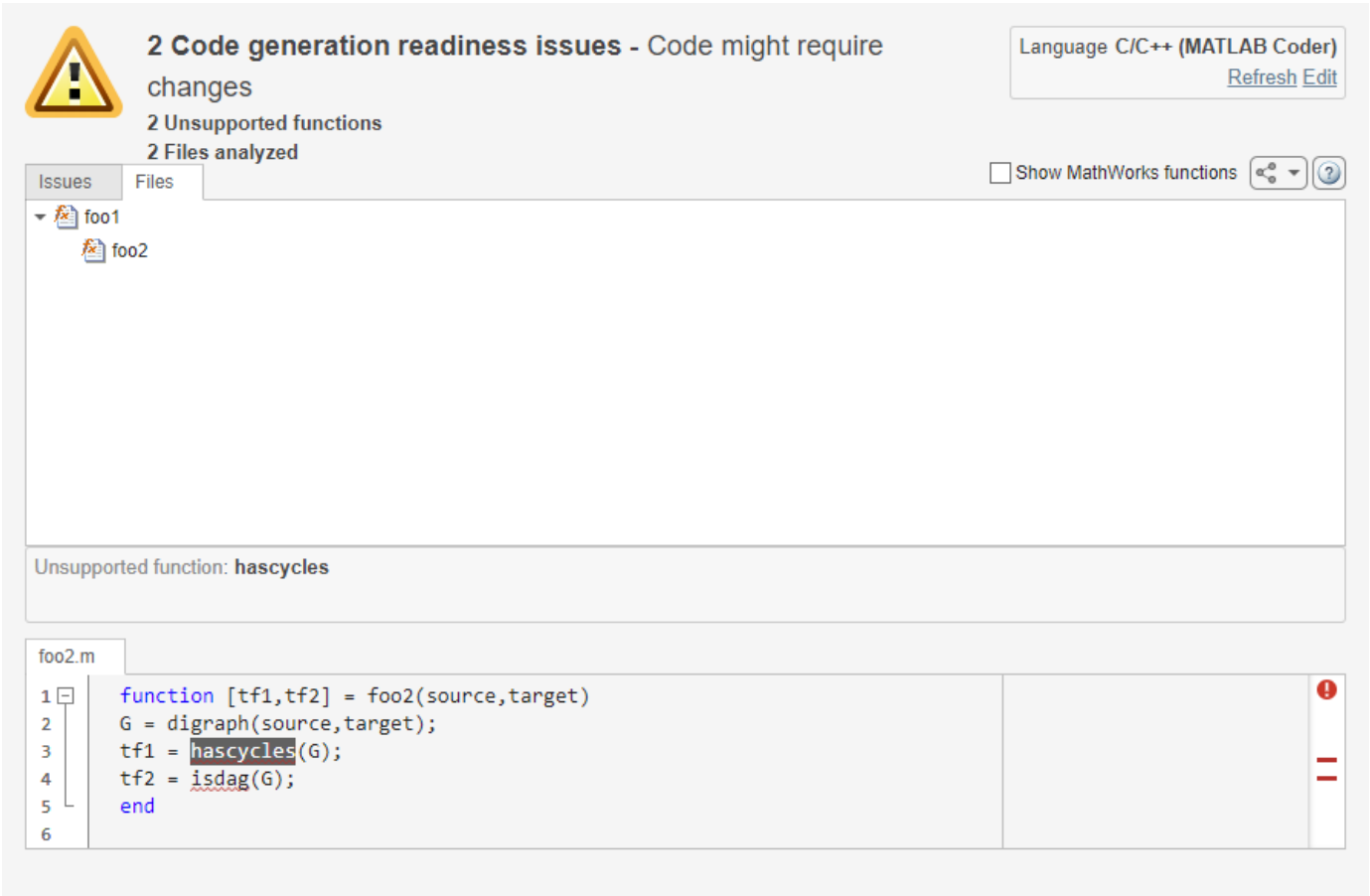
- MATLAB 语法问题。这些问题在 MATLAB 编辑器中报告。要了解有关这些问题以及如何解决这些问题的详细信息，请使用代码分析器。
- 不支持的 MATLAB 函数调用、语言功能和数据类型。

您还可以：

- 在代码生成就绪工具中查看您的 MATLAB 代码。当您选择问题时，您的 MATLAB 代码中导致该问题的部分会突出显示。

- 按问题或文件对就绪结果进行分组。
- 选择代码生成就绪分析使用的语言。
- 如果您更新 MATLAB 代码，请刷新代码生成就绪分析。
- 将分析报告导出为纯文本文件或基础工作区中的 `coder.ScreenerInfo` 对象。

“文件” 选项卡



如果您正在检查的代码调用其他 MATLAB 代码文件中的函数，则**文件**选项卡会显示这些文件之间的调用依存关系。如果选择**显示 MathWorks 函数**，报告还会列出您的函数调用的 MathWorks 函数。

另请参阅

`coder.screener` | `coder.ScreenerInfo` Properties

相关示例

- “支持 C/C++ 代码生成的 MATLAB 语言功能” （第 2-22 页）
- “C/C++ 代码生成支持的函数和对象” （第 3-2 页）

# 使用 MATLAB Coder App 生成 MEX 函数

本节内容
“使用 MATLAB Coder App 生成 MEX 函数的工作流” （第 25-5 页）
“使用 MATLAB Coder App 生成 MEX 函数” （第 25-5 页）
“配置工程设置” （第 25-7 页）
“编译 MATLAB Coder 工程” （第 25-7 页）
“另请参阅” （第 25-7 页）

## 使用 MATLAB Coder App 生成 MEX 函数的工作流

步骤	操作	详细信息
1	设置 MATLAB Coder 工程。	“设置 MATLAB Coder 工程” （第 24-2 页）
2	指定编译配置参数。将 <b>Build type</b> 设置为 “MEX” 。	“配置工程设置” （第 25-7 页）
3	编译工程。	“编译 MATLAB Coder 工程” （第 25-7 页）

MATLAB Online 不支持 MATLAB Coder App。要在 MATLAB Online 中生成 MEX 函数，请使用 `codegen` 命令。

## 使用 MATLAB Coder App 生成 MEX 函数

此示例说明如何使用 MATLAB Coder App 从 MATLAB 代码生成 MEX 函数。

### 创建入口函数

在一个本地可写文件夹中，创建 MATLAB 文件 `mcadd.m`，其中包含：

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

### 创建测试文件

在同一本地可写文件夹中，创建 MATLAB 文件 `mcadd_test.m`，该文件调用具有示例输入的 `mcadd`。示例输入是类型为 `int16` 的标量。

```
function y = mcadd_test
y = mcadd(int16(2), int16(3));
```

### 打开 MATLAB Coder App

在 MATLAB 工具条的 **App** 选项卡上，点击 **Code Generation** 下的 MATLAB Coder App 图标。

该 App 会打开 **Select Source Files** 页面。

### 指定源文件

- 1 在 **Select Source Files** 页面中，键入或选择入口函数 `mcadd` 的名称。

该 App 将使用默认名称 `mcadd.prj` 创建一个工程。

- 2 点击 **Next** 以转到 **Define Input Types** 步骤。该 App 将分析函数以查找编码问题并确定代码生成就绪情况。如果 App 发现问题，它将打开 **Review Code Generation Readiness** 页面，您可以在其中查看和解决问题。在此示例中，由于 App 没有检测到问题，因此将打开 **Define Input Types** 页面。

### 定义输入类型

由于 C 使用静态类型，MATLAB Coder 必须在编译时确定 MATLAB 文件中所有变量的属性。您必须指定所有入口函数输入的属性。根据入口函数输入的属性，MATLAB Coder 可以推断 MATLAB 文件中所有变量的属性。

指定 MATLAB Coder 用来自动定义 `u` 和 `v` 的类型的测试文件 `mcadd_test.m`：

- 1 输入或选择测试文件 `mcadd_test.m`。
- 2 点击 **Autodefine Input Types**。

测试文件 `mcadd_test.m` 使用示例输入类型调用入口函数 `mcadd`。MATLAB Coder 推断输入 `u` 和 `v` 为 `int16(1x1)`。

- 3 点击 **Next** 以转到 **Check for Run-Time Issues** 步骤。

### 检查运行时间问题

**Check for Run-Time Issues** 步骤从您的入口函数生成 MEX 文件，然后运行 MEX 函数并报告问题。此步骤是可选的。不过，建议最好执行此步骤。您可以检测并解决在生成的 C 代码中更难诊断出来的运行时错误。

- 1 要打开 **Check for Run-Time Issues** 对话框，请点击 **Check for Issues** 箭头 。


App 使用 `mcadd_test` 填充测试文件字段，该测试文件用于定义输入类型。

- 2 点击 **Check for Issues**。

App 将生成一个 MEX 函数。它运行测试文件，将对 `mcadd` 的调用替换为对 MEX 函数的调用。如果 App 在 MEX 函数生成或执行过程中检测到问题，它将提供警告和错误消息。您可以点击这些消息，导航到有问题的代码并修复问题。在本示例中，App 未检测到问题。

- 3 点击 **Next** 以转到 **Generate Code** 步骤。

### 生成 MEX 函数

- 1 要打开 **Generate** 对话框，请点击 **Generate** 箭头 。

- 2 在 **Generate** 对话框中，将 **Build type** 设置为 “MEX”，将 **Language** 设置为 C。对其他工程编译配置设置使用默认值。

- 3 点击 **Generate**。

App 指示代码生成成功。它在页面左侧显示源 MATLAB 文件和生成的输出文件。在 **Variables** 选项卡上，它显示有关 MATLAB 源变量的信息。在 **Target Build Log** 选项卡上，它会显示编译日志，包括编译器警告和错误。

MATLAB Coder 编译工程，并默认在当前文件夹中生成 MEX 函数 `mcadd_mex`。MATLAB Coder 还会在名为 `codegen/mex/mcadd` 的子文件夹中生成其他支持文件。MATLAB Coder 使用 MATLAB 函数的名称作为生成文件的根名称。它为 MEX 文件创建一个特定于平台的扩展名。请参阅 “Naming Conventions”。


- 4 要查看代码生成报告，请点击 **View Report**。
- 5 点击 **Next** 打开 **Finish Workflow** 页面。

### 查看 Finish Workflow 页面

**Finish Workflow** 页面指示已成功生成代码，还提供工程摘要以及指向生成的输出的链接。

## 配置工程设置

要打开工程设置对话框，请执行下列操作：

- 1 要打开 **Generate** 对话框，请点击 **Generate** 箭头 .
- 2 点击 **More settings**。

要更改工程设置，请点击包含要更改的设置的选项卡。例如，要更改 **Saturate on integer overflow** 设置，请点击 **Speed** 选项卡。

MEX 函数使用一组与库和可执行文件不同的配置参数。将输出类型从 “MEX Function” 更改为 “Source Code”、“Static Library”、“Dynamic Library” 或 “Executable” 时，请验证这些设置。

某些配置参数与 MEX 和独立代码生成相关。如果您在输出类型为 “MEX Function” 时启用这些参数中的任何参数，并且您要在 C/C++ 代码生成中也使用相同的设置，则必须为 “C/C++ Static Library”、“C/C++ Dynamic Library” 和 “C/C++ Executable” 再次启用它。

### 另请参阅

- “Using the MATLAB Coder App”
- “How to Disable Inlining Globally Using the MATLAB Coder App”
- “Disabling Run-Time Checks Using the MATLAB Coder App”

## 编译 MATLAB Coder 工程

要使用指定的设置编译工程，请在 **Generate Code** 页面上点击 **Generate**。当 MATLAB Coder App 编译工程时，它会显示编译进度。编译完成后，App 会在 **Target Build Log** 选项卡上提供有关编译的详细信息。

如果启用了代码生成报告或发生编译错误，则 App 会生成报告。该报告提供有关最近编译的详细信息，并提供指向该报告的链接。

要查看报告，请点击 **View report** 链接。该报告提供指向 MATLAB 代码和生成的 C/C++ 文件的链接，并提供 MATLAB 代码中变量的编译时类型信息。如果发生编译错误，报告将列出错误和警告。

### 另请参阅

- “配置编译设置” (第 27-12 页)

### 另请参阅

### 相关示例

- “配置编译设置” (第 27-12 页)
- “使用 MATLAB Coder App 生成 C 代码”



## 修复在代码生成时检测到的错误

当代码生成器检测到错误或警告时，它会自动生成错误报告。错误报告会描述这些问题，并提供指向有错误的 MATLAB 代码的链接。

要修复错误，请修改您的 MATLAB 代码，使其仅使用代码生成支持的那些 MATLAB 功能。有关详细信息，请参阅“有关代码生成的编程注意事项”。选择一种调试策略来检测和更正 MATLAB 代码中的代码生成错误。有关详细信息，请参阅“Debugging Strategies”。

在完成代码生成后，软件会生成一个 MEX 函数，您可以使用该函数在 MATLAB 中测试您的实现。

如果您的 MATLAB 代码调用 MATLAB 路径中的函数，除非代码生成器确定这些函数应为外部函数或您声明它们是外部函数，否则它会尝试编译这些函数。请参阅“代码生成的函数调用解析”（第 20-2 页）。要获取详细的诊断信息，请将 `%#codegen` 指令添加到您要 `codegen` 编译的每个外部函数中。

### 另请参阅

- “Code Generation Reports”
- “为什么要在 MATLAB 中测试 MEX 函数？”（第 26-2 页）
- “可从 MATLAB 算法生成代码的情况”（第 2-2 页）
- “Debugging Strategies”
- “使用 `coder.extrinsic` 构造”（第 20-6 页）



## 在 MATLAB 中测试 MEX 函数

---

## 为什么要在 MATLAB 中测试 MEX 函数？

在为您的 MATLAB 代码生成 C/C++ 代码之前，最好的做法是测试 MEX 函数，以确认它提供与原始 MATLAB 代码相同的功能。要进行此测试，请使用与运行原始 MATLAB 代码时所用的相同输入运行 MEX 函数，并对结果进行比较。有关如何使用 MATLAB Coder App 测试 MEX 函数的详细信息，请参阅“Check for Run-Time Issues by Using the App”和“Verify MEX Functions in the MATLAB Coder App”。有关如何在命令行中测试 MEX 函数的详细信息，请参阅“Verify MEX Functions at the Command Line”。

在生成代码之前，在 MATLAB 中运行 MEX 函数可以检测并修复运行时错误，而在生成的代码中诊断这些错误要难得多。如果在 MATLAB 函数中遇到运行时错误，请在生成代码之前修复它们。请参阅“修复在代码生成时检测到的错误”（第 25-9 页）和“Debug Run-Time Errors”。

在 MATLAB 中运行 MEX 函数时，默认情况下会执行以下运行时检查：

- 内存完整性检查。这些检查执行数组边界检查和维度检查，并在为 MATLAB 函数生成的代码中检测内存完整性违规情况。如果检测到违规，MATLAB 将停止执行并提供诊断消息。
- 在为 MATLAB 函数生成的代码中的响应能力检查。这些检查会定期检查为 MATLAB 函数生成的代码中的 **Ctrl+C** 中断，以使您可以使用 **Ctrl+C** 终止执行。

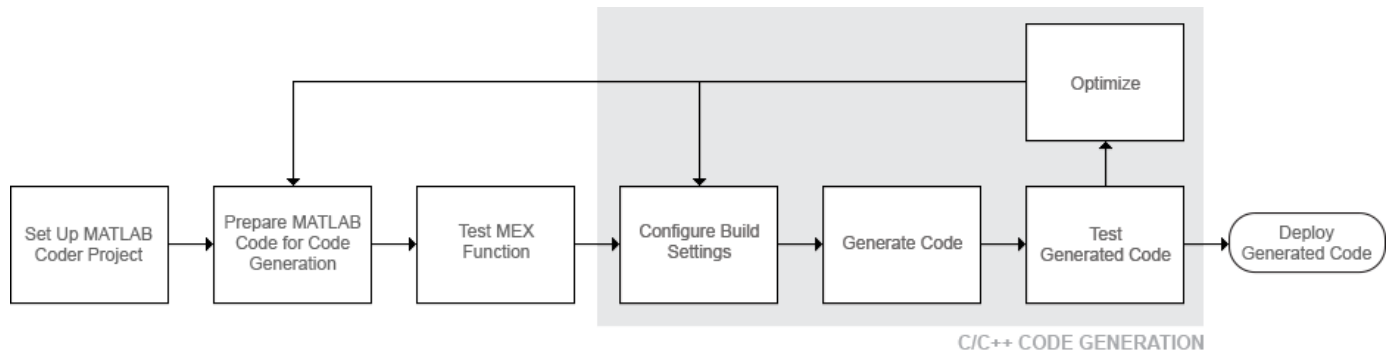
有关详细信息，请参阅“Control Run-Time Checks”。

# 从 MATLAB 代码生成 C/C++ 代码

---

- “代码生成 workflow” (第 27-2 页)
- “从 MATLAB 代码生成独立的 C/C++ 可执行文件” (第 27-3 页)
- “配置编译设置” (第 27-12 页)
- “在 macOS 平台上安装 OpenMP 库” (第 27-20 页)
- “生成代码以检测图像边缘” (第 27-21 页)
- “为 MATLAB 卡尔曼滤波算法生成 C 代码” (第 27-27 页)
- “使用 Black Litterman 方法生成用于优化投资组合的代码” (第 27-36 页)

## 代码生成 workflow



### 另请参阅

- “设置 MATLAB Coder 工程” (第 24-2 页)
- “准备 MATLAB 代码以用于代码生成的 workflow” (第 25-2 页)
- “Workflow for Testing MEX Functions in MATLAB”
- “配置编译设置” (第 27-12 页)

## 从 MATLAB 代码生成独立的 C/C++ 可执行文件

### 本节内容

“使用 MATLAB Coder App 生成 C 可执行文件” (第 27-3 页)  
 “在命令行中生成 C 可执行文件” (第 27-9 页)  
 “指定 C/C++ 可执行文件的主函数” (第 27-10 页)  
 “指定主函数” (第 27-10 页)

### 使用 MATLAB Coder App 生成 C 可执行文件

此示例说明如何使用 MATLAB Coder App 从 MATLAB 代码生成 C 可执行文件。在此示例中，您为 MATLAB 函数生成一个可执行文件，该函数用于生成随机标量值。使用该 App 可以执行以下操作：

- 1 生成示例 C `main` 函数，该函数调用生成的库函数。
- 2 复制并修改生成的 `main.c` 和 `main.h`。
- 3 修改工程设置，以便 App 可以找到修改后的 `main.c` 和 `main.h`。
- 4 生成可执行文件。

#### 创建入口函数

在本地可写文件夹中，创建 MATLAB 函数 `coderand`，该函数在开区间 (0,1) 上基于标准均匀分布生成一个随机标量值：

```
function r = coderand() %#codegen
r = rand();
```

#### 创建测试文件

在同一本地可写文件夹中，创建 MATLAB 文件 `coderand_test.m`，它调用 `coderand`。

```
function y = coderand_test()
y = coderand();
```

#### 打开 MATLAB Coder App

在 MATLAB 工具条的 **App** 选项卡上，单击**代码生成**下的 MATLAB Coder App 图标。

该 App 会打开**选择源文件**页面。

#### 指定源文件

- 1 在**选择源文件**页面中，键入或选择入口函数 `coderand` 的名称。

该 App 将使用默认名称 `coderand.prj` 在当前文件夹中创建一个工程。

- 2 单击**下一步**以转到**定义输入类型**步骤。该 App 将分析函数以查找编码问题并确定代码生成就绪情况。如果 App 发现问题，它将打开**检查代码就绪性**页面，您可以在其中查看和解决问题。在此示例中，由于 App 没有检测到问题，因此将打开**定义输入类型**页面。

## 定义输入类型

由于 C 使用静态定型，MATLAB Coder 必须在编译时确定 MATLAB 文件中所有变量的属性。您必须指定所有入口函数输入的属性。根据入口函数输入的属性，MATLAB Coder 可以推断 MATLAB 文件中所有变量的属性。

在此示例中，函数 `coderand` 没有输入。

点击**下一步**以转到**检查运行时问题**步骤。

## 检查运行时问题

**检查运行时问题**步骤从您的入口函数生成 MEX 文件，然后运行 MEX 函数并报告问题。此步骤是可选的。不过，建议最好执行此步骤。您可以检测并解决在生成的 C 代码中更难诊断出来的运行时错误。

- 1 要打开**检查运行时问题**对话框，请点击**检查问题**箭头 .

选择或输入测试文件 `coderand_test`。

- 2 点击**检查问题**。


App 将为 `coderand` 生成一个 MEX 函数。它运行测试文件，将对 `coderand` 的调用替换为对 MEX 函数的调用。如果 App 在 MEX 函数生成或执行过程中检测到问题，它将提供警告和错误消息。您可以点击这些消息，导航到有问题的代码并修复问题。在本示例中，App 未检测到问题。

- 3 点击**下一步**以转到**生成代码**步骤。

## 生成 C main 函数

在生成可执行文件时，您必须提供一个 C/C++ 主函数。默认情况下，当您生成 C/C++ 源代码、静态库、动态链接库或可执行文件时，MATLAB Coder 会生成 `main` 函数。这一生成的主函数是一个模板，您可以针对您的应用程序修改该模板。请参阅“使用示例主函数合并生成的代码”（第 32-13 页）。在复制和修改生成的主函数后，可以使用它来生成 C/C++ 可执行文件。您也可以编写自己的主函数。

在为 `coderand` 生成可执行文件之前，请生成 `main` 函数，该函数调用 `coderand`。

- 1 要打开**生成**对话框，请点击**生成**箭头 .
- 2 在**生成**对话框中，将**编译类型**设置为“源代码”，将**语言**设置为 C。对于其他工程编译配置设置，请使用默认值。
- 3 点击**更多设置**。
- 4 在**所有设置**选项卡的高级下，确认**生成示例主函数**设置为“生成但不编译示例主函数”。点击**关闭**。
- 5 点击**生成**。

MATLAB Coder 生成一个 `main.c` 文件和一个 `main.h` 文件。App 指示代码生成成功。

- 6 点击**下一步**打开**完成工作流**页面。

在**完成工作流**页面的**生成的输出**下，您会看到 `main.c` 位于子文件夹 `coderand\codegen\lib\coderand\examples` 中。

## 复制生成的示例主文件

由于后续代码生成可能覆盖生成的示例文件，因此在修改这些文件之前，请将它们复制到 `codegen` 文件夹之外的一个可写文件夹中。对于此示例，请将 `main.c` 和 `main.h` 从子文件夹 `coderand\codegen\lib\coderand\examples` 复制到一个可写文件夹中，例如 `c:\myfiles`。



## 修改生成的示例主文件

- 1 在包含示例主文件副本的文件夹中，打开 `main.c`。

### 生成的 `main.c`

```

/*****
/* This automatically generated example C main file shows how to call */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly. */
/* Instead, make a copy of this file, modify it, and integrate it into */
/* your development environment. */
/*
/* This file initializes entry-point function arguments to a default */
/* size and value before calling the entry-point functions. It does */
/* not store or use any values returned from the entry-point functions. */
/* If necessary, it does pre-allocate memory for returned values. */
/* You can use this file as a starting point for a main function that */
/* you can deploy in your application. */
/*
/* After you copy the file, and before you deploy it, you must make the */
/* following changes: */
/* * For variable-size function arguments, change the example sizes to */
/* the sizes that your application requires. */
/* * Change the example values of function arguments to the values that */
/* your application requires. */
/* * If the entry-point functions return values, store these values or */
/* otherwise use them as required by your application. */
/*
*****/

/* Include Files */
#include "main.h"
#include "coderand.h"
#include "coderand_terminate.h"

/* Function Declarations */
static void main_coderand(void);

/* Function Definitions */

/*
 * Arguments : void
 * Return Type : void
 */
static void main_coderand(void)
{
    double r;

    /* Call the entry-point 'coderand'. */
    r = coderand();
}

/*
 * Arguments : int argc
 *             const char * const argv[]
 * Return Type : int
 */

```

```
int main(int argc, const char * const argv[])
{
    (void)argc;
    (void)argv;

    /* The initialize function is being called automatically from your entry-point function. So, a call to initialize
    /* Invoke the entry-point functions.
    You can call entry-point functions multiple times. */
    main_coderand();

    /* Terminate the application.
    You do not need to do this more than one time. */
    coderand_terminate();
    return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */
```

## 2 修改 main.c, 使其打印 coderand 调用的结果:

- 在 main\_coderand 中, 删除以下行  
double r;
- 在 main\_coderand 中, 将  
r = coderand()  
替换为:  
printf("coderand=%g\n", coderand());
- 对于此示例, main 没有参数。在 main 中, 删除以下行:  
(void)argc;  
(void)argv;  
将 main 的定义更改为  
int main()

### 修改后的 main.c

```
/* Include Files */
#include "main.h"
#include "coderand.h"
#include "coderand_terminate.h"

/* Function Declarations */
static void main_coderand(void);

/* Function Definitions */

/*
 * Arguments : void
 * Return Type : void
 */
```

```

static void main_coderand(void)
{
    /* Call the entry-point 'coderand'. */
    printf("coderand=%g\n", coderand());
}

/*
 * Arguments  : int argc
 *              const char * const argv[]
 * Return Type : int
 */
int main()
{
    /* The initialize function is being called automatically from your entry-point function. So, a call to initialize
    /* Invoke the entry-point functions.
       You can call entry-point functions multiple times. */
    main_coderand();

    /* Terminate the application.
       You do not need to do this more than one time. */
    coderand_terminate();
    return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */

```

### 3 打开 main.h。

#### 生成的 main.h

```

/*****
/* This automatically generated example C main file shows how to call */
/* entry-point functions that MATLAB Coder generated. You must customize */
/* this file for your application. Do not modify this file directly. */
/* Instead, make a copy of this file, modify it, and integrate it into */
/* your development environment. */
/*
/* This file initializes entry-point function arguments to a default */
/* size and value before calling the entry-point functions. It does */
/* not store or use any values returned from the entry-point functions. */
/* If necessary, it does pre-allocate memory for returned values. */
/* You can use this file as a starting point for a main function that */
/* you can deploy in your application. */
/*
/* After you copy the file, and before you deploy it, you must make the */
/* following changes: */
/* * For variable-size function arguments, change the example sizes to */
/* the sizes that your application requires. */
/* * Change the example values of function arguments to the values that */
/* your application requires. */
/* * If the entry-point functions return values, store these values or */
/* otherwise use them as required by your application. */
/*
*****/
#endif MAIN_H

```

```

#define MAIN_H

/* Include Files */
#include <stddef.h>
#include <stdlib.h>
#include "rtwtypes.h"
#include "coderand_types.h"

/* Function Declarations */
extern int main(int argc, const char * const argv[]);

#endif

/*
 * File trailer for main.h
 *
 * [EOF]
 */

```

#### 4 修改 main.h:

- 将 `stdio` 添加到包含文件中:

```
#include <stdio.h>
```

- 将 `main` 的声明更改为

```
extern int main()
```

#### 修改后的 main.h

```

#ifndef MAIN_H
#define MAIN_H

/* Include Files */
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "rtwtypes.h"
#include "coderand_types.h"

/* Function Declarations */
extern int main();

#endif


/*
 * File trailer for main.h
 *
 * [EOF]
 */


```

#### 生成可执行文件

在修改生成的示例主文件后，打开之前创建的工程文件或从 App 中选择 `coderand.m`。您可以选择覆盖该工程，或以不同名称命名工程以同时保存这两个工程文件。

1

要打开**生成代码**页面，请展开工作流步骤 ，然后点击**生成**

- 2 要打开**生成**对话框，请点击**生成** 箭头 。
- 3 将**编译类型**设置为“可执行文件(.exe)”。
- 4 点击**更多设置**。
- 5 在**自定义代码**选项卡的**其他源文件**中，输入 `main.c`
- 6 在**自定义代码**选项卡的**其他包括目录**中，输入修改后的 `main.c` 和 `main.h` 文件的位置。例如，`c:\myfiles`。点击**关闭**。
- 7 要生成可执行文件，请点击**生成**。

App 指示代码生成成功。

- 8 点击**下一步**以转至**完成 workflow**步骤。
- 9 在**生成的输出**下，您可以看到生成的可执行文件 `coderand.exe` 的位置。

## 运行可执行文件

要在 Windows 平台上的 MATLAB 中运行可执行文件，请执行以下命令：

```
system('coderand')
```

---

**注意** 此示例中可执行的 `coderand` 函数返回介于 0 到 1 之间的固定值。要生成每次执行都会产生新值的代码，请在 MATLAB 代码中使用 `RandStream` 函数。

---

## 在命令行中生成 C 可执行文件

在此示例中，您将创建一个生成随机标量值的 MATLAB 函数和一个调用此 MATLAB 函数的 C 主函数。然后指定函数输入参数的类型，指定主函数，并为 MATLAB 代码生成 C 可执行文件。

- 1 编写一个 MATLAB 函数 `coderand`，该函数在开区间 (0,1) 上基于标准均匀分布生成一个随机标量值：

```
function r = coderand() %#codegen
r = rand();
```

- 2 编写一个 C 主函数 `c:\myfiles\main.c`，它调用 `coderand`。例如：

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderand.h"
#include "coderand_terminate.h"

int main()
{
    /* The initialize function is called automatically from the generated entry-point function.
       So, a call to initialize is not included here. */

    printf("coderand=%g\n", coderand());

    coderand_terminate();

    return 0;
}
```

**注意** 在此示例中，由于默认文件分区方法是每个 MATLAB 文件生成一个文件，因此您要包含 "coderand\_terminate.h"。如果您的文件分区方法设置为对所有函数只生成一个文件，请**不要**包含 "coderand\_terminate.h"。

- 3 将代码生成参数配置为包含 C 主函数，然后生成 C 可执行文件：

```
cfg = coder.config('exe');
cfg.CustomSource = 'main.c';
cfg.CustomInclude = 'c:\myfiles';
codegen -config cfg coderand
```

`codegen` 在当前文件夹中生成 C 可执行文件 `coderand.exe`。它在默认文件夹 `codegen/exe/` `coderand` 中生成支持文件。`codegen` 为所选代码替换库所需的头文件生成最小的 `#include` 语句集。

## 指定 C/C++ 可执行文件的主函数

生成可执行文件时，您必须提供 `main` 函数。对于 C 可执行文件，请提供 C 文件 `main.c`。对于 C++ 可执行文件，请提供 C++ 文件 `main.cpp`。确认包含主函数的文件夹中只有一个主文件。否则，`main.c` 将优先于 `main.cpp`，这会在生成 C++ 代码时导致错误。您可以从工程设置对话框、命令行或 Code Generation 对话框中指定主文件。


默认情况下，当您生成 C/C++ 源代码、静态库、动态链接库或可执行文件时，MATLAB Coder 会生成 `main` 函数。这一生成的主函数是一个模板，您可以针对您的应用程序修改该模板。请参阅“使用示例主函数合并生成的代码”（第 32-13 页）。在复制和修改生成的主函数后，可以使用它来生成 C/C++ 可执行文件。您也可以编写自己的主函数。

将 MATLAB 函数转换为 C/C++ 库函数或 C/C++ 可执行文件时，MATLAB Coder 会生成一个初始化函数和一个终止函数。

- 如果您的文件分区方法设置为对每个 MATLAB 文件生成一个文件，则您必须在 `main.c` 中包含终止头函数。否则，不要将其包含在 `main.c` 中。
- 有关调用初始化和终止函数的详细信息，请参阅“Use Generated Initialize and Terminate Functions”。

## 指定主函数

### 使用 MATLAB Coder App 指定主函数

- 1 要打开**生成**对话框，请在**生成代码**页上点击**生成**箭头 .
- 2 点击**更多设置**。
- 3 在**自定义代码**选项卡上，进行如下设置：
  - a 将**其他源文件**设置为包含 `main` 函数的 C/C++ 源文件的名称。例如，`main.c`。有关详细信息，请参阅“指定 C/C++ 可执行文件的主函数”（第 27-10 页）。
  - b 将**其他包括目录**设置为 `main.c` 的位置。例如，`c:\myfiles`。

### 在命令行中指定主函数

设置代码生成配置对象的 `CustomSource` 和 `CustomInclude` 属性（请参阅“使用配置对象”（第 27-17 页））。`CustomInclude` 属性指示由 `CustomSource` 指定的 C/C++ 文件的位置。

- 1 为可执行文件创建配置对象：

```
cfg = coder.config('exe');
```

- 2 将 **CustomSource** 属性设置为包含 **main** 函数的 C/C++ 源文件的名称。（有关详细信息，请参阅“指定 C/C++ 可执行文件的主函数”（第 27-10 页）。）例如：

```
cfg.CustomSource = 'main.c';
```

- 3 将 **CustomInclude** 属性设置为 **main.c** 的位置。例如：

```
cfg.CustomInclude = 'c:\myfiles';
```

- 4 使用命令行选项生成 C/C++ 可执行文件。例如，如果 **myFunction** 接受 **double** 类型的一个输入参数：

```
codegen -config cfg myMFunction -args {0}
```

MATLAB Coder 将对主函数进行编译并将它与从 **myMFunction.m** 生成的 C/C++ 代码进行链接。

配置编译设置

本节内容
“指定编译类型” (第 27-12 页)
“指定用于代码生成的语言” (第 27-14 页)
“指定输出文件名” (第 27-15 页)
“指定输出文件位置” (第 27-15 页)
“参数设定方法” (第 27-16 页)
“指定编译配置参数” (第 27-16 页)

指定编译类型

编译类型

MATLAB Coder 可以生成以下输出类型的代码：

- MEX 函数
- 独立 C/C++ 代码
- 独立 C/C++ 代码并将其编译为静态库
- 独立 C/C++ 代码并将其编译为动态链接库
- 独立 C/C++ 代码并将其编译为可执行文件


**注意** 生成可执行文件时，必须提供包含 **main** 函数的 C/C++ 文件，如“指定 C/C++ 可执行文件的主函数” (第 27-10 页) 中所述。

生成的文件的位置

默认情况下，MATLAB Coder 根据输出类型在输出文件夹中生成文件。有关详细信息，请参阅“Generated Files and Locations”。

**注意** 每次 MATLAB Coder 为相同的代码或工程生成相同类型的输出时，都会删除上一次编译生成的文件。如果要保留某次编译生成的文件，请在开始新的编译之前将这些文件复制到其他位置。

使用 MATLAB Coder App 指定编译类型

- 1 要打开 **Generate** 对话框，请在 **Generate Code** 页上点击 **Generate** 箭头 .
- 2 将 **Build type** 设置为以下项之一。
  - “Source Code”
  - “MEX”
  - “Static Library”
  - “Dynamic Library”
  - “Executable”



如果选择 “Source Code”，MATLAB Coder 将不会调用 make 命令，也不会生成编译的目标代码。如果您在修改 MATLAB 代码和生成 C/C++ 代码之间进行迭代并且要检查生成的代码，此选项可以节省时间。此选项等同于选中了 **Generate code only** 框的 “Static Library”。

代码生成对 MEX 函数和其他编译类型使用不同的配置参数集。当您在 “MEX Function” 和 “Source”、“Static Library”、“Dynamic Library” 或 “Executable” 之间切换输出类型时，请验证这些设置。

某些配置参数与 MEX 和独立代码生成相关。如果您在输出类型为 “MEX Function” 时启用这些参数中的任何参数，并且您要在 C/C++ 代码生成中也使用相同的设置，则必须为 “C/C++ Static Library”、“C/C++ Dynamic Library” 和 “C/C++ Executable” 再次启用它。

### 在命令行中指定编译类型

使用 `-config` 选项调用 `codegen`。例如，假设您有不需要输入参数的主函数 `foo`。下表显示在编译 `foo` 时如何指定不同输出类型。如果主函数有输入参数，则必须指定这些输入。有关详细信息，请参阅 “Specify Properties of Entry-Point Function Inputs”。

**注意** C 语言是使用 MATLAB Coder 生成代码的默认语言。要生成 C++ 代码，请参阅 “指定用于代码生成的语言”（第 27-14 页）。

要生成:	使用以下命令:
MEX 函数 (使用默认代码生成选项)	<code>codegen foo</code>
MEX 函数 (指定代码生成选项)	<pre> cfg = coder.config('mex'); % Set configuration parameters, for example, % enable a code generation report cfg.GenerateReport=true; % Call codegen, passing the configuration % object codegen -config cfg foo </pre>
独立 C/C++ 代码并将其编译为库 (使用默认代码生成选项)	<code>codegen -config:lib foo</code>
独立 C/C++ 代码并将其编译为库 (指定代码生成选项)	<pre> cfg = coder.config('lib'); % Set configuration parameters, for example, % enable a code generation report cfg.GenerateReport=true; % Call codegen, passing the configuration % object codegen -config cfg foo </pre>
独立 C/C++ 代码并将其编译为可执行文件 (使用默认代码生成选项并在命令行中指定 <code>main.c</code> 文件)	<pre> codegen -config:exe main.c foo </pre> <p><b>注意</b> 您必须指定 <code>main</code> 函数才能生成 C/C++ 可执行文件。请查阅 “指定 C/C++ 可执行文件的主函数”（第 27-10 页）</p>


要生成:	使用以下命令:
独立 C/C++ 代码并将其编译为可执行文件 (指定代码生成选项)	<pre>cfg = coder.config('exe'); % Set configuration parameters, for example, % specify main file cfg.CustomSource = 'main.c'; cfg.CustomInclude = 'c:\myfiles'; codegen -config cfg foo</pre> <p><b>注意</b> 您必须指定 <b>main</b> 函数才能生成 C/C++ 可执行文件。请查阅 “指定 C/C++ 可执行文件的主函数” (第 27-10 页)</p>

指定用于代码生成的语言

- “使用 MATLAB Coder App 指定语言” (第 27-14 页)
- “使用命令行界面指定语言” (第 27-14 页)

MATLAB Coder 可以生成 C 或 C++ 库和可执行文件。C 是默认语言。您可以从工程设置对话框或命令行显式指定语言。

使用 MATLAB Coder App 指定语言

- 1 要打开 **Generate** 对话框，请在 **Generate Code** 页上点击 **Generate** 箭头 。
- 2 将 **Language** 设置为 “C” 或 “C++”。

**注意** 如果您指定 C++，MATLAB Coder 会将 C 代码包装到 .cpp 文件中。您可以使用 C++ 编译器并与外部 C++ 应用程序对接。MATLAB Coder 不生成 C++ 类。

使用命令行界面指定语言

- 1 为您的目标语言选择合适的编译器。
- 2 创建用于代码生成的配置对象。例如，对于库，请使用：

```
cfg = coder.config('lib');
```

- 3 将 **TargetLang** 属性设置为 'C' 或 'C++'。例如：

```
cfg.TargetLang = 'C++';
```

**注意** 如果您指定 C++，MATLAB Coder 会将 C 代码封装到 .cpp 文件中。然后，您可以使用 C++ 编译器并与外部 C++ 应用程序对接。MATLAB Coder 不生成 C++ 类。

另请参阅

- “使用配置对象” (第 27-17 页)
- “设置 C 或 C++ 编译器”

## 指定输出文件名

### 使用 MATLAB Coder App 指定输出文件名

- 1 要打开 **Generate** 对话框，请在 **Generate Code** 页上点击 **Generate** 箭头 .
- 2 在 **Output file name** 字段中，输入文件名。

---

**注意** 文件名中不能包含空格。

---

默认情况下，如果第一个入口 MATLAB 文件的名称为 **fcn1**，则输出文件名为：

- **fcn1**（对于 C/C++ 库和可执行文件）。
- **fcn1\_mex**（对于 MEX 函数）。

默认情况下，MATLAB Coder 在文件夹 **project\_folder/codegen/target/fcn1** 中生成文件：

- **project\_folder** 是您的当前工程文件夹
- **target** 是：
  - **mex**（对于 MEX 函数）
  - **lib**（对于静态 C/C++ 库）
  - **dll**（对于动态 C/C++ 库）
  - **exe**（对于 C/C++ 可执行文件）

### 命令行替代方法


使用 **codegen** 函数和 **-o** 选项。

## 指定输出文件位置

### 使用 MATLAB Coder App 指定输出文件位置

输出文件位置不能包含：

- 空格（在某些操作系统配置中，空格可能导致代码生成失败）。
- 制表符
- \、\$、#、\*、？
- 非 7 位 ASCII 字符，如日文字符。

- 1 要打开 **Generate** 对话框，请在 **Generate Code** 页上点击 **Generate** 箭头 .
- 2 将 **Build type** 设置为 “Source Code”、“Static Library”、“Dynamic Library” 或 “Executable”（取决于您的具体要求）。
- 3 点击 **More settings**。
- 4 点击 **Paths** 选项卡。

**Build folder** 字段的默认设置为 “A subfolder of the project folder”。默认情况下，MATLAB Coder 在文件夹 **project\_folder/codegen/target/fcn1** 中生成文件：

- `fcn1` 是按字母顺序排列的第一个入口文件的名称。
  - `target` 是：
    - `mex` (对于 MEX 函数)
    - `lib` (对于静态 C/C++ 库)
    - `dll` (对于动态链接的 C/C++ 库)
    - `exe` (对于 C/C++ 可执行文件)
- 5 要更改输出位置，您可以使用下列方法之一：
- 将 **Build Folder** 设置为 “A subfolder of the current MATLAB working folder”  
  
MATLAB Coder 将在 `MATLAB_working_folder/codegen/target/fcn1` 文件夹中生成文件
  - 将 **Build Folder** 设置为 “Specified folder” 。在 **Build folder name** 字段中，提供文件夹的路径。

命令行替代方法

使用 `codegen` 函数和 `-d` 选项。

参数设定方法


如果您使用	使用	详细信息
MATLAB Coder App	工程设置对话框。	“指定编译配置参数 MATLAB Coder App” (第 27-16 页)
<code>codegen</code> (在命令行中) 并希望指定一些参数	配置对象	“使用配置对象在命令行中指定编译配置参数” (第 27-17 页)
<code>codegen</code> (在编译脚本中)		
<code>codegen</code> (在命令行中) 并希望指定许多参数	配置对象对话框	“使用对话框在命令行中指定编译配置参数” (第 27-19 页)

指定编译配置参数

- “指定编译配置参数 MATLAB Coder App” (第 27-16 页)
- “使用配置对象在命令行中指定编译配置参数” (第 27-17 页)
- “使用对话框在命令行中指定编译配置参数” (第 27-19 页)

您可以从 MATLAB Coder 工程设置对话框、命令行或配置对象对话框指定编译配置参数。

指定编译配置参数 MATLAB Coder App

- 1 要打开 **Generate** 对话框，请在 **Generate Code** 页上点击 **Generate** 箭头 。
- 2 将 **Build type** 设置为 “Source Code” 、 “Static Library” 、 “Dynamic Library” 或 “Executable” (取决于您的具体要求) 。
- 3 点击 **More settings**。

工程设置对话框提供适用于您选择的输出类型的配置参数集。代码生成对 MEX 函数和其他编译类型使用不同的配置参数集。当您在“MEX Function”和“Source Code”、“Static Library”、“Dynamic Library”或“Executable”之间切换输出类型时，请验证这些设置。

某些配置参数与 MEX 和独立代码生成相关。如果您在输出类型为“MEX Function”时启用这些参数中的任何参数，并且您要在 C/C++ 代码生成中也使用相同的设置，则必须为“C/C++ Static Library”、“C/C++ Dynamic Library”和“C/C++ Executable”再次启用它。

- 4 根据需要修改参数。有关选项卡上的参数的详细信息，请点击 **Help**。

对参数设置的更改会立即生效。

## 使用配置对象在命令行中指定编译配置参数

### 配置对象的类型

`codegen` 函数使用配置对象来自定义您的代码生成环境。下表列出了可用的配置对象。

配置对象	说明
<code>coder.CodeConfig</code>	如果没有可用的 Embedded Code 许可证或禁用了 Embedded Code 许可证，请指定 C/C++ 库或可执行文件生成的参数。 有关详细信息，请参阅 <code>coder.CodeConfig</code> 的类参考信息。
<code>coder.EmbeddedCodeConfig</code>	如果 Embedded Code 许可证可用，请指定 C/C++ 库或可执行文件生成的参数。 有关详细信息，请参阅 <code>coder.EmbeddedCodeConfig</code> 的类参考信息。
<code>coder.HardwareImplementation</code>	指定目标硬件实现的参数。如果未指定， <code>codegen</code> 会生成与 MATLAB 主机兼容的代码。 有关详细信息，请参阅 <code>coder.HardwareImplementation</code> 的类参考信息。
<code>coder.MexCodeConfig</code>	指定 MEX 代码生成的参数。 有关详细信息，请参阅 <code>coder.MexCodeConfig</code> 的类参考信息。

### 使用配置对象

要使用配置对象来自定义代码生成的环境，请执行以下操作：

- 1 在 MATLAB 工作区中，定义配置对象变量，如“创建配置对象”（第 27-18 页）中所述。

例如，要生成用于 C 静态库生成的配置对象，请执行以下操作：

```
cfg = coder.config('lib');
% Returns a coder.CodeConfig object if no
% Embedded Code license available.
% Otherwise, returns a coder.EmbeddedCodeConfig object.
```

- 2 根据需要使用以下方法之一修改配置对象的参数：

- 交互式命令，如“使用配置对象在命令行中指定编译配置参数”（第 27-17 页）中所述
- 对话框，如“使用对话框在命令行中指定编译配置参数”（第 27-19 页）中所述

3 使用 `-config` 选项调用 `codegen` 函数。将配置对象指定为其参数。

`-config` 选项会根据配置属性值，指示 `codegen` 为目标生成代码。在以下示例中，`codegen` 基于在第一步中定义的代码生成配置对象 `cfg` 的参数，从 MATLAB 函数 `foo` 生成 C 静态库：

```
codegen -config cfg foo
```

`-config` 选项指定要编译的输出的类型 - 在本例中为 C 静态库。有关详细信息，请参阅 `codegen`。

创建配置对象

您可以在 MATLAB 工作区中定义配置对象。

要创建...	使用如下命令...
MEX 配置对象 <code>coder.MexCodeConfig</code>	<code>cfg = coder.config('mex');</code>
用于生成独立 C/C++ 库或可执行文件的代码生成配置对象 <code>coder.CodeConfig</code>	<pre>% To generate a static library cfg = coder.config('lib'); % To generate a dynamic library cfg = coder.config('dll') % To generate an executable cfg = coder.config('exe');</pre> <div><p><b>注意</b> 如果 Embedded Coder 许可证可用，请创建一个 <code>coder.EmbeddedCodeConfig</code> 对象。</p><p>如果使用并发许可证，要禁用 Embedded Coder 许可证的签出，请使用以下命令之一：</p><pre>cfg = coder.config('lib', 'ecoder', false)  cfg = coder.config('dll', 'ecoder', false)  cfg = coder.config('exe', 'ecoder', false)</pre></div>
用于为嵌入式目标生成独立 C/C++ 库或可执行文件的代码生成配置对象 <code>coder.EmbeddedCodeConfig</code>	<pre>% To generate a static library cfg = coder.config('lib'); % To generate a dynamic library cfg = coder.config('dll') % To generate an executable cfg = coder.config('exe');</pre> <div><p><b>注意</b> 需要提供 Embedded Coder 许可证；否则请创建 <code>coder.CodeConfig</code> 对象。</p></div>
硬件实现配置对象 <code>coder.HardwareImplementation</code>	<code>hwcfg = coder.HardwareImplementation</code>

每个配置对象都附带一组参数，初始化为默认值。您可以更改这些设置，如“使用圆点表示法在命令行中修改配置对象”（第 27-18 页）中所述。

使用圆点表示法在命令行中修改配置对象

您可以使用圆点表示法一次修改一个配置对象参数的值。请使用以下语法：

```
configuration_object.property = value
```

圆点表示法使用赋值语句来修改配置对象属性：

- 要在 C/C++ 代码生成期间指定 `main` 函数，请执行以下操作：

```
cfg = coder.config('exe');
cfg.CustomInclude = 'c:\myfiles';
cfg.CustomSource = 'main.c';
codegen -config cfg foo
```

- 要在生成 C/C++ 静态库后自动生成并启动代码生成报告，请执行以下操作：

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.LaunchReport = true;
codegen -config cfg foo
```

### 保存配置对象

配置对象不会在 MATLAB 会话之间自动保留。请使用以下方法之一保留您的设置：

#### 将配置对象保存到 MAT 文件，然后在下一个会话中加载该 MAT 文件

例如，假设您在 MATLAB 工作区中创建和自定义了一个 MEX 配置对象 `mexcfg`。要保存该配置对象，请在 MATLAB 提示符下输入：

```
save mexcfg.mat mexcfg
```

`save` 命令将 `mexcfg` 保存到当前文件夹中的文件 `mexcfg.mat`。

要在新 MATLAB 会话中恢复 `mexcfg`，请在 MATLAB 提示符下输入：

```
load mexcfg.mat
```

`load` 命令将在 `mexcfg.mat` 中定义的对象加载到 MATLAB 工作区。

#### 编写用于创建配置对象并设置其属性的脚本。

无论您何时需要再次使用该配置对象，只需重新运行该脚本即可。

#### 使用对话框在命令行中指定编译配置参数

在创建配置对象后，可以使用配置参数对话框修改对象的属性。请参阅“Specify Configuration Parameters in Command-Line Workflow Interactively”。

## 在 macOS 平台上安装 OpenMP 库

您可以通过在 MATLAB 代码中使用 `parfor` 来在 macOS 平台上生成并行 `for` 循环。代码生成器使用 OpenMP (Open Multiprocessing) 应用程序接口来支持共享内存和多核代码生成。要在 MATLAB 之外运行 `parfor` 循环生成的代码，必须安装 OpenMP 库。

要在 macOS 平台上安装 OpenMP 库 `libomp`，请执行以下操作之一：

- 从 LLVM 下载页安装 `libomp`。
  - 1 导航到 LLVM 下载页。
  - 2 下载 OpenMP 源代码。
  - 3 编译源代码并安装。
- 使用 `homebrew` 安装 `libomp`。在终端上，运行以下命令。

```
brew install libomp
```

### 另请参阅

`parfor`

### 详细信息

- “使用并行 `for` 循环 (`parfor`) 生成代码” (第 35-5 页)

### 外部网站

- <https://releases.llvm.org/>
- <https://brew.sh/>



## 生成代码以检测图像边缘

此示例说明如何从 MATLAB® 代码生成独立 C 库，该库实现一个简单的 Sobel 滤波器，该滤波器对图像执行边缘检测。示例还说明如何在生成 C 代码之前在 MATLAB 中生成和测试 MEX 函数，以验证该 MATLAB 代码适合进行代码生成。

### 关于 sobel 函数

`sobel.m` 函数接受一个图像（表示为双精度矩阵）和一个阈值，并返回一个检测到边缘的图像（基于阈值）。

#### type `sobel`

```
% edgeImage = sobel(originalImage, threshold)
% Sobel edge detection. Given a normalized image (with double values)
% return an image where the edges are detected w.r.t. threshold value.
function edgeImage = sobel(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = [1 2 1; 0 0 0; -1 -2 -1];
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k', 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

### 生成 MEX 函数

使用 `codegen` 命令生成 MEX 函数。

#### codegen `sobel`

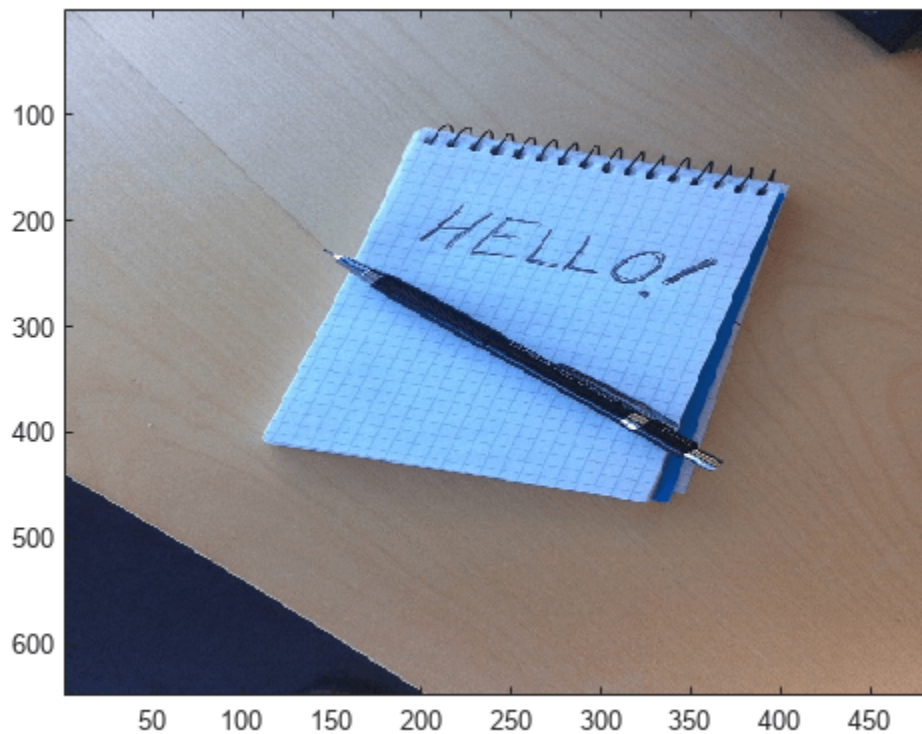
Code generation successful.

在生成 C 代码之前，应首先在 MATLAB 中测试 MEX 函数，以确保它在功能上等同于原始 MATLAB 代码，并且不会出现任何运行时错误。默认情况下，`codegen` 在当前文件夹中生成名为 `sobel_mex` 的 MEX 函数。这允许您测试 MATLAB 代码和 MEX 函数，并将结果进行比较。

### 读取原始图像

使用标准 `imread` 命令。

```
im = imread('hello.jpg');
image(im);
```



### 将图像转换为灰度版本

使用归一化值（0.0 代表黑色、1.0 代表白色），将彩色图像（如上所示）转换为等效的灰度图像。

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
```

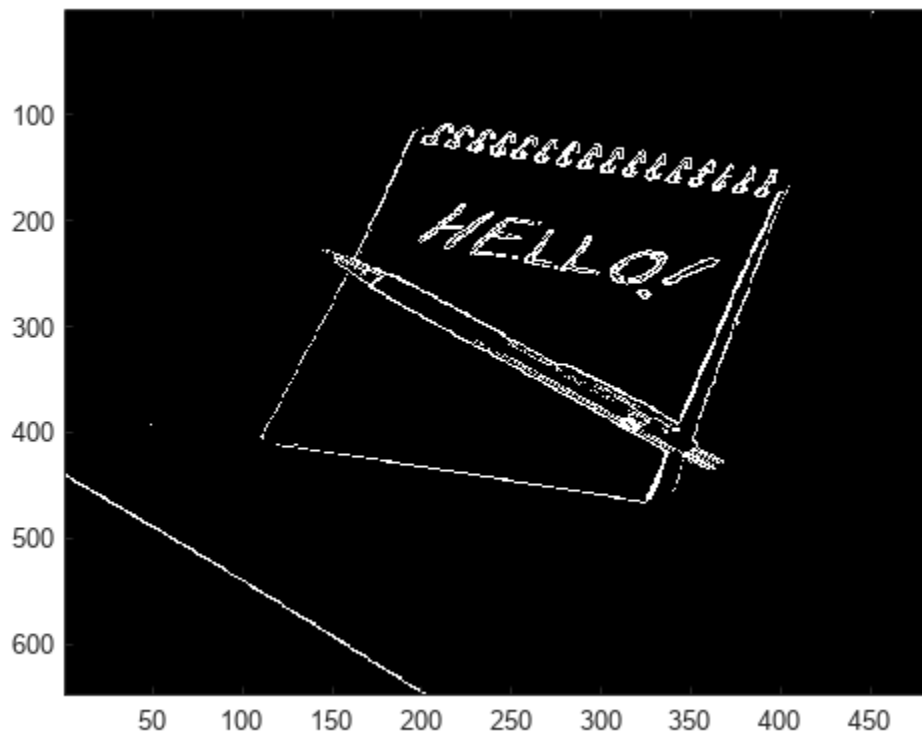
### 运行 MEX 函数 (Sobel 滤波器)

传递归一化图像和阈值。

```
edgeIm = sobel_mex(gray, 0.7);
```

### 显示结果

```
im3 = repmat(edgeIm, [1 1 3]);  
image(im3);
```



生成独立 C 代码。

```
codegen -config coder.config('lib') sobel
```

Code generation successful.

将 `codegen` 与 `-config coder.config('lib')` 选项结合使用生成独立 C 库。默认情况下，为库生成的代码位于文件夹 `codegen/lib/sobel/` 中。

检查生成的函数

```
type codegen/lib/sobel/sobel.c
```

```
/*
 * Prerelease License - for engineering feedback and testing purposes
 * only. Not for sale.
 * File: sobel.c
 *
 * MATLAB Coder version      : 5.6
 * C/C++ source code generated on : 30-Jan-2023 12:50:01
 */

/* Include Files */
#include "sobel.h"
#include "conv2AXPYSameCMP.h"
#include "sobel_data.h"
#include "sobel_emxutil.h"
#include "sobel_initialize.h"
```

```
#include "sobel_types.h"
#include <math.h>

/* Function Declarations */
static void binary_expand_op(emxArray_real_T *in1, const emxArray_real_T *in2);

/* Function Definitions */
/*
 * Arguments   : emxArray_real_T *in1
 *               const emxArray_real_T *in2
 * Return Type : void
 */
static void binary_expand_op(emxArray_real_T *in1, const emxArray_real_T *in2)
{
    emxArray_real_T *b_in1;
    const double *in2_data;
    double *b_in1_data;
    double *in1_data;
    int aux_0_1;
    int aux_1_1;
    int b_loop_ub;
    int i;
    int i1;
    int loop_ub;
    int stride_0_0;
    int stride_0_1;
    int stride_1_0;
    int stride_1_1;
    in2_data = in2->data;
    in1_data = in1->data;
    emxInit_real_T(&b_in1, 2);
    if (in2->size[0] == 1) {
        loop_ub = in1->size[0];
    } else {
        loop_ub = in2->size[0];
    }
    i = b_in1->size[0] * b_in1->size[1];
    b_in1->size[0] = loop_ub;
    if (in2->size[1] == 1) {
        b_loop_ub = in1->size[1];
    } else {
        b_loop_ub = in2->size[1];
    }
    b_in1->size[1] = b_loop_ub;
    emxEnsureCapacity_real_T(b_in1, i);
    b_in1_data = b_in1->data;
    stride_0_0 = (in1->size[0] != 1);
    stride_0_1 = (in1->size[1] != 1);
    stride_1_0 = (in2->size[0] != 1);
    stride_1_1 = (in2->size[1] != 1);
    aux_0_1 = 0;
    aux_1_1 = 0;
    for (i = 0; i < b_loop_ub; i++) {
        for (i1 = 0; i1 < loop_ub; i1++) {
            double b_in1_tmp;
            double in1_tmp;
            in1_tmp = in1_data[i1 * stride_0_0 + in1->size[0] * aux_0_1];
            b_in1_tmp = in2_data[i1 * stride_1_0 + in2->size[0] * aux_1_1];
```

```

        b_in1_data[i1 + b_in1->size[0] * i] =
            in1_tmp * in1_tmp + b_in1_tmp * b_in1_tmp;
    }
    aux_1_1 += stride_1_1;
    aux_0_1 += stride_0_1;
}
i = in1->size[0] * in1->size[1];
in1->size[0] = b_in1->size[0];
in1->size[1] = b_in1->size[1];
emxEnsureCapacity_real_T(in1, i);
in1_data = in1->data;
loop_ub = b_in1->size[1];
for (i = 0; i < loop_ub; i++) {
    b_loop_ub = b_in1->size[0];
    for (i1 = 0; i1 < b_loop_ub; i1++) {
        in1_data[i1 + in1->size[0] * i] = b_in1_data[i1 + b_in1->size[0] * i];
    }
}
emxFree_real_T(&b_in1);
}

/*
 * Arguments   : const emxArray_real_T *originalImage
 *               double threshold
 *               emxArray_uint8_T *edgeImage
 * Return Type : void
 */
void sobel(const emxArray_real_T *originalImage, double threshold,
           emxArray_uint8_T *edgeImage)
{
    emxArray_real_T *H;
    emxArray_real_T *V;
    double *H_data;
    double *V_data;
    int k;
    int loop_ub;
    unsigned char *edgeImage_data;
    if (!isInitialized_sobel) {
        sobel_initialize();
    }
    /* edgeImage = sobel(originalImage, threshold) */
    /* Sobel edge detection. Given a normalized image (with double values) */
    /* return an image where the edges are detected w.r.t. threshold value. */
    emxInit_real_T(&H, 2);
    conv2AXPYSameCMP(originalImage, H);
    H_data = H->data;
    emxInit_real_T(&V, 2);
    b_conv2AXPYSameCMP(originalImage, V);
    V_data = V->data;
    if ((H->size[0] == V->size[0]) && (H->size[1] == V->size[1])) {
        loop_ub = H->size[0] * H->size[1];
        for (k = 0; k < loop_ub; k++) {
            H_data[k] = H_data[k] * H_data[k] + V_data[k] * V_data[k];
        }
    } else {
        binary_expand_op(H, V);
        H_data = H->data;
    }
}

```

```
    emxFree_real_T(&V);
    loop_ub = H->size[0] * H->size[1];
    for (k = 0; k < loop_ub; k++) {
        H_data[k] = sqrt(H_data[k]);
    }
    k = edgeImage->size[0] * edgeImage->size[1];
    edgeImage->size[0] = H->size[0];
    edgeImage->size[1] = H->size[1];
    emxEnsureCapacity_uint8_T(edgeImage, k);
    edgeImage_data = edgeImage->data;
    for (k = 0; k < loop_ub; k++) {
        edgeImage_data[k] =
            (unsigned char)((unsigned int)(H_data[k] > threshold) * 255U);
    }
    emxFree_real_T(&H);
}

/*
 * File trailer for sobel.c
 *
 * [EOF]
 */
```

## 为 MATLAB 卡尔曼滤波算法生成 C 代码

此示例说明如何为 MATLAB® 卡尔曼滤波器函数 `kalmanfilter` 生成 C 代码，该函数基于过去的含噪测量值来估计运动物体的位置。示例还说明如何为此 MATLAB 代码生成 MEX 函数，以提高算法在 MATLAB 中的执行速度。

### 前提条件

此示例没有任何前提条件。

### 关于 `kalmanfilter` 函数

`kalmanfilter` 函数根据运动物体过去的位置值预测其位置。该函数使用一个卡尔曼滤波器估计器，这是一种递归自适应滤波器，它根据一系列含噪测量值估计动态系统的状态。卡尔曼滤波在信号和图像处理、控制设计和计算金融等领域有着广泛的应用。

### 关于卡尔曼滤波器估计器算法

卡尔曼估计器通过计算和更新卡尔曼状态向量来计算位置向量。状态向量定义为  $6 \times 1$  列向量，其中包括二维笛卡尔空间中的位置 ( $x$  和  $y$ )、速度 ( $V_x$   $V_y$ ) 和加速度 ( $A_x$  和  $A_y$ ) 测量值。基于经典的运动定律：

$$\begin{cases} X = X_0 + V_x dt \\ Y = Y_0 + V_y dt \\ V_x = V_{x0} + A_x dt \\ V_y = V_{y0} + A_y dt \end{cases}$$

捕获这些定律的迭代公式反映在卡尔曼状态转移矩阵 “A” 中。请注意，通过编写大约 10 行的 MATLAB 代码，您可以基于在许多自适应滤波教科书中找到的理论数学公式来实现卡尔曼估计器。

### type `kalmanfilter.m`

```
% Copyright 2010 The MathWorks, Inc.
function y = kalmanfilter(z)
%#codegen
dt=1;
% Initialize state transition matrix
A=[ 1 0 dt 0 0 0;... % [x ]
    0 1 0 dt 0 0;... % [y ]
    0 0 1 0 dt 0;... % [Vx]
    0 0 0 1 0 dt;... % [Vy]
    0 0 0 0 1 0;... % [Ax]
    0 0 0 0 0 1]; % [Ay]
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ]; % Initialize measurement matrix
Q = eye(6);
R = 1000 * eye(2);
persistent x_est p_est % Initial state conditions
if isempty(x_est)
    x_est = zeros(6, 1); % x_est=[x,y,Vx,Vy,Ax,Ay]'
    p_est = zeros(6, 6);
end
% Predicted state and covariance
x_prd = A * x_est;
p_prd = A * p_est * A' + Q;
% Estimation
```

```
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';
% Estimated state and covariance
x_est = x_prd + klm_gain * (z - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;
% Compute the estimated measurements
y = H * x_est;
end % of the function
```

### 加载测试数据

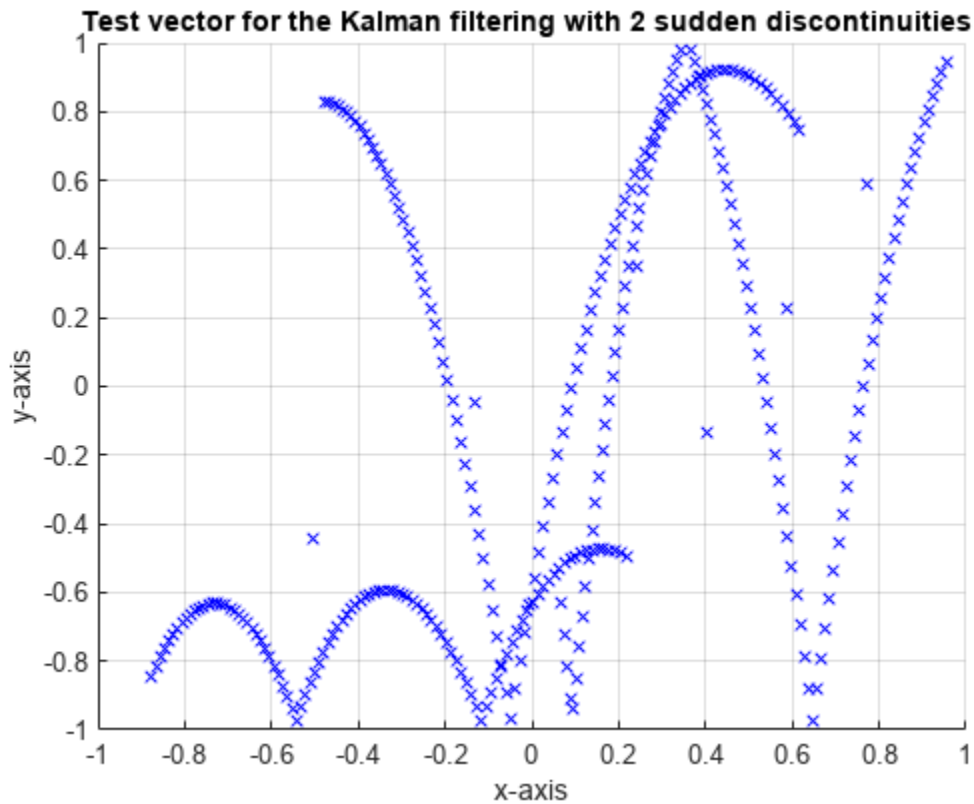
要跟踪的物体的位置记录为笛卡尔空间中的  $x$  和  $y$  坐标，存储在名为 **position\_data.mat** 的 MAT 文件中。以下代码会加载该 MAT 文件并绘制位置的轨迹。测试数据包括位置上的两次突然改变或不连续，用于检查卡尔曼滤波器能否快速重新调整和跟踪物体。

```
load position_data.mat
hold; grid;
```

Current plot held

```
for idx = 1: numPts
z = position(:,idx);
plot(z(1), z(2), 'bx');
axis([-1 1 -1 1]);
end
title('Test vector for the Kalman filtering with 2 sudden discontinuities ');
xlabel('x-axis');ylabel('y-axis');
hold;
```





Current plot released

### 检查并运行 ObjTrack 函数

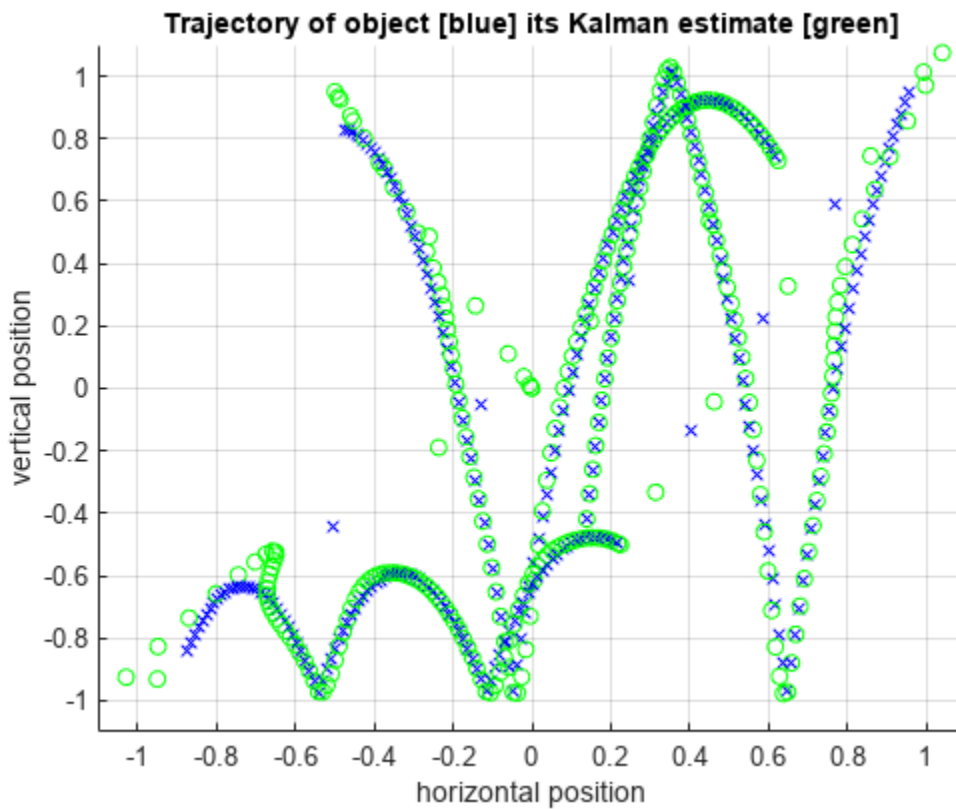
**ObjTrack.m** 函数调用卡尔曼滤波器算法，用蓝色绘制物体轨迹，用绿色绘制卡尔曼滤波器估算位置。最初，您会看到估计的位置只用很短时间就与物体的实际位置重合在一起。然后，位置会出现三次突然改变。每次卡尔曼滤波器都会重新调整并在经过几次迭代后跟踪上该物体的实际位置。

type **ObjTrack**

```
% Copyright 2010 The MathWorks, Inc.
function ObjTrack(position)
%#codegen
% First, setup the figure
numPts = 300;          % Process and plot 300 samples
figure;hold;grid;      % Prepare plot window
% Main loop
for idx = 1:numPts
    z = position(:,idx); % Get the input data
    y = kalmanfilter(z);  % Call Kalman filter to estimate the position
    plot_trajectory(z,y); % Plot the results
end
hold;
end % of the function
```

**ObjTrack(position)**

Current plot held



Current plot released

### 生成 C 代码

带 `-config:lib` 选项的 `codegen` 命令可生成打包为独立 C 库的 C 代码。

由于 C 使用静态类型，`codegen` 必须在编译时确定 MATLAB 文件中所有变量的属性。在这里，`-args` 命令行选项会提供一个示例输入，以便 `codegen` 基于输入类型推断新类型。

`-report` 选项生成一个编译报告，其中包含编译结果的汇总和所生成文件的链接。编译 MATLAB 代码后，`codegen` 提供指向该报告的超链接。

```
z = position(:,1);
codegen -config:lib -report -c kalmanfilter.m -args {z}
```

Code generation successful: To view the report, open('codegen/lib/kalmanfilter/html/report.mldatx')

### 检查生成的代码

生成的 C 代码位于 `codegen/lib/kalmanfilter/` 文件夹中。这些文件是：

```
dir codegen/lib/kalmanfilter/
```

```
.          ..          _clang-format          buildInfo.mat          codeInfo.mat          codedescriptor.d
```

### 检查 kalmanfilter.c 函数的 C 代码

```
type codegen/lib/kalmanfilter/kalmanfilter.c
```

```

/*
 * Prerelease License - for engineering feedback and testing purposes
 * only. Not for sale.
 * File: kalmanfilter.c
 *
 * MATLAB Coder version      : 5.6
 * C/C++ source code generated on : 30-Jan-2023 12:49:07
 */

/* Include Files */
#include "kalmanfilter.h"
#include "kalmanfilter_data.h"
#include "kalmanfilter_initialize.h"
#include <math.h>
#include <string.h>

/* Variable Definitions */
static double x_est[6];

static double p_est[36];

/* Function Definitions */
/*
 * Arguments   : const double z[2]
 *               double y[2]
 * Return Type : void
 */
void kalmanfilter(const double z[2], double y[2])
{
    static const short R[4] = {1000, 0, 0, 1000};
    static const signed char b_a[36] = {1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                         1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0,
                                         0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1};
    static const signed char iv[36] = {1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0,
                                         0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
                                         0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1};
    static const signed char c_a[12] = {1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};
    static const signed char iv1[12] = {1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0};
    double a[36];
    double p_prd[36];
    double B[12];
    double Y[12];
    double x_prd[6];
    double S[4];
    double b_z[2];
    double a21;
    double a22;
    double a22_tmp;
    double d;
    int i;
    int k;
    int r1;
    int r2;
    signed char Q[36];
    if (!isInitialized_kalmanfilter) {
        kalmanfilter_initialize();
    }
    /* Copyright 2010 The MathWorks, Inc. */

```

```

/* Initialize state transition matrix */
/* % [x ] */
/* % [y ] */
/* % [Vx] */
/* % [Vy] */
/* % [Ax] */
/* [Ay] */
/* Initialize measurement matrix */
for (i = 0; i < 36; i++) {
    Q[i] = 0;
}
/* Initial state conditions */
/* Predicted state and covariance */
for (k = 0; k < 6; k++) {
    Q[k + 6 * k] = 1;
    x_prd[k] = 0.0;
    for (i = 0; i < 6; i++) {
        r1 = k + 6 * i;
        x_prd[k] += (double)b_a[r1] * x_est[i];
        d = 0.0;
        for (r2 = 0; r2 < 6; r2++) {
            d += (double)b_a[k + 6 * r2] * p_est[r2 + 6 * i];
        }
        a[r1] = d;
    }
}
for (i = 0; i < 6; i++) {
    for (r2 = 0; r2 < 6; r2++) {
        d = 0.0;
        for (r1 = 0; r1 < 6; r1++) {
            d += a[i + 6 * r1] * (double)iv[r1 + 6 * r2];
        }
        r1 = i + 6 * r2;
        p_prd[r1] = d + (double)Q[r1];
    }
}
/* Estimation */
for (i = 0; i < 2; i++) {
    for (r2 = 0; r2 < 6; r2++) {
        d = 0.0;
        for (r1 = 0; r1 < 6; r1++) {
            d += (double)c_a[i + (r1 << 1)] * p_prd[r2 + 6 * r1];
        }
        B[i + (r2 << 1)] = d;
    }
    for (r2 = 0; r2 < 2; r2++) {
        d = 0.0;
        for (r1 = 0; r1 < 6; r1++) {
            d += B[i + (r1 << 1)] * (double)iv1[r1 + 6 * r2];
        }
        r1 = i + (r2 << 1);
        S[r1] = d + (double)R[r1];
    }
}
if (fabs(S[1]) > fabs(S[0])) {
    r1 = 1;
    r2 = 0;
} else {

```

```

    r1 = 0;
    r2 = 1;
}
a21 = S[r2] / S[r1];
a22_tmp = S[r1 + 2];
a22 = S[r2 + 2] - a21 * a22_tmp;
for (k = 0; k < 6; k++) {
    double d1;
    i = k << 1;
    d = B[r1 + i];
    d1 = (B[r2 + i] - d * a21) / a22;
    Y[i + 1] = d1;
    Y[i] = (d - d1 * a22_tmp) / S[r1];
}
for (i = 0; i < 2; i++) {
    for (r2 = 0; r2 < 6; r2++) {
        B[r2 + 6 * i] = Y[i + (r2 << 1)];
    }
}
/* Estimated state and covariance */
for (i = 0; i < 2; i++) {
    d = 0.0;
    for (r2 = 0; r2 < 6; r2++) {
        d += (double)c_a[i + (r2 << 1)] * x_prd[r2];
    }
    b_z[i] = z[i] - d;
}
for (i = 0; i < 6; i++) {
    d = B[i + 6];
    x_est[i] = x_prd[i] + (B[i] * b_z[0] + d * b_z[1]);
    for (r2 = 0; r2 < 6; r2++) {
        r1 = r2 << 1;
        a[i + 6 * r2] = B[i] * (double)c_a[r1] + d * (double)c_a[r1 + 1];
    }
    for (r2 = 0; r2 < 6; r2++) {
        d = 0.0;
        for (r1 = 0; r1 < 6; r1++) {
            d += a[i + 6 * r1] * p_prd[r1 + 6 * r2];
        }
        r1 = i + 6 * r2;
        p_est[r1] = p_prd[r1] - d;
    }
}
/* Compute the estimated measurements */
for (i = 0; i < 2; i++) {
    d = 0.0;
    for (r2 = 0; r2 < 6; r2++) {
        d += (double)c_a[i + (r2 << 1)] * x_est[r2];
    }
    y[i] = d;
}
}

/*
* Arguments   : void
* Return Type : void
*/
void kalmanfilter_init(void)

```

```

{
    int i;
    for (i = 0; i < 6; i++) {
        x_est[i] = 0.0;
    }
    /* x_est=[x,y,Vx,Vy,Ax,Ay]' */
    memset(&p_est[0], 0, 36U * sizeof(double));
}

/*
 * File trailer for kalmanfilter.c
 *
 * [EOF]
 */

```

### 加快 MATLAB 算法的执行速度

通过使用 `codegen` 命令从 MATLAB 代码生成 MEX 函数，您可以加快处理大型数据集的 `kalmanfilter` 函数的执行速度。

#### 调用 `kalman_loop` 函数来处理大型数据集

首先，在 MATLAB 中使用大量数据样本运行卡尔曼算法。`kalman_loop` 函数循环运行 `kalmanfilter` 函数。循环迭代次数等于函数输入的第二个维度。

#### type `kalman_loop`

```

% Copyright 2010 The MathWorks, Inc.
function y=kalman_loop(z)
% Call Kalman estimator in the loop for large data set testing
%#codegen
[DIM, LEN]=size(z);
y=zeros(DIM,LEN);      % Initialize output
for n=1:LEN             % Output in the loop
    y(:,n)=kalmanfilter(z(:,n));
end;

```

### 不进行编译时的基线执行速度

现在对 MATLAB 算法进行计时。使用 `randn` 命令生成随机数，并创建由 100000 个 (2×1) 位置向量样本组成的输入矩阵 `position`。从当前文件夹中删除所有 MEX 文件。运行 `kalman_loop` 函数时，使用 MATLAB 秒表计时器 (`tic` 和 `toc` 命令) 来测量处理这些样本所需的时间。

```

clear mex
delete(['*.mexext'])
position = randn(2,100000);
tic, kalman_loop(position); a=toc;

```

### 生成 MEX 函数用于测试

下一步，使用命令 `codegen` 后跟要编译的 MATLAB 函数的名称 `kalman_loop` 生成 MEX 函数。`codegen` 命令生成名为 `kalman_loop_mex` 的 MEX 函数。然后，您可以将此 MEX 函数的执行速度与原始 MATLAB 算法的执行速度进行比较。

```
codegen -args {position} kalman_loop.m
```

```
Code generation successful.
```

```
which kalman_loop_mex
```

```
/tmp/Bdoc23a_2181783_498783/tp07414c0d/coder-ex53054096/kalman_loop_mex.mexa64
```

### 对 MEX 函数进行计时

现在，对 MEX 函数 `kalman_loop_mex` 进行计时。使用与之前相同的信号 `position` 作为输入，以确保公平地比较执行速度。

```
tic, kalman_loop_mex(position); b=toc;
```

### 比较执行速度

请注意使用生成的 MEX 函数时的执行速度差异。

```
display(sprintf('The speedup is %.1f times using the generated MEX over the baseline MATLAB function.',a/b));
```

The speedup is 14.9 times using the generated MEX over the baseline MATLAB function.

## 使用 Black Litterman 方法生成用于优化投资组合的代码

此示例说明如何使用 Black Litterman 方法从 MATLAB® 代码生成可进行投资组合优化的 MEX 函数和 C 源代码。

### 前提条件

此示例没有任何前提条件。

### 关于 hlblacklitterman 函数

hlblacklitterman.m 函数读入关于投资组合的财务信息，并使用 Black Litterman 方法执行投资组合优化。

type hlblacklitterman

```
function [er, ps, w, pw, lambda, theta] = hlblacklitterman(delta, weq, sigma, tau, P, Q, Omega)%#codegen
% hlblacklitterman
% This function performs the Black-Litterman blending of the prior
% and the views into a new posterior estimate of the returns as
% described in the paper by He and Litterman.
% Inputs
% delta - Risk tolerance from the equilibrium portfolio
% weq - Weights of the assets in the equilibrium portfolio
% sigma - Prior covariance matrix
% tau - Coefficient of uncertainty in the prior estimate of the mean (pi)
% P - Pick matrix for the view(s)
% Q - Vector of view returns
% Omega - Matrix of variance of the views (diagonal)
% Outputs
% Er - Posterior estimate of the mean returns
% w - Unconstrained weights computed given the Posterior estimates
% of the mean and covariance of returns.
% lambda - A measure of the impact of each view on the posterior estimates.
% theta - A measure of the share of the prior and sample information in the
% posterior precision.

% Reverse optimize and back out the equilibrium returns
% This is formula (12) page 6.
pi = weq * sigma * delta;
% We use tau * sigma many places so just compute it once
ts = tau * sigma;
% Compute posterior estimate of the mean
% This is a simplified version of formula (8) on page 4.
er = pi' + ts * P' * inv(P * ts * P' + Omega) * (Q - P * pi');
% We can also do it the long way to illustrate that d1 + d2 = I
d = inv(inv(ts) + P' * inv(Omega) * P);
d1 = d * inv(ts);
d2 = d * P' * inv(Omega) * P;
er2 = d1 * pi' + d2 * pinv(P) * Q;
% Compute posterior estimate of the uncertainty in the mean
% This is a simplified and combined version of formulas (9) and (15)
ps = ts - ts * P' * inv(P * ts * P' + Omega) * P * ts;
posteriorSigma = sigma + ps;
% Compute the share of the posterior precision from prior and views,
% then for each individual view so we can compare it with lambda
```



```

theta=zeros(1,2+size(P,1));
theta(1,1) = (trace(inv(ts) * ps) / size(ts,1));
theta(1,2) = (trace(P'*inv(Omega)*P* ps) / size(ts,1));
for i=1:size(P,1)
    theta(1,2+i) = (trace(P(i,:)*inv(Omega(i,i))*P(i,:)* ps) / size(ts,1));
end
% Compute posterior weights based solely on changed covariance
w = (er' * inv(delta * posteriorSigma));
% Compute posterior weights based on uncertainty in mean and covariance
pw = (pi * inv(delta * posteriorSigma));
% Compute lambda value
% We solve for lambda from formula (17) page 7, rather than formula (18)
% just because it is less to type, and we've already computed w*.
lambda = pinv(P)' * (w*(1+tau) - weq);
end

% Black-Litterman example code for MatLab (hlblacklitterman.m)
% Copyright (c) Jay Walters, blacklitterman.org, 2008.
%
% Redistribution and use in source and binary forms,
% with or without modification, are permitted provided
% that the following conditions are met:
%
% Redistributions of source code must retain the above
% copyright notice, this list of conditions and the following
% disclaimer.
%
% Redistributions in binary form must reproduce the above
% copyright notice, this list of conditions and the following
% disclaimer in the documentation and/or other materials
% provided with the distribution.
%
% Neither the name of blacklitterman.org nor the names of its
% contributors may be used to endorse or promote products
% derived from this software without specific prior written
% permission.
%
% THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
% CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
% INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
% DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
% CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
% SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
% BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
% SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
% INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
% WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
% NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
% OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
% DAMAGE.
%
% This program uses the examples from the paper "The Intuition
% Behind Black-Litterman Model Portfolios", by He and Litterman,
% 1999. You can find a copy of this paper at the following url.
% http://papers.ssrn.com/sol3/papers.cfm?abstract\_id=334304
%
% For more details on the Black-Litterman model you can also view

```

```
% "The BlackLitterman Model: A Detailed Exploration", by this author
% at the following url.
%   http://www.blacklitterman.org/Black-Litterman.pdf
%
```

%#codegen 指令指示 MATLAB 代码用于代码生成。

### 生成用于测试的 MEX 函数

使用 `codegen` 命令生成 MEX 函数。

```
codegen hlblacklitterman -args {0, zeros(1, 7), zeros(7,7), 0, zeros(1, 7), 0, 0}
```

Code generation successful.

在生成 C 代码之前，应首先在 MATLAB 中测试 MEX 函数，以确保它在功能上等同于原始 MATLAB 代码，并且不会出现任何运行时错误。默认情况下，`codegen` 在当前文件夹中生成名为 `hlblacklitterman_mex` 的 MEX 函数。这允许您测试 MATLAB 代码和 MEX 函数，并将结果进行比较。

### 运行 MEX 函数

调用生成的 MEX 函数

```
testMex();
```

```
View 1
Country    P    mu    w*
Australia    0  4.328  1.524
Canada      0  7.576  2.095
France     -29.5  9.288 -3.948
Germany     100  11.04  35.41
Japan        0  4.506  11.05
UK         -70.5  6.953 -9.462
USA         0  8.069  58.57
q           5
omega/tau    0.0213
lambda       0.317
theta        0.0714
pr theta     0.929
```

```
View 1
Country    P    mu    w*
Australia    0  4.328  1.524
Canada      0  7.576  2.095
France     -29.5  9.288 -3.948
Germany     100  11.04  35.41
Japan        0  4.506  11.05
UK         -70.5  6.953 -9.462
USA         0  8.069  58.57
q           5
omega/tau    0.0213
lambda       0.317
theta        0.0714
pr theta     0.929
```

```
Execution Time - MATLAB function: 0.050615 seconds
Execution Time - MEX function : 0.012026 seconds
```

## 生成 C 代码

```
cfg = coder.config('lib');
codegen -config cfg hlblacklitterman -args {0, zeros(1, 7), zeros(7,7), 0, zeros(1, 7), 0, 0}
```

Code generation successful.

将 `codegen` 与指定的 `-config cfg` 选项结合使用生成独立 C 库。

## 检查生成的代码

默认情况下，为库生成的代码位于文件夹 `codegen/lib/hlblacklitterman/` 中。

这些文件是：

```
dir codegen/lib/hlblacklitterman/
```

```
.          ..          _clang-format          buildInfo.mat          codeInfo.mat          codedo
```

## 检查 hlblacklitterman.c 函数的 C 代码

```
type codegen/lib/hlblacklitterman/hlblacklitterman.c
```

```
/*
 * Prerelease License - for engineering feedback and testing purposes
 * only. Not for sale.
 * File: hlblacklitterman.c
 *
 * MATLAB Coder version      : 5.6
 * C/C++ source code generated on : 30-Jan-2023 12:46:45
 */

/* Include Files */
#include "hlblacklitterman.h"
#include "inv.h"
#include "rt_nonfinite.h"
#include "rt_nonfinite.h"
#include <math.h>

/* Function Definitions */
/*
 * hlblacklitterman
 * This function performs the Black-Litterman blending of the prior
 * and the views into a new posterior estimate of the returns as
 * described in the paper by He and Litterman.
 * Inputs
 * delta - Risk tolerance from the equilibrium portfolio
 * weq - Weights of the assets in the equilibrium portfolio
 * sigma - Prior covariance matrix
 * tau - Coefficient of uncertainty in the prior estimate of the mean (pi)
 * P - Pick matrix for the view(s)
 * Q - Vector of view returns
 * Omega - Matrix of variance of the views (diagonal)
 * Outputs
 * Er - Posterior estimate of the mean returns
 * w - Unconstrained weights computed given the Posterior estimates
 * of the mean and covariance of returns.
 * lambda - A measure of the impact of each view on the posterior estimates.
 * theta - A measure of the share of the prior and sample information in the
```

```

*      posterior precision.
*
* Arguments  : double delta
*              const double weq[7]
*              const double sigma[49]
*              double tau
*              const double P[7]
*              double Q
*              double Omega
*              double er[7]
*              double ps[49]
*              double w[7]
*              double pw[7]
*              double *lambda
*              double theta[3]
* Return Type : void
*/
void hlblacklitterman(double delta, const double weq[7], const double sigma[49],
                     double tau, const double P[7], double Q, double Omega,
                     double er[7], double ps[49], double w[7], double pw[7],
                     double *lambda, double theta[3])
{
    double b_er_tmp[49];
    double dv[49];
    double posteriorSigma[49];
    double ts[49];
    double er_tmp[7];
    double pi[7];
    double y_tmp[7];
    double absxk;
    double b_P;
    double b_y_tmp;
    double nrm;
    double scale;
    int br;
    int i;
    int ib;
    int ic;
    int ps_tmp;
    boolean_T p;
    /* Reverse optimize and back out the equilibrium returns */
    /* This is formula (12) page 6. */
    for (i = 0; i < 7; i++) {
        nrm = 0.0;
        for (ic = 0; ic < 7; ic++) {
            nrm += weq[ic] * sigma[ic + 7 * i];
        }
        pi[i] = nrm * delta;
    }
    /* We use tau * sigma many places so just compute it once */
    for (i = 0; i < 49; i++) {
        ts[i] = tau * sigma[i];
    }
    /* Compute posterior estimate of the mean */
    /* This is a simplified version of formula (8) on page 4. */
    b_y_tmp = 0.0;
    b_P = 0.0;
    for (i = 0; i < 7; i++) {

```

```

nrm = 0.0;
scale = 0.0;
for (ic = 0; ic < 7; ic++) {
    absxk = P[ic];
    nrm += ts[i + 7 * ic] * absxk;
    scale += absxk * ts[ic + 7 * i];
}
y_tmp[i] = scale;
er_tmp[i] = nrm;
nrm = P[i];
b_y_tmp += scale * nrm;
b_P += nrm * pi[i];
}
absxk = 1.0 / (b_y_tmp + Omega);
scale = Q - b_P;
/* We can also do it the long way to illustrate that d1 + d2 = I */
b_y_tmp = 1.0 / Omega;
/* Compute posterior estimate of the uncertainty in the mean */
/* This is a simplified and combined version of formulas (9) and (15) */
nrm = 0.0;
for (i = 0; i < 7; i++) {
    er[i] = pi[i] + er_tmp[i] * absxk * scale;
    nrm += y_tmp[i] * P[i];
}
absxk = 1.0 / (nrm + Omega);
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        b_er_tmp[ic + 7 * i] = er_tmp[ic] * absxk * P[i];
    }
}
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        nrm = 0.0;
        for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
            nrm += b_er_tmp[i + 7 * ps_tmp] * ts[ps_tmp + 7 * ic];
        }
        ps_tmp = i + 7 * ic;
        ps[ps_tmp] = ts[ps_tmp] - nrm;
    }
}
for (i = 0; i < 49; i++) {
    posteriorSigma[i] = sigma[i] + ps[i];
}
/* Compute the share of the posterior precision from prior and views, */
/* then for each individual view so we can compare it with lambda */
inv(ts, dv);
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        nrm = 0.0;
        for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
            nrm += dv[i + 7 * ps_tmp] * ps[ps_tmp + 7 * ic];
        }
        ts[i + 7 * ic] = nrm;
    }
}
b_P = 0.0;
for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    b_P += ts[ps_tmp + 7 * ps_tmp];
}

```

```

}
theta[0] = b_P / 7.0;
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        b_er_tmp[ic + 7 * i] = P[ic] * b_y_tmp * P[i];
    }
}
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        nrm = 0.0;
        for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
            nrm += b_er_tmp[i + 7 * ps_tmp] * ps[ps_tmp + 7 * ic];
        }
        ts[i + 7 * ic] = nrm;
    }
}
b_P = 0.0;
for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    b_P += ts[ps_tmp + 7 * ps_tmp];
}
theta[1] = b_P / 7.0;
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        b_er_tmp[ic + 7 * i] = P[ic] * b_y_tmp * P[i];
    }
}
for (i = 0; i < 7; i++) {
    for (ic = 0; ic < 7; ic++) {
        nrm = 0.0;
        for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
            nrm += b_er_tmp[i + 7 * ps_tmp] * ps[ps_tmp + 7 * ic];
        }
        ts[i + 7 * ic] = nrm;
    }
}
b_P = 0.0;
for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    b_P += ts[ps_tmp + 7 * ps_tmp];
}
theta[2] = b_P / 7.0;
/* Compute posterior weights based solely on changed covariance */
for (i = 0; i < 49; i++) {
    b_er_tmp[i] = delta * posteriorSigma[i];
}
inv(b_er_tmp, dv);
for (i = 0; i < 7; i++) {
    nrm = 0.0;
    for (ic = 0; ic < 7; ic++) {
        nrm += er[ic] * dv[ic + 7 * i];
    }
    w[i] = nrm;
}
/* Compute posterior weights based on uncertainty in mean and covariance */
for (i = 0; i < 49; i++) {
    posteriorSigma[i] *= delta;
}
inv(posteriorSigma, dv);
/* Compute lambda value */

```

```

/* We solve for lambda from formula (17) page 7, rather than formula (18) */
/* just because it is less to type, and we've already computed w*. */
for (i = 0; i < 7; i++) {
    nrm = 0.0;
    for (ic = 0; ic < 7; ic++) {
        nrm += pi[ic] * dv[ic + 7 * i];
    }
    pw[i] = nrm;
    er_tmp[i] = P[i];
}
p = true;
for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    pi[ps_tmp] = 0.0;
    if (p) {
        nrm = P[ps_tmp];
        if (rtIsInf(nrm) || rtIsNaN(nrm)) {
            p = false;
        }
    } else {
        p = false;
    }
}
if (!p) {
    for (i = 0; i < 7; i++) {
        pi[i] = rtNaN;
    }
} else {
    nrm = 0.0;
    scale = 3.3121686421112381E-170;
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        absxk = fabs(P[ps_tmp]);
        if (absxk > scale) {
            b_P = scale / absxk;
            nrm = nrm * b_P * b_P + 1.0;
            scale = absxk;
        } else {
            b_P = absxk / scale;
            nrm += b_P * b_P;
        }
    }
    nrm = scale * sqrt(nrm);
    if (nrm > 0.0) {
        if (P[0] < 0.0) {
            absxk = -nrm;
        } else {
            absxk = nrm;
        }
        if (fabs(absxk) >= 1.0020841800044864E-292) {
            scale = 1.0 / absxk;
            for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
                er_tmp[ps_tmp] *= scale;
            }
        } else {
            for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
                er_tmp[ps_tmp] /= absxk;
            }
        }
    }
    er_tmp[0]++;
}

```

```

    absxk = -absxk;
} else {
    absxk = 0.0;
}
for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    y_tmp[ps_tmp] = er_tmp[ps_tmp];
}
if (absxk != 0.0) {
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        y_tmp[ps_tmp] = -y_tmp[ps_tmp];
    }
    y_tmp[0]++;
    nrm = fabs(absxk);
    scale = absxk / nrm;
    absxk = nrm;
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        y_tmp[ps_tmp] *= scale;
    }
} else {
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        y_tmp[ps_tmp] = 0.0;
    }
    y_tmp[0] = 1.0;
}
if (rtIsInf(absxk)) {
    scale = rtNaN;
} else if (absxk < 4.4501477170144028E-308) {
    scale = 4.94065645841247E-324;
} else {
    frexp(absxk, &br);
    scale = ldexp(1.0, br - 53);
}
if (absxk > 7.0 * scale) {
    scale = 1.0 / absxk;
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        i = ps_tmp + 1;
        for (ic = i; ic <= i; ic++) {
            pi[ic - 1] = 0.0;
        }
    }
    br = 0;
    for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
        br++;
        for (ib = br; ib <= br; ib += 7) {
            i = ps_tmp + 1;
            for (ic = i; ic <= i; ic++) {
                pi[ic - 1] += y_tmp[ib - 1] * scale;
            }
        }
    }
}
}
}
*lambda = 0.0;
for (ps_tmp = 0; ps_tmp < 7; ps_tmp++) {
    *lambda += pi[ps_tmp] * (w[ps_tmp] * (tau + 1.0) - weq[ps_tmp]);
}
}

```



```
/*  
 * File trailer for hlblacklitterman.c  
 *  
 * [EOF]  
 */
```



# MATLAB 中的半精度

---

- “代码生成 workflow” (第 27-2 页)
- “从 MATLAB 代码生成独立的 C/C++ 可执行文件” (第 27-3 页)
- “配置编译设置” (第 27-12 页)
- “在 macOS 平台上安装 OpenMP 库” (第 27-20 页)
- “生成代码以检测图像边缘” (第 27-21 页)
- “为 MATLAB 卡尔曼滤波算法生成 C 代码” (第 27-27 页)
- “使用 Black Litterman 方法生成用于优化投资组合的代码” (第 27-36 页)



## 验证生成的 C/C++ 代码

---



## MATLAB 代码的代码替换

---





## 自定义工具链注册

---

- “创建和编辑工具链定义文件” (第 31-2 页)
- “使用 MSVC 工具链在 64 位 Windows® 平台上编译 32 位 DLL” (第 31-3 页)

## 创建和编辑工具链定义文件

此示例说明如何通过复制和粘贴示例文件来创建工具链定义文件。然后，您可以更新相关元素，并根据自定义工具链的需要添加或删除其他元素。这是创建和使用自定义工具链定义的典型工作流的第一步。有关工作流的详细信息，请参阅“Typical Workflow”。

- 1 查看已注册工具链的列表。在 MATLAB 命令行窗口中，输入：

```
coder.make.getToolchains
```

生成的输出包括您的开发计算机环境的出厂工具链列表以及以前注册的自定义工具链。

- 2 通过打开“Add Custom Toolchains to MATLAB® Coder™ Build Process”示例创建示例文件的文件夹。
- 3 将示例工具链定义文件复制到另一个位置并重命名它。例如：

```
copyfile('intel_tc.m','./newtoolchn_tc.m')
```

- 4 在 MATLAB 编辑器中打开新的工具链定义文件。例如：

```
cd ./  
edit newtoolchn_tc.m
```

- 5 编辑该新工具链定义文件的内容，以提供自定义工具链的信息。

有关示例工具链定义文件的扩展注释，请参阅“Toolchain Definition File with Commentary”。

有关可在工具链定义文件中使用的类属性和方法的参考信息，请参阅 **coder.make.ToolchainInfo**。

- 6 保存对工具链定义文件所做的更改。

接下来，从工具链定义文件中创建一个 **coder.make.ToolchainInfo** 对象并进行验证，如“Create and Validate ToolchainInfo Object”中所述。

## 使用 MSVC 工具链在 64 位 Windows® 平台上编译 32 位 DLL

**注意：**如果只需要在安装了 Microsoft® Visual Studio 的 Windows® 上生成 32 位 DLL，请使用 CMake 工具链，如 **Microsoft Visual Studio Project 2017 | CMake (32-bit Windows)**。本页说明如何创建自定义工具链来更改编译器选项和集成新编译器。有关使用 CMake 编译所生成代码的详细信息，请参阅“Configure CMake Build Process”。

注册并使用运行在 64 位 Windows® 平台上的 Microsoft® Visual C/C++ (MSVC) 工具链，编译 32 位动态链接库 (DLL)。此示例使用 Microsoft® 编译器。然而，概念和编程接口适用于其他工具链。一旦注册工具链，就可以从工具链列表中选择它，代码生成器会生成联编文件来使用该工具链编译代码。工具链由几个工具组成，如编译器、链接器和具有多个不同配置选项的存档器。工具链在指定的平台上编译、链接和运行代码。要访问此示例使用的文件，请点击[打开脚本](#)。

### 检查平台并确定 MSVC 版本

以下代码检查平台是否受支持，以及您是否有受支持的 Microsoft® Visual C/C++ 版本。

`my_msvc_32bit_tc.m` 工具链定义可以使用 Microsoft® Visual Studio 版本 9.0、10.0、11.0、12.0、14.0 或 15.0。

如果您没有使用 Windows® 平台，或如果您没有支持的 Microsoft® Visual C/C++ 版本，该示例仅生成代码和联编文件，而不运行生成的联编文件。

```
VersionNumbers = {'14.0'}; % Placeholder value
if ~ispc
    supportedCompilerInstalled = false;
else
    installed_compilers = mex.getCompilerConfigurations('C', 'Installed');
    MSVC_InstalledVersions = regexp({installed_compilers.Name}, 'Microsoft Visual C\+\+ 20\d\d');
    MSVC_InstalledVersions = cellfun(@(a)~isempty(a), MSVC_InstalledVersions);
    if ~any(MSVC_InstalledVersions)
        supportedCompilerInstalled = false;
    else
        VersionNumbers = {installed_compilers(MSVC_InstalledVersions).Version}';
        supportedCompilerInstalled = true;
    end
end
```

### 动态链接库的函数

动态链接库的示例函数 `myMatlabFunction.m` 将数字乘以 2。

```
function y = myMatlabFunction(u)
% myMatlabFunction: Returns twice its input.
% Copyright 2017 The MathWorks, Inc.

%#codegen
assert(isa(u, 'double'), 'The input must be a "double".');
assert(all([1, 1] == size( u )), 'The input must be a scalar.');
```

`y = double(u + u);`

## 创建和配置 MSVC 工具链

`my_msvc_32bit_tc.m` 工具链定义函数接受包含 Visual Studio 版本号的参数。在此示例中，创建和配置该工具链的命令是：

```
tc = my_msvc_32bit_tc(VersionNumbers{end});
save my_msvc_32bit_tc tc;
```

```
Executing "C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc22b.j2019614\coder-ex19875030\my_msvc_32bit_tc.m"
Executed "C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc22b.j2019614\coder-ex19875030\my_msvc_32bit_tc.m"
```

## 注册工具链

在代码生成器可以将工具链用于编译过程之前，`RTW.TargetRegistry` 必须包含工具链注册。此注册可以来自 MATLAB 路径中的任何 `rtwTargetInfo.m` 文件。如果重置 `RTW.TargetRegistry`，MATLAB 将加载新注册。

从对应的文本文件 `myRtwTargetInfo.txt` 创建 `rtwTargetInfo.m` 文件。

```
function myRtwTargetInfo(tr)
%RTWTARGETINFO Registration file for custom toolchains.

% Copyright 2012-2017 The MathWorks, Inc.

tr.registerTargetInfo(@createToolchainRegistryFor32BitMSVCToolchain);

end

% -----
% Create the ToolchainInfoRegistry entries
% -----
function config = createToolchainRegistryFor32BitMSVCToolchain

config(1) = coder.make.ToolchainInfoRegistry;
config(1).Name = 'Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)';
config(1).FileName = fullfile(fileparts(mfilename('fullpath')), 'my_msvc_32bit_tc.mat');
config(1).TargetHWDeviceType = {'Intel->x86-32 (Windows32)', 'AMD->x86-32 (Windows32)', 'Generic->Unspecified'};
config(1).Platform = {'win64'};

end

copyfile myRtwTargetInfo.txt rtwTargetInfo.m
RTW.TargetRegistry.getInstance('reset');
```

## 创建代码生成配置对象

要生成 32 位动态链接库 (DLL)，请创建一个 'dll' 代码生成配置对象。指定 'dll' 会指示链接器（工具链中的一个编译工具）使用“共享库”链接器命令。

```
cfg = coder.config('dll');
```

## 为 32 位硬件配置代码生成

要成功生成与 32 位硬件兼容的代码，生成的代码必须使用正确的底层 C 类型（例如，`int`、`signed char` 等）。这些类型是大小类型的 `typedef` 语句的基础（例如，`uint8`、`int16` 等）。使用以下命令设置配置：

```
cfg.HardwareImplementation.ProdHWDeviceType = ...
    'Generic->Unspecified (assume 32-bit Generic)';
```

### 将代码生成配置为使用 32 位工具链

设置 Toolchain 属性的名称，以匹配您在 rtwTargetInfo.m 文件中指定的 Name。

```
cfg.Toolchain = ...
    'Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)';
```

### 选择详细状态报告

要确认工具链用于构建 DLL 的编译器标志，请选择详细状态报告。

```
cfg.Verbose = true;
```

### 确定是否仅生成代码

当没有安装 Microsoft® 编译器时，代码生成器只生成代码和联编文件。安装支持的编译器后，代码生成器会构建 32 位二进制文件。

```
if supportedCompilerInstalled
    cfg.GenCodeOnly = false;
else
    cfg.GenCodeOnly = true;
end
```

### 生成代码并编译 DLL

要使用工具链生成代码并编译 DLL（如果已启用编译），请在命令提示符下输入：

```
codegen -config cfg myMatlabFunction -args { double(1.0) };
```

```
### Compiling function(s) myMatlabFunction ...
```

```
### Using toolchain: Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)
```

```
### 'C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc22bj2019614\coder-ex19875030\codegen\dll\
```

```
### Building 'myMatlabFunction': nmake -f myMatlabFunction_rtw.mk all
```

```
*****
```

```
** Visual Studio 2019 Developer Command Prompt v16.4.5
```

```
** Copyright (c) 2019 Microsoft Corporation
```

```
*****
```

```
[vcvarsall.bat] Environment initialized for: 'x64_x86'
```

```
Microsoft (R) Program Maintenance Utility Version 14.24.28316.0
```

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

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_data.c
```

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_initialize.c
```

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_terminate.c
```

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction.c
```

```
### Creating dynamic library ".\myMatlabFunction.dll" ...
```

```
link /MACHINE:X86 /DEBUG /DEBUGTYPE:cv /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsoc
Creating library .\myMatlabFunction.lib and object .\myMatlabFunction.exp
```

```
### Created: .\myMatlabFunction.dll
```

```
### Successfully generated all binary outputs.
```

---

```
Code generation successful.
```

### 编译并运行可执行文件

如果您安装支持的编译器版本，您可以通过使用 C 主函数来构建 32 位可执行文件。您可以使用可执行文件来测试生成的代码是否按预期工作。

```
cfge = coder.config('exe');
cfge.CustomInclude = pwd;
cfge.CustomSource = 'myMatlabFunction_main.c';
cfge.GenCodeOnly = cfge.GenCodeOnly;
cfge.Verbose = true;
cfge.Toolchain = ...
    'Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)';
codegen -config cfge myMatlabFunction -args { double(1.0) };
if supportedCompilerInstalled
    pause(5); %wait for EXE to get generated
    system('myMatlabFunction 3.1416'); % Expected output: myMatlabFunction(3.1416) = 6.2832
end
```

```
### Compiling function(s) myMatlabFunction ...
```

```
### Using toolchain: Microsoft 32 Bit Toolchain | nmake makefile (64-bit Windows)
### 'C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc22b.j2019614\coder-ex19875030\codegen\exe
### Building 'myMatlabFunction': nmake -f myMatlabFunction_rtw.mk all
*****
** Visual Studio 2019 Developer Command Prompt v16.4.5
** Copyright (c) 2019 Microsoft Corporation
*****
[vcvarsall.bat] Environment initialized for: 'x64_x86'

Microsoft (R) Program Maintenance Utility Version 14.24.28316.0
Copyright (C) Microsoft Corporation. All rights reserved.

    cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_main.c
    cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_data.c
    cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_initialize.c
    cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction_terminate.c
    cl -c -nologo -GS -W4 -DWIN32 -D_MT -MTd -D_CRT_SECURE_NO_WARNINGS /O2 /Oy- -DMODEL=myMatlab
myMatlabFunction.c
### Creating standalone executable "C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc22b.j2019614
    link /MACHINE:X86 /DEBUG /DEBUGTYPE:cv /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsoc
### Created: C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc22b.j2019614\coder-ex19875030\myM
### Successfully generated all binary outputs.
```

---

```
Code generation successful.
```

```
myMatlabFunction(3.1416) = 6.2832
```

### 可选步骤：注销工具链

要注销工具链，请输入：

```
delete ./rtwTargetInfo.m  
RTW.TargetRegistry.getInstance('reset');
```

### 另请参阅

### 详细信息

- “Add Custom Toolchains to MATLAB® Coder™ Build Process”





## 部署生成的代码

---

- “在生成的函数接口中使用 C 数组” (第 32-2 页)
- “使用示例主函数合并生成的代码” (第 32-13 页)
- “在应用程序中使用示例 C 主函数” (第 32-15 页)
- “生成的示例 C/C++ 主函数的结构” (第 32-31 页)

## 在生成的函数接口中使用 C 数组

在大多数情况下，当您为接受或返回数组的 MATLAB 函数生成代码时，生成的 C/C++ 函数接口包含数组。要使用生成的函数接口，请学习如何定义和构造生成的 C/C++ 数组。特别是，学习使用为表示动态分配数组而生成的 `emxArray` 数据结构体。

当您生成 C/C++ 代码时，系统会创建一个示例主文件，它显示如何将数组与生成的函数代码结合使用。您可以使用示例主文件作为您自己的应用程序的模板或起点。

### 生成的 C/C++ 代码中数组的实现

代码生成器会生成 C/C++ 数组定义，这些定义取决于数组元素类型以及数组使用的是静态还是动态内存分配。数组的两种内存分配形式需要两种不同的实现方式：

- 对于其大小在预定义阈值内的数组，生成的 C/C++ 定义包含一个指向内存的指针和一个存储数组元素总数（数组大小）的整数。此数组的内存来自程序堆栈，并且是静态分配的。
- 对于在编译时其大小未知且无界的数组，或者其界限超过预定义阈值的数组，生成的 C/C++ 定义包含称为 `emxArray` 的数据结构体。创建 `emxArray` 时，会根据当前数组大小设置中间存储边界。在程序执行期间，当超过中间存储边界时，生成的代码会从堆中占用额外的内存空间，并将其添加到 `emxArray` 存储中。此数组的内存是动态分配的。

默认情况下，限制在阈值大小内的数组不会在生成的代码中使用动态分配。您也可以禁用动态内存分配和更改动态内存分配阈值。请参阅“Control Memory Allocation for Variable-Size Arrays”。

下表列出了在生成的代码中数组表示的一些典型情况。

算法说明和数组大小	MATLAB 函数	生成的 C 函数接口
在一个固定大小的 1×500 行向量的所有位置上放置元素 1。  固定大小，限制在阈值内。	<pre>function B = create_vec0 %#codegen B = zeros(1,500); j = 1; for i = 1:500     if round(rand)         B(1,j) = 1;         j = j + 1;     end end</pre>	<pre>void create_vec0(double B[500])</pre>
将元素 1 推送到一个限制为 300 个元素的可变大小的行向量上。  可变大小，限制在阈值内。	<pre>function B = create_vec %#codegen B = zeros(1,0); coder.varsize('B',[1 300],[0 1]); for i = 1:500     if round(rand)         B = [1 B];     end end</pre>	<pre>void create_vec(double B_data[],... int B_size[2])</pre>
将元素 1 推送到一个限制为 30000 个元素的可变大小的行向量上。  可变大小，不在阈值范围内。	<pre>function B = create_vec2 %#codegen B = zeros(1,0); coder.varsize('B',[1 30000],[0 1]); for i = 1:500     if round(rand)         B = [1 B];     end end</pre>	<pre>void create_vec2(emxArray_real_T *B)</pre>
创建一个数组，其大小由无界整数输入决定。  编译时未知且无界。	<pre>function y = create_vec3(n) %#codegen y = int8(ones(1,n));</pre>	<pre>void create_vec3(int n,... emxArray_int8_T *y)</pre>

## emxArray 动态数据结构体定义

在生成的 C/C++ 代码中，**emxArray** 数据结构体定义取决于它存储的元素的数据类型。一般定义采用以下形式：

```
struct emxArray_<name>
{
    <type> *data;
    int *size;
    int allocatedSize;
    int numDimensions;
    boolean_T canFreeData;
};
```

在定义中，<type> 表示数据类型，<name> 表示用于标识 **emxArray** 结构体的名称。代码生成器根据为 MEX 代码生成定义的类型选择 <name>，如 “Mapping MATLAB Types to Types in Generated Code” 中所列。

例如，假设为函数 **create\_vec2** 生成了 **emxArray** 定义。<name> 为 **emxArray\_real\_T**，<type> 为 **double**。

```
struct emxArray_real_T
{
    double *data;
    int *size;
    int allocatedSize;
    int numDimensions;
    boolean_T canFreeData;
};
```

在代码生成之前，不要尝试预测 <type> 和 <name> 项。在代码生成完成后，可以从代码生成报告中检查文件 <myFunction>\_types.h。<myFunction> 是您的入口函数的名称。

生成的代码还可以使用 **typedef** 语句来定义 **emxArray** 结构体，如下示例中所示。

```
typedef struct {
    emxArray_real_T *f1;
} cell_wrap_0;

typedef struct {
    cell_wrap_0 *data;
    int *size;
    int allocatedSize;
    int numDimensions;
    boolean_T canFreeData;
} emxArray_cell_wrap_0;
```

下表说明了 **emxArray** 结构体字段。

字段	描述
<type> *data	指向 <type> 类型的元素数组的指针。
int *size	指向大小向量的指针。大小向量的第 i 个元素存储数组的第 i 个维度的长度。
int allocatedSize	为数组分配的内存元素数。如果数组大小发生变化，生成的代码会根据新大小重新分配内存。

字段	描述
<b>int numDimensions</b>	大小向量的长度。在不越界进入未分配或未使用内存的情况下可访问的维度数。
<b>boolean_T canFreeData</b>	指示如何取消分配内存的布尔标志。仅由内部 <b>emxArray</b> 处理例程使用。 <ul style="list-style-type: none"> <li><b>true</b> - 生成的代码自行取消分配内存。</li> <li><b>false</b> - 实例化 <b>emxArray</b> 的程序必须手动取消分配 <b>data</b> 指向的内存。</li> </ul>

## 与 emxArray 数据交互的工具函数

为了在您的 C/C++ 代码中创建 **emxArray** 数据并与之交互，代码生成器使用一个用户友好的 API 导出一组 C/C++ 辅助函数。使用这些函数可确保您正确地初始化和销毁 **emxArray** 数据类型。要使用这些函数，请在您的 C 代码中为生成的头文件 **<myFunction>\_emxAPI.h** 插入 **include** 语句。

**<myFunction>** 是您的入口函数的名称。对于由代码生成器生成的、在 **<myFunction>\_emxutil.h** 中定义的、对 **emxArray** 数据进行操作的其他函数，最好不要手动修改。

默认情况下，为 **lib**、**dll** 和 **exe** 代码生成的示例主文件包括对 **emxArray** API 函数的调用。示例主代码将 **emxArray** 数据初始化为泛型零值。要使用实际数据输入和值，请修改示例主文件或创建您自己的主文件。有关使用主函数的详细信息，请参阅“使用示例主函数合并生成的代码”（第 32-13 页）。

下表显示了导出的 **emxArray** API 函数的列表。一些 API 函数接受 **emxArray** 数据的初始行数、列数或维数。每个维度都可以根据需要增长以容纳新数据。使用指针实例化的 **emxArray** 数组保留输入值的副本。在运行时更改输入变量的值不会更改 **emxArray** 的大小。

<b>emxArray 辅助函数</b>	描述
<b>emxArray_&lt;name&gt; *emxCreate_&lt;name&gt;(int rows, int cols)</b>	创建一个指向二维 <b>emxArray</b> 的指针，且将数据元素初始化为零。为数据分配新内存。
<b>emxArray_&lt;name&gt; *emxCreateND_&lt;name&gt;(int numDimensions, int *size)</b>	创建一个指向 N 维 <b>emxArray</b> 的指针，且将数据元素初始化为零。为数据分配新内存。
<b>emxArray_&lt;name&gt; *emxCreateWrapper_&lt;name&gt;(&lt;type&gt; *data, int rows, int cols)</b>	创建一个指向二维 <b>emxArray</b> 的指针。使用您提供的数据和内存，并将其打包到一个 <b>emxArray</b> 数据结构体中。将 <b>canFreeData</b> 设置为 <b>false</b> 以防止无意中释放用户内存。
<b>emxArray_&lt;name&gt; *emxCreateWrapperND_&lt;name&gt;(&lt;type&gt; *data, int numDimensions, int *size)</b>	创建一个指向 N 维 <b>emxArray</b> 的指针。使用您提供的数据和内存，并将其打包到一个 <b>emxArray</b> 数据结构体中。将 <b>canFreeData</b> 设置为 <b>false</b> 以防止无意中释放用户内存。
<b>void emxInitArray_&lt;name&gt;(emxArray_&lt;name&gt; **pEmxArray, int numDimensions)</b>	为指向 <b>emxArray</b> 的双精度指针分配内存。
<b>void emxDestroyArray_&lt;name&gt;(emxArray_&lt;name&gt; *emxArray)</b>	释放由 <b>emxCreate</b> 或 <b>emxInitArray</b> 函数分配的动态内存。

代码生成器仅对作为入口函数参数或由 `coder.ceval` 调用的函数所使用的数组导出 `emxArray` API 函数。

## 示例

### 对静态分配的数组使用函数接口

以 “Generate Code for Variable-Size Data” 中的 MATLAB 函数 `myuniquetol` 为例。

```
function B = myuniquetol(A, tol) %#codegen
A = sort(A);
coder.varsize('B', [1 100], [0 1]);
B = zeros(1,0);
k = 1;
for i = 2:length(A)
    if abs(A(k) - A(i)) > tol
        B = [B A(i)];
        k = i;
    end
end
```

为 `myuniquetol` 生成代码。使用 `coder.typeof` 将输入类型指定为一个有界可变大小数组和一个双精度标量。

```
codegen -config:lib -report myuniquetol -args {coder.typeof(0,[1 100],[0 1]),coder.typeof(0)}
```

语句 `coder.varsize('B', [1 100], [0 1])` 指定 `B` 是可变大小数组，其第一个维度固定为 1，第二个维度最多可以有 100 个元素。由于数组 `B` 的最大大小限制在默认阈值大小内，因此代码生成器对数组使用静态内存分配。

生成的函数接口是：

```
void myuniquetol(const double A_data[], const int A_size[2], double tol,
double B_data[], int B_size[2])
```

该函数接口声明输入参数 `A` 和输出参数 `B`。`A_size` 包含 `A` 的大小。在调用 `myuniquetol` 后，`B_size` 包含 `B` 的大小。

使用 `B_size` 来确定在调用 `myuniquetol` 后 `B` 中可访问元素的数量。`B_size[0]` 包含第一个维度的大小。`B_size[1]` 包含第二个维度的大小。因此，`B` 中元素的数量是 `B_size[0]*B_size[1]`。即使 `B` 在 C 代码中有 100 个元素，但只有 `B_size[0]*B_size[1]` 个元素包含有效数据。

以下 C 主函数显示如何调用 `myuniquetol`。

```
void main()
{
    double A[100], B[100];
    int A_size[2] = { 1, 100 };
    int B_size[2];
    int i;
    for (i = 0; i < 100; i++) {
        A[i] = (double)1/i;
    }
    myuniquetol(A, A_size, 0.1, B, B_size);
}
```

### 使用 emxCreate 或 emxInitArray 函数创建 emxArray

`emxCreate` 和 `emxCreateND` API 函数可创建一个 `emxArray`，并根据需要从堆中分配新内存。然后，您可以使用 `emxArray` 作为生成代码的输入或输出。以下 C 代码示例说明如何使用 `emxCreate`。假设您已为使用数据类型 `emxArray_uint32_T` 的函数 `myFunction` 生成源代码。

```
#include <stdio.h>
#include <stdlib.h>
#include "myFunction_emxAPI.h"
#include "myFunction.h"

int main(int argc, char *argv[])
{
    /* Create a 10-by-10 uint32_T emxArray */
    emxArray_uint32_T *pEmx = emxCreate_uint32_T(10,10);

    /* Initialize the emxArray memory, if needed */
    int k = 0;
    for (k = 0; k < 100; ++k) {
        pEmx->data[k] = (uint32_T) k;
    }

    /* Use pEmx array here; */
    /* Insert call to myFunction */

    /* Deallocate any memory allocated in pEmx */
    /* This DOES free pEmx->data */
    emxDestroyArray_uint32_T(pEmx);

    /* Unused */
    (void)argc;
    (void)argv;

    return 0;
}
```

在本示例中，您知道 `emxArray` 的初始大小。如果您不知道数组的大小，例如当您使用数组存储输出时，您可以为 `rows` 和 `cols` 字段输入值 0。例如，如果不知道列数，您可以编写如下代码：

```
emxArray_uint32_T *pEmx = emxCreate_uint32_T(10,0);
```

数据结构体会根据需要增长以容纳数据。在函数运行后，可通过访问 `size` 和 `numDimensions` 字段来确定输出大小。

使用 `emxInitArray` API 函数创建一个作为输出返回的数组，您事先不知道该数组的大小。例如，要创建一个二维 `emxArray`，其任一维度的大小均未知，您可以编写如下代码：

```
emxArray_uint32_T *s;
emxInitArray_uint32_T(&s, 2);
```

### 将现有数据加载到 emxArray 中

`emxCreateWrapper` 和 `emxCreateWrapperND` API 函数使您能够将现有内存和数据加载或包装到一个 `emxArray` 中，以便将数据传递给生成的函数。以下 C 代码示例说明如何使用 `emxCreateWrapper`。假设您已为使用数据类型 `emxArray_uint32_T` 的函数 `myFunction` 生成源代码。

```

#include <stdio.h>
#include <stdlib.h>
#include "myFunction_emxAPI.h"
#include "myFunction.h"

int main(int argc, char *argv[])
{
    /* Create a 10-by-10 C array of uint32_T values */
    uint32_T x[100];
    int k = 0;
    emxArray_uint32_T *pEmx = NULL;
    for (k = 0; k < 100; k++) {
        x[k] = (uint32_T) k;
    }

    /* Load existing data into an emxArray */
    pEmx = emxCreateWrapper_uint32_T(x,10,10);

    /* Use pEmx here; */
    /* Insert call to myFunction */

    /* Deallocate any memory allocated in pEmx */
    /* This DOES NOT free pEmx->data because the wrapper function was used */
    emxDestroyArray_uint32_T(pEmx);

    /* Unused */
    (void)argc;
    (void)argv;

    return 0;
}

```

### 创建和使用嵌套的 emxArray 数据

此示例说明如何使用生成的代码，该代码包含嵌套在其他 **emxArray** 数据中的 **emxArray** 数据。要使用生成的代码，请在主函数或调用函数中，从底部节点开始自下而上初始化 **emxArray** 数据。

### MATLAB 算法

此 MATLAB 算法循环访问名为 **myarray** 的结构体数组。每个结构体都包含一个较低级别的值数组。该算法对每个 **struct** 的较低级别的数组的元素进行排序和求和。

```

% y is an array of structures of the form
% struct('values', [...], 'sorted', [...], 'sum', ... )
function y = processNestedArrays(y) %#codegen
coder.cstructname(y, 'myarray');
for i = 1:numel(y)
    y(i).sorted = sort(y(i).values);
    y(i).sum = sum(y(i).values);
end

```

### 生成用于测试的 MEX 函数

为了能够测试算法，请先生成一个 MEX 函数。使用 **coder.typeof** 函数手动将输入指定为由 **structs** 组成的无界可变大小行向量，这些结构体本身也包含无界可变大小行向量。

```
myarray = coder.typeof( ...
    struct('values', coder.typeof(0, [1 inf]), ...
        'sorted', coder.typeof(0, [1 inf]), ...
        'sum', coder.typeof(0)) , [1 inf]);
codegen -args {myarray} processNestedArrays
```

Code generation successful.

### 检查生成的函数接口

MEX 函数源代码包含使其能够与 MATLAB 运行时环境对接的专用代码，这使得它更加复杂，难以读懂。要生成更简化的源代码，请生成库代码。

```
codegen -config:lib -args {myarray} processNestedArrays -report
```

Code generation successful: To view the report, open('codegen/lib/processNestedArrays/html/report.mldatx')

从代码生成报告中检查生成的函数代码 **processNestedArrays.c**。生成的示例主文件 **main.c** 显示如何通过 **emxCreate** API 函数创建和初始化输入来调用生成的函数代码。

### 编写并使用您自己的自定义主文件来初始化 **emxArray** 数据

虽然生成的示例主文件显示了如何调用生成的函数代码，但它不包含所需输入值的信息。基于示例主文件编写您自己的主文件。使用您选择的编码样式和预设项。指定输入的值，并根据需要插入预处理和后处理代码。

文件 **processNestedArrays\_main.c** 显示了一个示例。此主文件使用 **emxArray** API 函数来创建和初始化结构体数据。对于生成的示例主文件和此手写主文件，代码都在底部（叶）节点初始化 **emxArray** 数据，并将该数据分配给上面的节点。

```
type processNestedArrays_main.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include "processNestedArrays_emxAPI.h"
#include "processNestedArrays.h"

static void print_vector(emxArray_real_T *v)
{
    int i;
    printf("[");
    for (i = 0; i < v->size[1]; i++) {
        if (i > 0) printf(" ");
        printf("%.0f", v->data[i]);
    }
    printf("] \n");
}

int main(int argc, char *argv[])
{
    int i;
    static double values_1[] = { 5, 3, 4, 1, 2, 6 };
    static double values_2[] = { 50, 30, 40, 10, 20, 60 };
    static double values_3[] = { 42, 4711, 1234 };
    static double * values[] = { values_1, values_2, values_3 };
```



```

static int values_len[] = { 6, 6, 3 };

/* Setup myarray emxArrays */
emxArray_myarray *myarr = emxCreate_myarray(1, 3); /* Create outer array */
for (i = 0; i < 3; i++) {
    /* Setup field 'values'. Don't allocate memory; reuse the data pointer. */
    myarr->data[i].values = emxCreateWrapper_real_T(values[i], 1, values_len[i]);
    /* Initialize the 'sorted' field to the empty vector. */
    myarr->data[i].sorted = emxCreate_real_T(1, 0);
    /* Initailize the 'sum' field. */
    myarr->data[i].sum = 0;
}

/* Call process function */
processNestedArrays(myarr);

/* Print result */
for (i = 0; i < myarr->size[1]; i++) {
    printf("  values: "); print_vector(myarr->data[i].values);
    printf("  sorted: "); print_vector(myarr->data[i].sorted);
    printf("    sum: %.0f \n\n", myarr->data[i].sum);
}

/* Cleanup memory */
emxDestroyArray_myarray(myarr);

/* Unused */
(void)argc;
(void)argv;

return 0;
}

```

## 生成可执行文件并将结果与 MEX 函数进行比较

使用提供的主文件，您可以为算法生成独立的可执行文件。

```
codegen -config:exe -args {myarray} processNestedArrays ...
processNestedArrays_main.c -report
```

Code generation successful: To view the report, open('codegen/exe/processNestedArrays/html/report.mldatx')

声明 MEX 函数的输入数据，这些数据与在 `processNestedArrays_main.c` 中定义的独立可执行文件的输入相匹配。

```
myarray = [struct('values', [5 3 4 1 2 6], 'sorted', zeros(1,0), 'sum', 0), ...
    struct('values', [50 30 40 10 20 60], 'sorted', zeros(1,0), 'sum', 0), ...
    struct('values', [42 4711 1234], 'sorted', zeros(1,0), 'sum', 0)];
```

将 MEX 函数结果与独立可执行文件结果进行比较。

```
fprintf('.mex output \n----- \n');
r = processNestedArrays_mex(myarray);
disp(r(1));
disp(r(2));
disp(r(3));
```

```
fprintf('exe output \n----- \n');
if isunix
    system('./processNestedArrays')
elseif ispc
    system('processNestedArrays.exe')
else
    disp('Platform is not supported')
end
```

.mex output

```
values: [5 3 4 1 2 6]
sorted: [1 2 3 4 5 6]
sum: 21

values: [50 30 40 10 20 60]
sorted: [10 20 30 40 50 60]
sum: 210

values: [42 4711 1234]
sorted: [42 1234 4711]
sum: 5987
```

.exe output

```
values: [5 3 4 1 2 6]
sorted: [1 2 3 4 5 6]
sum: 21

values: [50 30 40 10 20 60]
sorted: [10 20 30 40 50 60]
sum: 210

values: [42 4711 1234]
sorted: [42 1234 4711]
sum: 5987
```

ans =

0

输出结果是相同的。

### 使用 `emxArray_char_T` 数据和字符串输入

在此示例中，MATLAB 函数在运行时更改字符向量的大小。由于向量的最终长度可以变化，因此生成的 C 代码将向量实例化为动态大小的 `emxArray`。此示例说明如何编写一个主函数，该主函数使用 `emxArray_char_T` 和生成的函数接口。以此示例作为参考来使用 `emxArray_char_T` 数据类型。

### MATLAB 算法

函数 `replaceCats` 以字符向量作为输入，并用 'velociraptor' 和 'Velociraptor' 替换单词 'cat' 或 'Cat' 的所有实例。由于代码生成器无法在编译时确定输出长度，因此生成的代码使用 `emxArray` 数据类型。

```
function cstrNew = replaceCats(cstr)
%#codegen
cstrNew = replace(cstr,'cat','velociraptor');
cstrNew = replace(cstrNew,'Cat','Velociraptor');
```

## 生成源代码

要为 `replaceCats` 生成代码，请将函数的输入类型指定为可变大小的字符数组。

```
t = coder.typeof('a',[1 inf]);
codegen replaceCats -args {t} -report -config:lib
```

Code generation successful: To view the report, open('codegen/lib/replaceCats/html/report.mldatx')

在生成的代码中，示例主文件 `/codegen/lib/replaceCats/examples/main.c` 为您编写自己的主函数提供了模板。

## 基于模板创建主函数

修改主函数，以从命令行接受字符输入。使用 `emxCreate` 和 `emxCreateWrapper` API 函数来初始化您的 `emxArray` 数据。完成主源文件和头文件的编写后，将修改后的文件放在根文件夹中。

type `main_replaceCats.c`

```
#include "main_replaceCats.h"
#include "replaceCats.h"
#include "replaceCats_terminate.h"
#include "replaceCats_emxAPI.h"
#include "replaceCats_initialize.h"
#include <string.h>
#include <stdio.h>

#define MAX_STRING_SZ 512

static void main_replaceCats(char *inStr)
{
    /* Create emxArray's & other variables */
    emxArray_char_T *cstr = NULL;
    emxArray_char_T *cstrFinal = NULL;
    char outStr[MAX_STRING_SZ];
    int initCols = (int) strlen(inStr);
    int finCols;

    /* Initialize input & output emxArrays */
    cstr = emxCreateWrapper_char_T(inStr, 1, initCols);
    cstrFinal = emxCreate_char_T(1, 0);

    /* Call generated code on emxArrays */
    replaceCats(cstr, cstrFinal);

    /* Write output string data with null termination */
    finCols = cstrFinal->size[0]*cstrFinal->size[1];
    if (finCols >= MAX_STRING_SZ) {
        printf("Error: Output string exceeds max size.");
    }
}
```

```

    exit(-1);
}
memcpy(outStr, cstrFinal->data, finCols);
outStr[finCols]=0;

/* Print output */
printf("\nOld C string: %s \n", inStr);
printf( "New C string: %s \n", outStr);

/* Free the emxArray memory */
emxDestroyArray_char_T(cstrFinal);
}

int main(int argc, char *argv[])
{
    if (argc != 2 ) {
        printf("Error: Must provide exactly one input string, e.g.\n");
        printf(">replaceCats \"hello cat\"\n");
        exit(-1);
    }

    replaceCats_initialize();
    main_replaceCats(argv[1]);
    replaceCats_terminate();

    return 0;
}

```

## 生成可执行文件

生成可执行代码：

```

t = coder.typeof('a',[1 inf]);
codegen replaceCats -args {t} -config:exe main_replaceCats.c

```

Code generation successful.

在您的平台上测试可执行文件，并根据需要修改您的主文件。例如，在 Windows 上，您会得到以下输出：

```
C:\>replaceCats.exe "The pet owner called themselves a 'Catdad'"
```

```
Old C string: The pet owner called themselves a 'Catdad'
```

```
New C string: The pet owner called themselves a 'Velociraptordad'
```

## 另请参阅

[coder.typeof](#) | [coder.varsizes](#)

## 详细信息

- “Using Dynamic Memory Allocation for an Atoms Simulation”
- “Generate Code for Variable-Size Data”
- “多维数组”

## 使用示例主函数合并生成的代码

### 本节内容

- “使用示例主函数的工作流” (第 32-13 页)
- “使用 MATLAB Coder App 控制示例主函数生成” (第 32-13 页)
- “使用命令行界面控制示例主函数生成” (第 32-14 页)

当您编译使用生成的 C/C++ 代码的应用程序时，必须提供调用生成的代码的 C/C++ 主函数。

默认情况下，对于 C/C++ 源代码、静态库、动态库和可执行文件的代码生成，MATLAB Coder 会生成示例 C/C++ 主函数。此函数是可以帮助您将生成的 C/C++ 代码合并到应用程序中的模板。示例主函数声明和初始化数据，包括动态分配的数据。它会调用入口函数，但不使用入口函数返回的值。

MATLAB Coder 在编译文件夹的 **examples** 子文件夹中生成示例主函数的源文件和头文件。对于 C 代码生成，它会生成 **main.c** 和 **main.h** 文件。对于 C++ 代码生成，它会生成 **main.cpp** 和 **main.h** 文件。

不要修改 **examples** 子文件夹中的 **main.c** 和 **main.h** 文件。如果修改这些文件，则当您重新生成代码时，MATLAB Coder 不会重新生成示例主文件。它会发出警告，提示检测到生成文件发生了更改。在使用示例主函数之前，将示例主函数源文件和头文件复制到编译文件夹以外的某个位置。修改新位置的文件以满足您的应用程序的要求。


当您使用默认配置设置生成文件时，MATLAB Coder App 的 **packNGo** 函数和 **Package** 选项不会打包示例主函数源文件和头文件。要打包示例主文件，请配置代码生成以生成和编译示例主函数，生成代码，然后打包编译文件。

### 使用示例主函数的工作流

- 1 准备您的 MATLAB 代码以进行代码生成。
- 2 检查是否存在运行时问题。
- 3 确保启用了生成示例主函数的功能。
- 4 生成入口函数的 C/C++ 代码。
- 5 将示例主文件从 **examples** 子文件夹复制到另一个文件夹。
- 6 修改新文件夹中的示例主文件以满足应用程序的要求。
- 7 针对所需的平台部署示例主函数和生成的代码。
- 8 编译应用程序。

有关如何生成示例主函数并使用它编译可执行文件的示例，请参阅“在应用程序中使用示例 C 主函数” (第 32-15 页)。

### 使用 MATLAB Coder App 控制示例主函数生成

- 1 在 **Generate Code** 页上，要打开 **Generate** 对话框，请点击 **Generate** 箭头 .
- 2 在 **Generate** 对话框中，将 **Build Type** 设置为以下项之一：
  - “Source Code”
  - “Static Library”
  - “Dynamic Library”
  - “Executable”

- 3 点击 **More Settings**。
- 4 在 **All Settings** 选项卡上的 **Advanced** 下，将 **Generate example main** 设置为以下项之一：

设置为	目的
"Do not generate an example main function"	不生成示例 C/C++ 主函数
"Generate, but do not compile, an example main function" (默认值)	生成示例 C/C++ 主函数但不进行编译
"Generate and compile an example main function"	生成示例 C/C++ 主函数并进行编译

使用命令行界面控制示例主函数生成

- 1 为 'lib'、'dll' 或 'exe' 创建代码配置对象。例如：

```
cfg = coder.config('lib'); % or dll or exe
```

- 2 设置 **GenerateExampleMain** 属性。

设置为	目的
'DoNotGenerate'	不生成示例 C/C++ 主函数
'GenerateCodeOnly' (默认值)	生成示例 C/C++ 主函数但不进行编译
'GenerateCodeAndCompile'	生成示例 C/C++ 主函数并进行编译

例如：

```
cfg.GenerateExampleMain = 'GenerateCodeOnly';
```

另请参阅

详细信息

- “生成的示例 C/C++ 主函数的结构” (第 32-31 页)
- “指定 C/C++ 可执行文件的主函数” (第 27-10 页)

## 在应用程序中使用示例 C 主函数

此示例说明如何从 MATLAB 代码编译 C 可执行文件，该代码实现一个简单的 Sobel 滤波器以对图像执行边缘检测。该可执行文件从磁盘读取一个图像，应用 Sobel 滤波算法，然后保存修改后的图像。

此示例说明如何生成和修改可在编译可执行文件时使用的示例主函数。

### 本节内容

“前提条件” (第 32-15 页)  
 “创建文件夹并复制相关文件” (第 32-15 页)  
 “对图像运行 Sobel 滤波器” (第 32-17 页)  
 “生成并测试 MEX 函数” (第 32-18 页)  
 “为 sobel.m 生成示例主函数” (第 32-18 页)  
 “复制示例主文件” (第 32-21 页)  
 “修改生成的示例主函数” (第 32-21 页)  
 “生成 Sobel 滤波器应用程序” (第 32-29 页)  
 “运行 Sobel 滤波器应用程序” (第 32-29 页)  
 “显示生成的图像” (第 32-29 页)

### 前提条件

要完成此示例，请安装以下产品：

- MATLAB
- MATLAB Coder
- C 编译器（对于大多数平台，MATLAB 随附提供了一个默认 C 编译器）。要查看支持的编译器列表，请参阅 [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/)。

可以使用 `mex -setup` 更改默认编译器。请参阅“更改默认编译器”。

### 创建文件夹并复制相关文件

您在此示例中使用的文件包含：

文件名	文件类型	描述
sobel.m	函数代码	Sobel 滤波算法的 MATLAB 实现。 <code>sobel.m</code> 接受图像（表示为双精度矩阵）和阈值作为输入。该算法检测图像中的边缘（基于阈值）。 <code>sobel.m</code> 返回修改后且显示边缘的图像。
hello.jpg	图像文件	Sobel 滤波器修改的图像。

### sobel.m 文件的内容

```
function edgeImage = sobel(originalImage, threshold) %#codegen
```

```
% edgeImage = sobel(originalImage, threshold)
% Sobel edge detection. Given a normalized image (with double values)
% return an image where the edges are detected w.r.t. threshold value.

assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = [1 2 1; 0 0 0; -1 -2 -1];
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k, 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

hello.jpg 的内容



要将示例文件复制到本地工作文件夹，请执行下列操作：

- 1 创建一个本地工作文件夹。例如，`c:\coder\edge_detection`。
- 2 导航到该工作文件夹。
- 3 将文件 `sobel.m` 和 `hello.jpg` 从示例文件夹 `sobel` 复制到工作文件夹。

```
copyfile(fullfile(docroot, 'toolbox', 'coder', 'examples', 'sobel'))
```



## 对图像运行 Sobel 滤波器

- 1 将原始图像读入 MATLAB 矩阵并显示该图像。

```
im = imread('hello.jpg');
```

- 2 显示该图像以便与 Sobel 滤波器结果进行比较。

```
image(im);
```



- 3 Sobel 滤波算法对灰度图像进行运算。使用归一化值（0.0 代表黑色、1.0 代表白色），将彩色图像转换为等效的灰度图像。

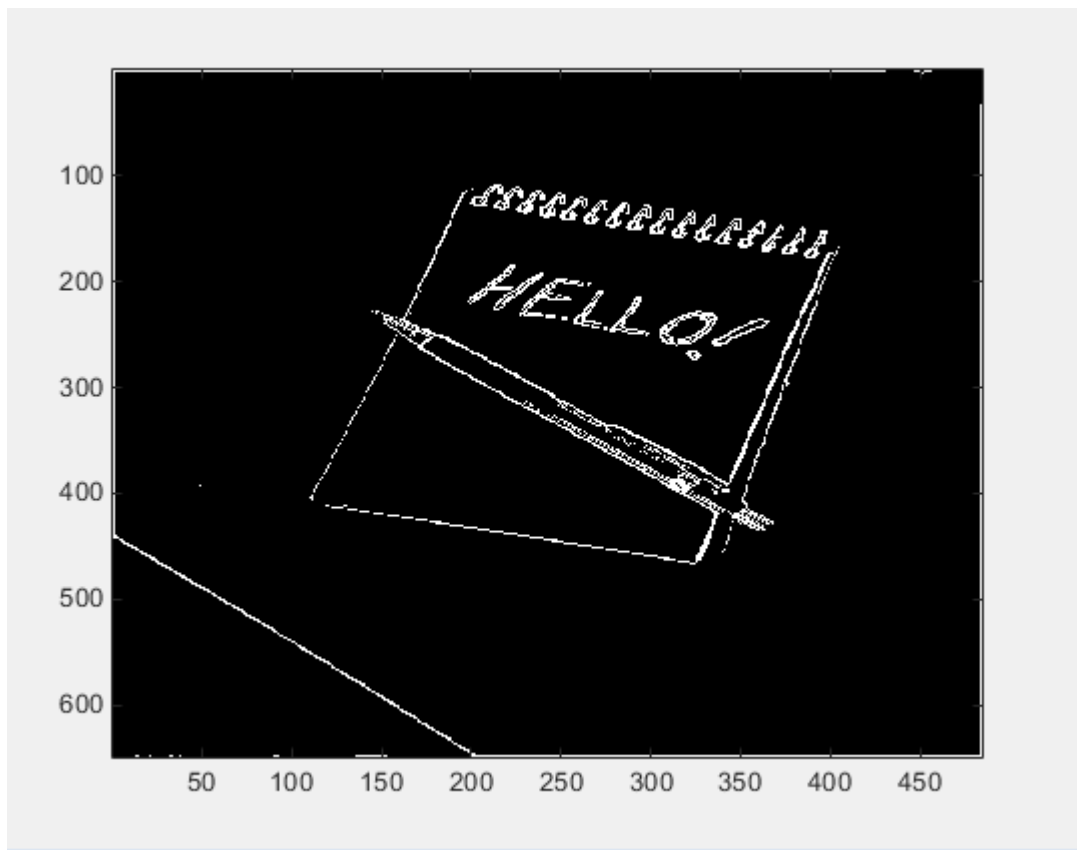
```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
```

- 4 要运行针对 Sobel 滤波器的 MATLAB 函数，请将灰度图像矩阵 **gray** 和阈值传递给函数 **sobel**。此示例使用 0.7 作为阈值。

```
edgeIm = sobel(gray, 0.7);
```

- 5 要显示修改后的图像，请使用 **repmat** 函数重新格式化矩阵 **edgeIm**，以便将其传递给 **image** 命令。

```
im3 = repmat(edgeIm, [1 1 3]);  
image(im3);
```



## 生成并测试 MEX 函数

- 1 要测试生成的代码是否在功能上等效于原始 MATLAB 代码并且不会发生运行时错误，请生成一个 MEX 函数。

`codegen -report sobel`

`codegen` 在当前工作文件夹中生成名为 `sobel_mex` 的 MEX 函数。

- 2 要运行针对 Sobel 滤波器的 MEX 函数，请将灰度图像矩阵 `gray` 和阈值传递给函数 `sobel_mex`。此示例使用 0.7 作为阈值。

```
edgeImMex = sobel_mex(gray, 0.7);
```

- 3 要显示修改后的图像，请使用 `repmat` 函数重新格式化矩阵 `edgeImMex`，以便将其传递给 `image` 命令。

```
im3Mex = repmat(edgeImMex, [1 1 3]);
image(im3Mex);
```

此图像与使用 MATLAB 函数创建的图像相同。

## 为 `sobel.m` 生成示例主函数

虽然您可以针对您的应用程序编写自定义主函数，但示例主函数提供了一个模板来帮助您合并生成的代码。

要为 Sobel 滤波器生成示例主函数，请执行下列操作：

- 1 为 C 静态库创建一个配置对象。

```
cfg = coder.config('lib');
```

对于 C/C++ 源代码、静态库、动态库和可执行文件的配置对象，GenerateExampleMain 的设置可控制示例主函数的生成。默认情况下，该设置设为 'GenerateCodeOnly'，即生成示例主函数但不编译它。对于此示例，请勿更改 GenerateExampleMain 设置的值。

- 2 使用该配置对象生成一个 C 静态库。

```
codegen -report -config cfg sobel
```

生成的静态库文件位于文件夹 codegen/lib/sobel 中。示例主文件位于子文件夹 codegen/lib/sobel/examples 中。

### 示例主文件 main.c 的内容

```
/*
 * File: main.c
 *
 */

/*****
 * This automatically generated example C main file shows how to call
 * entry-point functions that MATLAB Coder generated. You must customize
 * this file for your application. Do not modify this file directly.
 * Instead, make a copy of this file, modify it, and integrate it into
 * your development environment.
 */
/* This file initializes entry-point function arguments to a default
 * size and value before calling the entry-point functions. It does
 * not store or use any values returned from the entry-point functions.
 * If necessary, it does pre-allocate memory for returned values.
 * You can use this file as a starting point for a main function that
 * you can deploy in your application.
 */
/* After you copy the file, and before you deploy it, you must make the
 * following changes:
 * * For variable-size function arguments, change the example sizes to
 * the sizes that your application requires.
 * * Change the example values of function arguments to the values that
 * your application requires.
 * * If the entry-point functions return values, store these values or
 * otherwise use them as required by your application.
 */
*****/

/* Include Files */
#include "main.h"
#include "sobel.h"
#include "sobel_emxAPI.h"
#include "sobel_terminate.h"
#include "sobel_types.h"

/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(void);
```

```

static double argInit_real_T(void);

static void main_sobel(void);

/* Function Definitions */
/*
 * Arguments : void
 * Return Type : emxArray_real_T *
 */
static emxArray_real_T *argInit_d1024xd1024_real_T(void)
{
    emxArray_real_T *result;
    double *result_data;
    int idx0;
    int idx1;
    /* Set the size of the array.
    Change this size to the value that the application requires. */
    result = emxCreate_real_T(2, 2);
    result_data = result->data;
    /* Loop over the array to initialize each element. */
    for (idx0 = 0; idx0 < result->size[0U]; idx0++) {
        for (idx1 = 0; idx1 < result->size[1U]; idx1++) {
            /* Set the value of the array element.
            Change this value to the value that the application requires. */
            result_data[idx0 + result->size[0] * idx1] = argInit_real_T();
        }
    }
    return result;
}

/*
 * Arguments : void
 * Return Type : double
 */
static double argInit_real_T(void)
{
    return 0.0;
}

/*
 * Arguments : void
 * Return Type : void
 */
static void main_sobel(void)
{
    emxArray_real_T *originalImage;
    emxArray_uint8_T *edgeImage;
    emxInitArray_uint8_T(&edgeImage, 2);
    /* Initialize function 'sobel' input arguments. */
    /* Initialize function input argument 'originalImage'. */
    originalImage = argInit_d1024xd1024_real_T();
    /* Call the entry-point 'sobel'. */
    sobel(originalImage, argInit_real_T(), edgeImage);
    emxDestroyArray_uint8_T(edgeImage);
    emxDestroyArray_real_T(originalImage);
}

```

```

/*
 * Arguments   : int argc
 *               char **argv
 * Return Type : int
 */
int main(int argc, char **argv)
{
    (void)argc;
    (void)argv;
    /* The initialize function is being called automatically from your entry-point
     * function. So, a call to initialize is not included here. */
    /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
    main_sobel();
    /* Terminate the application.
     You do not need to do this more than one time. */
    sobel_terminate();
    return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */

```

## 复制示例主文件

请不要修改 `examples` 子文件夹中的 `main.c` 和 `main.h` 文件。如果修改这些文件，则当您重新生成代码时，MATLAB Coder 不会重新生成示例主文件。它会发出警告，提示检测到生成文件发生了更改。

将文件 `main.c` 和 `main.h` 从文件夹 `codegen/lib/sobel/examples` 复制到另一个位置。对于此示例，请将文件复制到当前工作文件夹中。修改新位置中的文件。

## 修改生成的示例主函数

- “修改 main 函数”（第 32-22 页）
- “修改初始化函数 `arglinit_d1024xd1024_real_T`”（第 32-23 页）
- “编写 `savelmage` 函数”（第 32-24 页）
- “修改 `main_sobel` 函数”（第 32-25 页）
- “修改函数声明”（第 32-26 页）
- “修改 include 文件”（第 32-26 页）
- “修改后的 `main.c` 文件的内容”（第 32-27 页）

示例主函数声明数据（包括动态分配的数据）并将数据初始化为零值。它调用入口函数（其参数设置为零值），但它不使用从入口函数返回的值。

C 主函数必须满足您的应用程序的要求。此示例修改示例主函数以满足 Sobel 滤波器应用程序的要求。

此示例修改文件 `main.c` 以便 Sobel 滤波器应用程序：

- 从二进制文件中读入灰度图像。

- 应用 Sobel 滤波算法。
- 将修改后的图像保存为二进制文件。

### 修改 main 函数

将 main 函数修改为：

- 接受包含灰度图像数据的文件和阈值作为输入参数。
- 使用灰度图像数据流的地址和阈值作为输入参数调用函数 `main_sobel`。

在函数 main 中：

- 1 删除 `(void)argc` 和 `(void)argv` 声明。
- 2 声明变量 `filename` 以保存包含灰度图像数据的二进制文件的名称。  
`const char *filename;`
- 3 声明变量 `threshold` 以保存阈值。  
`double threshold;`
- 4 声明变量 `fd` 以保存应用程序从 `filename` 读入的灰度图像数据的地址。  
`FILE *fd;`
- 5 添加一个用于检查三个参数的 if 语句。  

```
if (argc != 3) {
    printf("Expected 2 arguments: filename and threshold\n");
    exit(-1);
}
```
- 6 将包含灰度图像数据的文件的输入参数 `argv[1]` 赋给 `filename`。  
`filename = argv[1];`
- 7 将阈值的输入参数 `argv[2]` 赋给 `threshold`，从而将输入从字符串转换为双精度数值。  
`threshold = atof(argv[2]);`
- 8 打开包含其名称在 `filename` 中指定的灰度图像数据的文件。将数据流的地址赋给 `fd`。  
`fd = fopen(filename, "rb");`
- 9 要验证可执行文件可以打开 `filename`，请编写一个 if 语句，其功能是在 `fd` 的值为 `NULL` 时退出程序。  

```
if (fd == NULL) {
    exit(-1);
}
```
- 10 将对 `main_sobel` 的函数调用替换为带输入参数 `fd` 和 `threshold` 调用 `main_sobel`。  
`main_sobel(fd, threshold);`
- 11 调用 `sobel_terminate` 后关闭灰度图像文件。  
`fclose(fd);`

### 修改后的 main 函数

```
int main(int argc, char **argv)
{
```

```

const char *filename;
double threshold;
FILE *fd;
if (argc != 3) {
    printf("Expected 2 arguments: filename and threshold\n");
    exit(-1);
}
filename = argv[1];
threshold = atof(argv[2]);

fd = fopen(filename, "rb");
if (fd == NULL) {
    exit(-1);
}

main_sobel(fd, threshold);
fclose(fd);
sobel_terminate();

return 0;
}

```

### 修改初始化函数 `argInit_d1024xd1024_real_T`

在示例主文件中，函数 `argInit_d1024xd1024_real_T` 为传递给 Sobel 滤波器的图像创建一个动态分配的可变大小数组 (`emxArray`)。此函数将 `emxArray` 初始化为默认大小，并将 `emxArray` 的元素初始化为 0。它返回初始化后的 `emxArray`。

对于 Sobel 滤波器应用程序，请修改该函数以将灰度图像数据从二进制文件读入 `emxArray` 中。

在函数 `argInit_d1024xd1024_real_T` 中：

- 1 将输入参数 `void` 替换为参数 `FILE *fd`。此变量指向该函数读入的灰度图像数据。

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
```

- 2 更改变量 `idx0` 和 `idx1` 的值，以匹配灰度图像矩阵 `gray` 的维度。这些变量保留 `argInit_d1024xd1024_real_T` 创建的 `emxArray` 的维度的大小值。

```
int idx0 = 484;
int idx1 = 648;
```

MATLAB 以列优先格式存储矩阵数据，而 C 以行优先格式存储矩阵数据。请相应地声明维度。

- 3 将 `emxCreate_real_T` 函数的值更改为图像大小。

```
result = emxCreate_real_T(484, 648);
```

- 4 定义变量 `element` 以保存从灰度图像数据读入的值。

```
double element;
```

- 5 通过向内部 `for` 循环添加 `fread` 命令，更改 `for` 循环构造以将数据点从归一化图像读入 `element` 中。

```
fread(&element, 1, sizeof(element), fd);
```

- 6 在 `for` 循环内，将 `element` 指定为 `emxArray` 数据的值集合。

```
result->data[idx0 + result->size[0] * idx1] = element;
```

**修改后的初始化函数 `argInit_d1024xd1024_real_T`**

```
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
    emxArray_real_T *result;
    double *result_data;
    int idx0 = 484;
    int idx1 = 648;
    /* Set the size of the array.
    Change this size to the value that the application requires. */
    double element;
    result = emxCreate_real_T(484, 648);
    result_data = result->data;
    /* Loop over the array to initialize each element. */
    for (idx0 = 0; idx0 < result->size[0U]; idx0++) {
        for (idx1 = 0; idx1 < result->size[1U]; idx1++) {
            /* Set the value of the array element.
            Change this value to the value that the application requires. */
            fread(&element, 1, sizeof(element), fd);
            result_data[idx0 + result->size[0] * idx1] = element;
        }
    }
    return result;
}
```

**编写 `saveImage` 函数**

MATLAB 函数 `sobel.m` 与 MATLAB 数组交互，而 Sobel 滤波器应用程序与二进制文件交互。

要将用 Sobel 滤波算法修改后的图像保存为二进制文件，请创建函数 `saveImage`。函数 `saveImage` 将来自 `emxArray` 的数据写入二进制文件。它使用的构造类似于函数 `argInit_d1024xd1024_real_T` 所使用的构造。

在文件 `main.c` 中：

- 1 定义函数 `saveImage`，它将 `emxArray edgeImage` 的地址作为输入，输出类型为 `void`。

```
static void saveImage(emxArray_uint8_T *edgeImage)
{
}
```

- 2 定义变量 `idx0` 和 `idx1`，就像在函数 `argInit_d1024xd1024_real_T` 中定义它们一样。

```
int idx;
int idx1;
```

- 3 定义变量 `element` 以存储从 `emxArray` 读取的数据。

```
uint8_T element;
```

- 4 打开二进制文件 `edge.bin` 以写入修改后的图像。将 `edge.bin` 的地址赋给 `FILE *fd`。

```
FILE *fd = fopen("edge.bin", "wb");
```

- 5 要验证可执行文件可以打开 `edge.bin`，请编写一个 `if` 语句，其功能是在 `fd` 的值为 `NULL` 时退出程序。

```
if (fd == NULL) {
    exit(-1);
}
```



- 6 编写一个嵌套的 for 循环构造，该构造类似于函数 `argInit_d1024xd1024_real_T` 中的构造。

```
for (idx0 = 0; idx0 < edgeImage->size[0U]; idx0++)
{
    for (idx1 = 0; idx1 < edgeImage->size[1U]; idx1++)
    {
    }
}
```

- 7 在内部 for 循环内，将来自修改后的图像数据的值赋给 `element`。

```
element = edgeImage->data[idx0 + edgeImage->size[0] * idx1];
```

- 8 在为 `element` 赋值后，将来自 `element` 的值写入文件 `edge.bin`。

```
fwrite(&element, 1, sizeof(element), fd);
```

- 9 在 for 循环构造后，关闭 `fd`。

```
fclose(fd);
```

### saveImage 函数

```
static void saveImage(emxArray_uint8_T *edgeImage)
{
    int idx0;
    int idx1;
    uint8_T element;

    FILE *fd = fopen("edge.bin", "wb");
    if (fd == NULL) {
        exit(-1);
    }
    /* Loop over the array to save each element. */
    for (idx0 = 0; idx0 < edgeImage->size[0U]; idx0++) {
        for (idx1 = 0; idx1 < edgeImage->size[1U]; idx1++) {
            element = edgeImage->data[idx0 + edgeImage->size[0] * idx1];
            fwrite(&element, 1, sizeof(element), fd);
        }
    }
    fclose(fd);
}
```

### 修改 main\_sobel 函数

在示例主函数中，函数 `main_sobel` 为灰度图像和修改后的图像的数据创建 `emxArray`。它调用函数 `argInit_d1024xd1024_real_T` 以初始化灰度图像的 `emxArray`。`main_sobel` 将 `emxArray` 和初始化函数 `argInit_real_T` 返回的阈值 0 传递给函数 `sobel`。当函数 `main_sobel` 结束时，它会丢弃函数 `sobel` 的结果。

对于 Sobel 滤波器应用程序，请将函数 `main_sobel` 修改为：

- 将灰度图像数据的地址和阈值作为输入。
- 使用 `argInit_d1024xd1024_real_T` 从地址读取数据。
- 使用阈值 `threshold` 将数据传递给 Sobel 滤波算法。
- 使用 `saveImage` 保存结果。

在函数 `main_sobel` 中：

- 1 将函数的输入参数替换为 `FILE *fd` 和 `double threshold` 参数。  
`static void main_sobel(FILE *fd, double threshold)`
- 2 将输入参数 `fd` 传递给 `argInit_d1024xd1024_real_T` 的函数调用。  
`originalImage = argInit_d1024xd1024_real_T(fd);`
- 3 将对 `sobel` 的函数调用中的阈值输入替换为 `threshold`。  
`sobel(originalImage, threshold, edgeImage);`
- 4 调用函数 `sobel` 后, 带 `edgeImage` 输入调用函数 `saveImage`。  
`saveImage(edgeImage);`

### 修改后的 main\_sobel 函数

```
static void main_sobel(FILE *fd, double threshold)
{
    emxArray_real_T *originalImage;
    emxArray_uint8_T *edgeImage;
    emxInitArray_uint8_T(&edgeImage, 2);
    /* Initialize function 'sobel' input arguments. */
    /* Initialize function input argument 'originalImage'. */
    originalImage = argInit_d1024xd1024_real_T(fd);
    /* Call the entry-point 'sobel'. */
    sobel(originalImage, threshold, edgeImage);
    saveImage(edgeImage);

    emxDestroyArray_uint8_T(edgeImage);
    emxDestroyArray_real_T(originalImage);
}
```

### 修改函数声明

为了匹配您对函数定义所做的更改, 请对函数声明进行以下更改:

- 1 将函数 `*argInit_d1024xd1024_real_T` 的输入更改为 `FILE *fd`。  
`static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);`
- 2 将函数 `main_sobel` 的输入更改为 `FILE *fd` 和 `double threshold`。  
`static void main_sobel(FILE *fd, double threshold);`
- 3 添加函数 `saveImage`。  
`static void saveImage(emxArray_uint8_T *edgeImage);`

### 修改后的函数声明

```
/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
static void saveImage(emxArray_uint8_T *edgeImage);
static double argInit_real_T(void);
static void main_sobel(FILE *fd, double threshold);
```

### 修改 include 文件

对于在 `main.c` 中使用的输入/输出函数, 将头文件 `stdio.h` 添加到包含的文件列表中。

```
#include <stdio.h>
```

## 修改后的 include 文件

```
/* Include Files */
#include <stdio.h>

#include "sobel.h"
#include "main.h"
#include "sobel_terminate.h"
#include "sobel_emxAPI.h"
#include "sobel_initialize.h"
```

## 修改后的 main.c 文件的内容

### main.c

```
/*
 * File: main.c
 *
 */

/*****
 * This automatically generated example C main file shows how to call
 * entry-point functions that MATLAB Coder generated. You must customize
 * this file for your application. Do not modify this file directly.
 * Instead, make a copy of this file, modify it, and integrate it into
 * your development environment.
 */
/* This file initializes entry-point function arguments to a default
 * size and value before calling the entry-point functions. It does
 * not store or use any values returned from the entry-point functions.
 * If necessary, it does pre-allocate memory for returned values.
 * You can use this file as a starting point for a main function that
 * you can deploy in your application.
 */
/* After you copy the file, and before you deploy it, you must make the
 * following changes:
 * * For variable-size function arguments, change the example sizes to
 * the sizes that your application requires.
 * * Change the example values of function arguments to the values that
 * your application requires.
 * * If the entry-point functions return values, store these values or
 * otherwise use them as required by your application.
 */
*****/

/* Include Files */
#include <stdio.h>
#include "main.h"
#include "sobel.h"
#include "sobel_emxAPI.h"
#include "sobel_terminate.h"
#include "sobel_types.h"

/* Function Declarations */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd);
static void saveImage(emxArray_uint8_T *edgeImage);
static double argInit_real_T(void);
static void main_sobel(FILE *fd, double threshold);

/* Function Definitions */
/*
 * Arguments : void
 * Return Type : emxArray_real_T
 */
static emxArray_real_T *argInit_d1024xd1024_real_T(FILE *fd)
{
    emxArray_real_T *result;
    double *result_data;
    int idx0 = 484;
    int idx1 = 648;
    /* Set the size of the array.
     * Change this size to the value that the application requires.
     */
    double element;
```

```

    result = emxCreate_real_T(484, 648);
    result_data = result->data;
    /* Loop over the array to initialize each element. */
    for (idx0 = 0; idx0 < result->size[0U]; idx0++) {
        for (idx1 = 0; idx1 < result->size[1U]; idx1++) {
            /* Set the value of the array element.
            Change this value to the value that the application requires. */
            fread(&element, 1, sizeof(element), fd);
            result_data[idx0 + result->size[0] * idx1] = element;
        }
    }
    return result;
}

static void saveImage(emxArray_uint8_T *edgeImage)
{
    int idx0;
    int idx1;
    uint8_T element;

    FILE *fd = fopen("edge.bin", "wb");
    if (fd == NULL) {
        exit(-1);
    }
    /* Loop over the array to save each element. */
    for (idx0 = 0; idx0 < edgeImage->size[0U]; idx0++) {
        for (idx1 = 0; idx1 < edgeImage->size[1U]; idx1++) {
            element = edgeImage->data[idx0 + edgeImage->size[0] * idx1];
            fwrite(&element, 1, sizeof(element), fd);
        }
    }
    fclose(fd);
}

/*
 * Arguments : void
 * Return Type : double
 */
static double argInit_real_T(void)
{
    return 0.0;
}

/*
 * Arguments : void
 * Return Type : void
 */
static void main_sobel(FILE *fd, double threshold)
{
    emxArray_real_T *originalImage;
    emxArray_uint8_T *edgeImage;
    emxInitArray_uint8_T(&edgeImage, 2);
    /* Initialize function 'sobel' input arguments. */
    /* Initialize function input argument 'originalImage'. */
    originalImage = argInit_d1024xd1024_real_T(fd);
    /* Call the entry-point 'sobel'. */
    sobel(originalImage, threshold, edgeImage);
    saveImage(edgeImage);

    emxDestroyArray_uint8_T(edgeImage);
    emxDestroyArray_real_T(originalImage);
}

/*
 * Arguments : int argc
 *             char **argv
 * Return Type : int
 */
int main(int argc, char **argv)
{
    const char *filename;
    double threshold;
    FILE *fd;
    if (argc != 3) {
        printf("Expected 2 arguments: filename and threshold\n");
        exit(-1);
    }
    filename = argv[1];
    threshold = atof(argv[2]);

    fd = fopen(filename, "rb");

```

```

if (fd == NULL) {
    exit(-1);
}

/* The initialize function is being called automatically from your entry-point
 * function. So, a call to initialize is not included here. */
/* Invoke the entry-point functions.
You can call entry-point functions multiple times. */
main_sobel(fd, threshold);
fclose(fd);
/* Terminate the application.
You do not need to do this more than one time. */
sobel_terminate();
return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */

```

## 生成 Sobel 滤波器应用程序

- 1 导航到工作文件夹（如果当前不在该文件夹中）。
- 2 为 C 独立可执行文件创建一个配置对象。

```
cfg = coder.config('exe');
```

- 3 使用该配置对象和修改后的主函数为 Sobel 滤波器生成 C 独立可执行文件。

```
codegen -report -config cfg sobel main.c main.h
```

默认情况下，如果在 Windows 平台上运行 MATLAB，则会在当前工作文件夹中生成可执行文件 `sobel.exe`。如果在 Windows 以外的平台上运行 MATLAB，则文件扩展名是该平台的对应扩展名。默认情况下，为可执行文件生成的代码位于文件夹 `codegen/exe/sobel` 中。

## 运行 Sobel 滤波器应用程序

- 1 创建 MATLAB 矩阵 `gray`（如果它当前不在 MATLAB 工作区中）：

```
im = imread('hello.jpg');
```

```
gray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + 0.1140 * double(im(:,:,3)))/255;
```

- 2 使用 `fopen` 和 `fwrite` 命令将矩阵 `gray` 写入一个二进制文件。应用程序读入此二进制文件。

```

fid = fopen('gray.bin', 'w');
fwrite(fid, gray, 'double');
fclose(fid);

```

- 3 运行可执行文件，将文件 `gray.bin` 和阈值 0.7 传递给它。

要在 Windows 平台上运行 MATLAB 中的示例，请执行以下命令：

```
system('sobel.exe gray.bin 0.7');
```

可执行文件会生成 `edge.bin` 文件。

## 显示生成的图像

- 1 使用 `fopen` 和 `fread` 命令将文件 `edge.bin` 读入 MATLAB 矩阵 `edgeImExe`。

```
fd = fopen('edge.bin', 'r');  
edgeImExe = fread(fd, size(gray), 'uint8');  
fclose(fd);
```

- 2 将矩阵 `edgeImExe` 传递给函数 `repmat` 并显示图像。

```
im3Exe = repmat(edgeImExe, [1 1 3]);  
image(im3Exe);
```

图像与来自 MATLAB 和 MEX 函数的图像匹配。

## 另请参阅

## 相关示例

- “生成的示例 C/C++ 主函数的结构” (第 32-31 页)
- “使用示例主函数合并生成的代码” (第 32-13 页)

## 生成的示例 C/C++ 主函数的结构

本节内容
“文件 main.c 或 main.cpp 的内容” (第 32-31 页)
“文件 main.h 的内容” (第 32-33 页)

当您编译使用生成的 C/C++ 代码的应用程序时，必须提供调用生成的代码的 C/C++ 主函数。

默认情况下，对于 C/C++ 源代码、静态库、动态库和可执行文件的代码生成，MATLAB Coder 会生成示例 C/C++ 主函数。此函数是可以帮助您将生成的 C/C++ 代码合并到应用程序中的模板。示例主函数声明和初始化数据，包括动态分配的数据。它会调用入口函数，但不使用入口函数返回的值。要使用示例主函数，请将示例主函数源文件和头文件复制到编译文件夹以外的某个位置，然后在新位置修改文件以满足您的应用程序的要求。

MATLAB Coder 在编译文件夹的 **examples** 子文件夹中生成示例主函数的源文件和头文件。对于 C 代码生成，它会生成 **main.c** 和 **main.h** 文件。对于 C++ 代码生成，它会生成 **main.cpp** 和 **main.h** 文件。

### 文件 main.c 或 main.cpp 的内容

对于示例主函数源文件 **main.c** 或 **main.cpp**，MATLAB Coder 会生成以下各节内容：

- “Include 文件” (第 32-31 页)
- “函数声明” (第 32-31 页)
- “参数初始化函数” (第 32-31 页)
- “入口函数” (第 32-32 页)
- “主函数” (第 32-32 页)

默认情况下，MATLAB Coder 还会在示例主函数源文件中生成注释，这些注释可帮助您修改示例主函数以在您的应用程序中使用。

#### Include 文件

本节包含调用不在示例主函数源文件中的代码所需的头文件。如果在修改示例主函数源文件时调用外部函数，请包含任何其他必需的头文件。

#### 函数声明

本节声明在示例主函数源文件中定义的参数初始化和入口函数的函数原型。修改函数原型以匹配您在函数定义中所做的修改。为在示例主函数源文件中定义的函数声明新的函数原型。

#### 参数初始化函数

本节为入口函数用作参数的每种数据类型定义初始化函数。参数初始化函数将参数的大小初始化为默认值，将数据的值初始化为零。然后，该函数返回经过初始化的数据。请更改这些大小和数据值以满足您的应用程序的要求。

对于维度大小为 **<dimSizes>** 且使用 MATLAB C/C++ 数据类型 **<baseType>** 的参数，示例主函数源文件定义名称为 **argInit\_<dimSizes>\_<baseType>** 的初始化函数。例如，对于使用 MATLAB 双精度类型的数据的 5×5 数组，示例主函数源文件将定义参数初始化函数 **argInit\_5x5\_real\_T**。

MATLAB Coder 会更改参数初始化函数的名称，如下所示：

- 如果有任何维度的大小是可变的，则 MATLAB Coder 会将这些维度的大小指定为 `d<maxSize>`，其中 `<maxSize>` 是该维度的最大大小。例如，对于具有 MATLAB 双精度类型数据的数组，如果其第一个维度的静态大小为 2，第二个维度为可变大小且最大为 10，则示例主函数源文件将定义参数初始化函数 `argInit_2xd10_real_T`。
- 如果有无界维度，则 MATLAB Coder 会将这些维度的大小指定为 `Unbounded`。
- 如果初始化函数的返回类型为 `emxArray`，则 MATLAB Coder 会将函数定义为返回指向 `emxArray` 的指针。
- 如果初始化函数名称的长度超过了在配置设置中为函数名称设置的最大字符数，则 MATLAB Coder 会在函数名称的前面添加一个标识符。然后，MATLAB Coder 会将函数名称截断为标识符长度允许的最大字符数。

**注意** 默认情况下，对于生成的标识符，允许的最大字符数为 31。要使用 MATLAB Coder App 将该值指定为设置的最大标识符长度，请在代码生成设置的 **Code Appearance** 选项卡上选择 **Maximum identifier length** 值。要使用命令行界面将该值指定为设置的最大标识符长度，请更改 `MaxIdLength` 配置对象设置的值。

## 入口函数

本节为每个 MATLAB 入口函数定义函数。对于 MATLAB 函数 `foo.m`，示例主函数源文件将定义入口函数 `main_foo`。此函数会创建变量并调用 C/C++ 源函数 `foo.c` 或 `foo.cpp` 所需的数据初始化函数。它调用此 C/C++ 源函数，但不返回结果。修改 `main_foo`，使其能根据您的应用程序的需要接受输入和返回输出。

## 主函数

本节定义执行以下操作的 `main` 函数：

- 如果您的输出语言为 C，则它将声明并命名变量 `argc` 和 `argv`，但会将其转换为 `void` 类型。如果您的输出语言为 C++，则生成的示例主函数将声明变量 `argc` 和 `argv`，但不会命名这些变量。
- 对每个入口函数执行一次调用。
- 调用终止函数 `foo_terminate`，该函数是针对为代码生成而声明的第一个 MATLAB 入口函数 `foo` 命名的。即使在函数 `main` 中调用多个入口函数，也只需调用一次终止函数。
- 返回零。

默认情况下，示例 `main` 函数不调用初始化函数 `foo_initialize`。代码生成器会在生成的 C/C++ 入口函数开始处包含一个对初始化函数的调用。生成的代码中还包括一些检查，以确保初始化函数只被自动调用一次，即使有多个入口函数也是如此。

您可以选择在生成的入口函数中不包含对初始化函数的调用。要实现此目的，请执行以下操作之一：

- 在 `coder.CodeConfig` 或 `coder.EmbeddedCodeConfig` 对象中，将 `RunInitializeFcn` 设置为 `false`。
- 在 MATLAB Coder App 中，在 **All Settings** 选项卡上，将 **Automatically run the initialize function** 设置为 “No”。

如果您照此操作，示例 `main` 函数将会包括对初始化函数 `foo_initialize` 的调用。

请参阅 “Use Generated Initialize and Terminate Functions”。

修改函数 `main`，包括 `main` 和入口函数的输入与输出，以满足您的应用程序的要求。



## 文件 main.h 的内容

对于示例主函数头文件 **main.h**，MATLAB Coder 会生成以下内容：

- “Include Guard” （第 32-33 页）
- “Include 文件” （第 32-33 页）
- “函数声明” （第 32-33 页）

默认情况下，MATLAB Coder 还会在 **main.h** 中生成注释，这些注释可帮助您修改示例主函数以在您的应用程序中使用。

### Include Guard

**main.h** 使用 include guard 来防止文件的内容被多次包含。include guard 包含 `#ifndef` 构造内的 include 文件和函数声明。

### Include 文件

**main.h** 包含调用未在本函数内定义的代码所需的头文件。

### 函数声明

**main.h** 声明在示例主函数源文件 **main.c** 或 **main.cpp** 中定义的主函数的函数原型。

## 另请参阅

### 相关示例

- “使用示例主函数合并生成的代码” （第 32-13 页）
- “在应用程序中使用示例 C 主函数” （第 32-15 页）

### 详细信息

- “Mapping MATLAB Types to Types in Generated Code”
- “Use Generated Initialize and Terminate Functions”

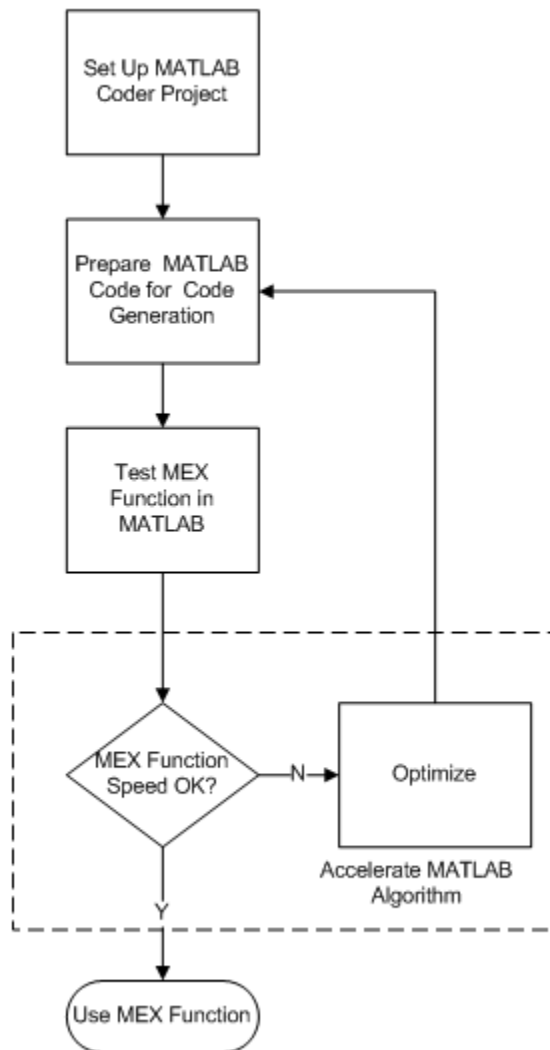


# 加速 MATLAB 算法

---

- “加速 MATLAB 算法的工作流” (第 33-2 页)
- “使用 MATLAB 探查器探查 MEX 函数” (第 33-3 页)
- “使用并行 for 循环 (parfor) 的算法加速” (第 33-7 页)

## 加速 MATLAB 算法的工作流



### 另请参阅

- “设置 MATLAB Coder 工程” (第 24-2 页)
- “准备 MATLAB 代码以用于代码生成的工作流” (第 25-2 页)
- “Workflow for Testing MEX Functions in MATLAB”
- “Modifying MATLAB Code for Acceleration”

## 使用 MATLAB 探查器探查 MEX 函数

您可以使用 MATLAB 探查器来探查 MATLAB Coder 生成的 MEX 函数的执行时间。所生成代码的探查文件显示对应 MATLAB 函数的调用次数和运行每行代码所花费的时间。使用探查器可识别产生耗时最长的生成代码的 MATLAB 代码行。这些信息可以帮助您在开发周期的早期确定和更正性能问题。有关 MATLAB 探查器的详细信息，请参阅 **profile** 和“探查您的代码以改善性能”。

MATLAB Online 不支持探查器的图形界面。

### 生成 MEX 探查文件

您可以将 MATLAB 探查器与生成的 MEX 函数结合使用。如果您有调用您的 MATLAB 函数的测试文件，也可以一步生成 MEX 函数并对其进行探查。您可以在命令行上或 MATLAB Coder App 中执行这些操作。

要将探查器与生成的 MEX 函数结合使用，请执行以下操作：

- 1 通过将配置对象属性 **EnableMexProfiling** 设置为 **true** 启用 MEX 探查。

您也可以将 **codegen** 与 **-profile** 选项结合使用。

MATLAB Coder App 中的等效设置是 **Generate** 步骤中的 **Enable execution profiling**。

- 2 生成 MEX 文件 **MyFunction\_mex**。
- 3 运行 MATLAB 探查器，并查看探查摘要报告，该报告将在单独窗口中打开。

```
profile on;
MyFunction_mex;
profile viewer;
```

确保您没有更改或移动原始 MATLAB 文件 **MyFunction.m**。否则，探查器不会将 **MyFunction\_mex** 视为探查对象。

如果您有调用您的 MATLAB 函数的测试文件 **MyFunctionTest.m**，您可以：

- 通过将 **codegen** 与 **-test** 和 **-profile** 选项结合使用，一步生成 MEX 函数并对其进行探查。如果您以前打开过 MATLAB 探查器，请在结合使用这两个选项之前将其关闭。

```
codegen MyFunction -test MyFunctionTest -profile
```

- 通过在 App 的 **Verify** 步骤中选择 **Enable execution profiling** 来探查 MEX 函数。如果您以前打开过 MATLAB 探查器，请在执行此操作之前将其关闭。

### 示例

您使用探查器来识别产生耗时最长的生成代码的函数或 MATLAB 代码行。下面是 MATLAB 函数的示例，该函数在一个代码行中将其输入矩阵 **A** 和 **B** 的表示从行优先布局转换为列优先布局。对于大型矩阵，这种转换需要很长的执行时间。通过修改特定代码行来避免这种转换会使函数更加高效。

以 MATLAB 函数为例：

```
function [y] = MyFunction(A,B) %#codegen

% Generated code uses row-major representation of matrices A and B
coder.rowMajor;
```

```

length = size(A,1);

% Summing absolute values of all elements of A and B by traversing over the
% matrices row by row
sum_abs = 0;
for row = 1:length
    for col = 1:length
        sum_abs = sum_abs + abs(A(row,col)) + abs(B(row,col));
    end
end

% Calling external C function 'foo.c' that returns the sum of all elements
% of A and B
sum = 0;
sum = coder.ceval('foo',coder.ref(A),coder.ref(B),length);

% Returning the difference of sum_abs and sum
y = sum_abs - sum;
end

```

为该函数生成的代码使用方阵 A 和 B 的行优先表示。该代码首先通过逐行遍历矩阵来计算 `sum_abs` (A 和 B 的所有元素的绝对值之和)。该算法针对以行优先布局方式表示的矩阵进行了优化。然后，代码使用 `coder.ceval` 调用外部 C 函数 `foo.c`：

```

#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double *A, double *B, double length)
{
    int i,j,s;
    double sum = 0;
    s = (int)length;

    /*Summing all the elements of A and B*/
    for(i=0;i<s*s;i++)
    {
        sum += A[i] + B[i];
    }
    return(sum);
}

```

对应的 C 头文件 `foo.h` 是：

```

#include "rtwtypes.h"

double foo(double *A, double *B, double length);

```

`foo.c` 返回变量 `sum`，它是 A 和 B 的所有元素的总和。函数 `foo.c` 的性能与矩阵 A 和 B 以行优先布局还是列优先布局表示无关。`MyFunction` 返回 `sum_abs` 和 `sum` 的差值。

您可以针对大型输入矩阵 A 和 B 测量 `MyFunction` 的性能，然后进一步对该函数进行优化：

- 1 对 `MyFunction` 启用 MEX 探查并生成 MEX 代码。对两个大型随机矩阵 A 和 B 运行 `MyFunction_mex`。查看探查摘要报告。

```

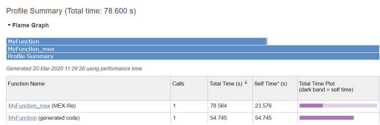
A = rand(20000);
B = rand(20000);

```

`codegen MyFunction -args {A,B} foo.c foo.h -profile`

`profile on;`  
`MyFunction_mex(A,B);`  
`profile viewer;`

将打开单独的窗口，显示探查摘要报告。



探查摘要报告显示 MEX 文件及其子文件（为原始 MATLAB 函数生成的代码）的总时间和自用时间。

- 2 在 Function Name 下，点击第一个链接查看为 **MyFunction** 生成的代码的探查详细信息报告。您可以查看耗时最多的代码行：

\*Lines that take the most time

Line Number	Code	Calls	Total Time (s)	% Time	Time Plot
12	sum_val = sum_val + size(A(1:row,col)) * size(B);	40000000	16.181	35.0%	
13	end	40000000	16.048	34.1%	
18	sum = coder.ceval('foo',coder.ref(S),coder.ref(S));	1	16.914	30.0%	
19	end	20000	0.001	0.0%	
21	for col = 1:length	20000	0.001	0.0%	
All other lines			0.000	0.0%	
Totals			54.740	100%	

- 3 调用 `coder.ceval` 的代码行需要很长时间（16.914 秒）。此代码行的执行时间很长，因为 `coder.ceval` 将矩阵 **A** 和 **B** 的表示从行优先布局转换为列优先布局，然后将其传递给外部 **C** 函数。您可以通过在 `coder.ceval` 中使用附加参数 `-layout:rowMajor` 来避免此转换：

`sum = coder.ceval('-layout:rowMajor','foo',coder.ref(A),coder.ref(B),length);`

- 4 使用修改后的 **MyFunction** 再次生成 MEX 函数和探查。

`A = rand(20000);`  
`B = rand(20000);`

`codegen MyFunction -args {A,B} foo.c foo.h -profile`

`profile on;`  
`MyFunction_mex(A,B);`  
`profile viewer;`

**MyFunction** 的探查详细信息报告显示，调用 `coder.ceval` 的代码行现在只需 0.653 秒：

\*Lines that take the most time

Line Number	Code	Calls	Total Time (s)	% Time	Time Plot
12	sum_val = sum_val + size(A(1:row,col)) * size(B);	40000000	26.590	48.2%	
13	end	40000000	26.331	48.0%	
18	sum = coder.ceval('-layout:rowMajor','foo',coder.ref(S),coder.ref(S));	1	0.653	1.2%	
19	end	20000	0.001	0.0%	
21	for col = 1:length	20000	0.001	0.0%	
All other lines			2.535	4.6%	
Totals			55.112	100%	

## 折叠表达式对 MEX 代码覆盖率的影响

当您使用 `coder.const` 将表达式折叠成常量时，会导致 MATLAB 函数和 MEX 函数的代码覆盖率不同。以如下函数为例：

`function y = MyFoldFunction %#codegen`  
`a = 1;`  
`b = 2;`  
`c = a + b;`

```
y = 5 + coder.const(c);
end
```

探查 MATLAB 函数 `MyFoldFunction` 会在探查详细信息报告中显示以下代码覆盖率：

```
* Function listing
Time    Calls    Line
1 function y = MyFoldFunction %#codegen
2 a = 1;
3 b = 2;
4 c = a + b;
< 0.001 1 5 y = 5 + coder.const(c);
< 0.001 1 6 end
```

但是，探查 MEX 函数 `MyFoldFunction_mex` 会显示不同的代码覆盖率：

```
* Function listing
Time    Calls    Line
1 function y = MyFoldFunction %#codegen
< 0.001 1 2 a = 1;
< 0.001 1 3 b = 2;
< 0.001 1 4 c = a + b;
0.008 1 5 y = 5 + coder.const(c);
< 0.001 1 6 end
```

生成的代码中不会执行第 2、3 和 4 行，因为您已将表达式 `c = a + b` 折叠成常量以进行代码生成。

此示例使用用户定义的表达式折叠。代码生成器有时会折叠某些表达式，以优化生成代码的性能。这种优化还导致 MEX 函数的覆盖率不同于 MATLAB 函数。

## 另请参阅

`profile` | `codegen` | `coder.MexCodeConfig` | `coder.rowMajor` | `coder.ceval` | `coder.const`

## 详细信息

- “探查您的代码以改善性能”
- “Generate Code That Uses Row-Major Array Layout”



# 使用并行 for 循环 (parfor) 的算法加速

本节内容
“生成的代码中的并行 for 循环 (parfor)” (第 33-7 页)
“parfor 循环如何提高执行速度” (第 33-7 页)
“何时使用 parfor 循环” (第 33-8 页)
“何时不使用 parfor 循环” (第 33-8 页)
“parfor 循环语法” (第 33-8 页)
“parfor 限制” (第 33-8 页)

## 生成的代码中的并行 for 循环 (parfor)

为了尽可能加速执行，您可以从包含并行 for 循环 (**parfor** 循环) 的 MATLAB 代码生成 MEX 函数或 C/C++ 代码。

**parfor** 循环 (与标准 MATLAB **for** 循环类似) 对一个范围内的值执行一系列语句 (循环体)。然而，与 **for** 循环不同，**parfor** 循环的迭代可以在目标硬件的多个核上并行运行。

并行运行迭代可显著提高生成的代码的执行速度。有关详细信息，请参阅 “parfor 循环如何提高执行速度” (第 33-7 页)。

**注意** 并行执行只发生在生成的 MEX 函数或 C/C++ 代码中；而不会不发生在原始 MATLAB 代码中。要加速您的 MATLAB 代码执行速度，请从 **parfor** 循环中生成 MEX 函数。然后，从您的代码中调用 MEX 函数。有关详细信息，请参阅 “加速 MATLAB 算法的工作流” (第 33-2 页)。

要在您的 MATLAB 代码中使用 **parfor**，您需要 Parallel Computing Toolbox™ 许可证。

MATLAB Coder 软件使用 Open Multiprocessing (OpenMP) 应用程序接口来支持共享内存、多核代码生成。如果需要分布式并行机制，请使用 Parallel Computing Toolbox 产品。默认情况下，MATLAB Coder 会使用它所能找到的所有可用核。如果指定要使用的线程数，MATLAB Coder 会使用针对这些线程的最大核数量，即使有其他可用核也是如此。有关详细信息，请参阅 **parfor**。

由于循环体可以在多个线程上并行执行，因此它必须遵守某些限制。如果 MATLAB Coder 软件检测到不符合 **parfor** 设定的循环，则会产生错误。有关详细信息，请参阅 “parfor 限制” (第 33-8 页)。

## parfor 循环如何提高执行速度

**parfor** 循环可提供高于其同类 **for** 循环的执行速度，因为多个线程可以对同一循环以并发方式执行计算。

**parfor** 循环体的每次执行称为一次迭代。这些线程以任意顺序执行迭代计算，且彼此独立。由于每次迭代都是独立的，因此它们不必同步。如果线程数等于循环迭代次数，则每个线程将执行一次循环迭代。如果迭代次数超过线程数，则某些线程将执行多次循环迭代。

例如，当包含 100 次迭代的循环在 20 个线程上运行时，每个线程将同时执行五次循环迭代。如果循环因包含大量迭代或单个迭代过长而需要很长时间才能运行完毕，您可以使用多个线程来显著减少运行时间。不过，在此示例中，速度提升可能达不到 20 倍，因为存在并行化开销 (如线程创建和删除)。

## 何时使用 parfor 循环

在下列情况下，请使用 `parfor`：

- 多次简单计算迭代。`parfor` 会将这些循环迭代分为若干组，以使每个线程执行一组迭代。
- 一次循环迭代需要很长时间才能执行完毕。`parfor` 可在不同线程上同时执行迭代。虽然这种同时执行不会减少单个迭代所花费的时间，但它可以大大减少整个循环所花费的总时间。

## 何时不使用 parfor 循环

在下列情况下，请不要使用 `parfor`：

- 循环的某次迭代依赖于其他迭代。并行运行迭代可能导致错误的结果。

为了避免在循环的某次迭代依赖于其他迭代时使用 `parfor`，MATLAB Coder 指定了变量的刚性分类。有关详细信息，请参阅“Classification of Variables in parfor-Loops”。如果 MATLAB Coder 检测到循环不符合 `parfor` 设定，将不会生成代码并且会产生错误。

规则要求循环迭代必须是独立的，但归约是一个例外。归约变量会将依赖于所有迭代的值累加在一起，但与迭代顺序无关。有关详细信息，请参阅“Reduction Variables”。

- 只有几次执行一些简单计算的迭代。

---

**注意** 对于少量循环迭代，由于存在并行化开销，可能并不能提升执行速度。此类开销包括线程创建、线程之间的数据同步以及线程删除所花费的时间。

---

## parfor 循环语法

- 对于 `parfor` 循环，请使用以下语法：

```
parfor i = InitVal:EndVal
parfor (i = InitVal:EndVal)
```

- 要指定最大线程数，请使用以下语法：

```
parfor (i = InitVal:EndVal, NumThreads)
```

有关详细信息，请参阅 `parfor`。

## parfor 限制

- `parfor` 循环不支持以下语法：

```
parfor (i=initVal:step:endVal)
parfor i=initVal:step:endVal
```

- 必须使用支持 Open Multiprocessing (OpenMP) 应用程序接口的编译器。请参阅 [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/)。如果使用不支持 OpenMP 的编译器，则 MATLAB Coder 会将 `parfor` 循环视为 `for` 循环。在生成的 MEX 函数或 C/C++ 代码中，循环迭代在单个线程上运行。
- OpenMP 应用程序接口与 JIT MEX 编译不兼容。请参阅“JIT Compilation Does Not Support OpenMP”。
- 循环索引的类型必须可由目标硬件上的整数类型表示。在生成的代码中使用不需要多字类型的类型。

- 用于独立代码生成的 **parfor** 需要工具链方法来编译可执行文件或库。不要更改使代码生成器使用模板联编文件方法的设置。请参阅“Project or Configuration Is Using the Template Makefile”。
- 不要在 **parfor** 循环体中使用以下构造：

- **嵌套 parfor 循环**

一个 **parfor** 循环中可以嵌套另一个 **parfor** 循环。但是，内嵌的 **parfor** 循环将作为普通的 **for** 循环在单个线程上执行。

在 **parfor** 循环内，可以调用包含另一个 **parfor** 循环的函数。

- **break 和 return 语句**

不能在 **parfor** 循环内使用 **break** 或 **return** 语句。

- **全局变量**

不能在 **parfor** 循环内写入全局变量。

- **对 MATLAB 类使用归约**

不能在 **parfor** 循环内对 MATLAB 类使用归约。

- **对 char 变量使用归约**

不能在 **parfor** 循环内对 **char** 变量使用归约。

例如，对于以下 MATLAB 代码，您不能生成 C 代码：

```
c = char(0);
parfor i=1:10
    c = c + char(1);
end
```

在 **parfor** 循环中，MATLAB 会使 **c** 变为双精度类型。对于代码生成，**c** 不能更改类型。

- **使用外部 C 代码的归约**

在 **parfor** 循环内的归约中，不能使用 **coder.ceval**。例如，不能为以下 **parfor** 循环生成代码：

```
parfor i=1:4
    y=coder.ceval('myCFcn',y,i);
end
```

在这种情况下，应使用 **coder.ceval** 编写一个调用 C 代码的局部函数，并在 **parfor** 循环中调用此函数。例如：

```
parfor i=1:4
    y = callMyCFcn(y,i);
end
...
function y = callMyCFcn(y,i)
    y = coder.ceval('mCyFcn', y , i);
end
```

- **外部函数调用**

不能在 **parfor** 循环内使用 **coder.extrinsic** 调用外部函数。调用包含外部调用的函数会导致运行时错误。

- **内联函数**

MATLAB Coder 不会将函数内联到 **parfor** 循环中，包括使用 `coder.inline('always')` 的函数。

- **展开循环**

不能在 **parfor** 循环内使用 `coder.unroll`。

如果在 **parfor** 循环内展开某个循环，MATLAB Coder 将无法对变量进行分类。例如：

```
for j=coder.unroll(3:6)
    y(i,j)=y(i,j)+i+j;
end
```

此代码展开为：

```
y(i,3)=y(i,3)+i+3;
...
y(i,6)=y(i,6)+i+6;
```

在展开的代码中，MATLAB Coder 不能对变量 `y` 进行分类，因为 `y` 在 **parfor** 循环内的索引方式不同。

MATLAB Coder 不支持它不能分类的变量。有关详细信息，请参阅 “Classification of Variables in parfor-Loops”。

- **varargin/varargout**

不能在 **parfor** 循环内使用 `varargin` 或 `varargout`。

## 外部代码集成

---

# 从生成的代码中调用自定义 C/C++ 代码

本节内容
“调用 C 代码” (第 34-2 页)
“从一个 C 函数返回多个值” (第 34-3 页)
“按引用传递数据” (第 34-4 页)
“集成使用自定义数据类型的外部代码” (第 34-5 页)
“集成使用指针、结构体和数组的外部代码” (第 34-6 页)

在 MATLAB 代码中，您可以直接调用外部 C/C++ 代码，也称为自定义代码或原有代码。要调用 C/C++ 函数，请使用 `coder.ceval`。代码生成器将您的 C/C++ 代码集成到从 MATLAB 生成的 C/C++ 代码中。当您要将在生成的代码与外部库、优化的代码或在 C/C++ 中开发的目标文件结合使用时，请集成代码。当外部代码使用 MATLAB 未定义或不能识别的变量类型时，请将 `coder.opaque` 函数与 `coder.ceval` 结合使用。要保留某些标识符名称，以便在要与生成的代码集成的自定义 C/C++ 代码中使用，请使用 `coder.reservedName` 函数。

以下是外部代码集成的一些主要工作流。有关更多示例，请参阅 `coder.ceval` 参考页。

**注意** 使用 `coder.ceval`，您可以不受限制地访问外部代码。在代码中误用这些函数或者代码中的错误可能会导致 MATLAB 不稳定并停止工作。要调试代码并分析编译中的错误消息，请查看代码生成报告中的 **Build Logs** 选项卡。

`coder.ceval` 只能在 MATLAB 代码中用于代码生成。在未编译的 MATLAB 代码中，`coder.ceval` 将生成错误。要确定 MATLAB 函数是否在 MATLAB 中执行，请使用 `coder.target`。如果函数在 MATLAB 中执行，请调用 MATLAB 版本的 C/C++ 函数。

## 调用 C 代码

此示例说明如何使用 `coder.ceval` 将简单的 C 函数与 MATLAB® 代码集成。以 MATLAB 函数 `mathOps` 为例：

```
function [added, mulated] = mathOps(in1, in2)
added = in1+in2;
mulated = in1*in2;
end
```

对于此示例，假设您要使用外部 C 代码实现加法运算。请考虑在 `adder.c` 文件中实现的 C 函数 `adder`：

```
#include <stdio.h>
#include <stdlib.h>
#include "adder.h"

double adder(double in1, double in2)
{
    return in1 + in2;
}
```

要将 `adder` 与 MATLAB 代码集成，需要具有包含函数原型的头文件。请参阅文件 `adder.h`：

```
double adder(double in1, double in2);
```

使用 `coder.ceval` 命令在 `mathOpsIntegrated.m` 中调用 C 函数。使用 `coder.cinclude` 包含头文件。

```
function [added, multed] = mathOpsIntegrated(in1, in2)
%#codegen
% for code generation, preinitialize the output variable
% data type, size, and complexity
added = 0;
% generate an include in the C code
coder.cinclude('adder.h');
% evaluate the C function
added = coder.ceval('adder', in1, in2);
multed = in1*in2;
end
```

要生成代码，请使用 `codegen` 命令。将源文件 `adder.c` 指定为输入。要测试 C 代码，请执行 MEX 函数并检查输出结果。

```
codegen mathOpsIntegrated -args {1, 2} adder.c
```

```
[test1, test2] = mathOpsIntegrated_mex(10, 20)
```

Code generation successful.

```
test1 =
```

```
    30
```

```
test2 =
```

```
   200
```

## 从一个 C 函数返回多个值

C 语言限制函数返回多个输出。函数只能返回单个标量值。您可以使用 MATLAB 函数 `coder.ref`、`coder.rref` 和 `coder.wref` 从一个外部 C/C++ 函数返回多个输出。

例如，假设您编写 MATLAB 函数 `foo`，该函数接受 `x` 和 `y` 两个输入并返回 `a`、`b` 和 `c` 三个输出。在 MATLAB 中，您可以按如下方式调用此函数：

```
[a,b,c] = foo(x,y)
```

如果将 `foo` 重写为 C 函数，则不能通过一个 `return` 语句返回三个单独的值 `a`、`b` 和 `c`。在这种情况下，请创建一个具有多个指针类型参量的 C 函数，并按引用传递输出参数。例如：

```
void foo(double x,double y,double *a,double *b,double *c)
```

然后，您可以使用 `coder.ceval` 函数从 MATLAB 函数调用该 C 函数。

```
coder.ceval('foo',x,y,coder.ref(a),coder.ref(b),coder.ref(c));
```

如果您的外部 C 函数仅按引用方式读写内存，则可以使用 `coder.wref` 或 `coder.rref` 函数而不是 `coder.ref`。在某些情况下，这些函数可以进一步优化生成的代码。当您使用 `coder.wref(arg)` 按引用传递 `arg` 时，外部 C/C++ 函数必须对 `arg` 引用的内存进行完全初始化。

## 按引用传递数据

此示例说明如何按引用向外部 C 函数传递数据或传递来自外部 C 函数的数据。

按引用传递是 C/C++ 代码集成的一种重要方法。按引用传递数据时，程序不需要将数据从一个函数复制到另一个函数。按值传递时，C 代码只能返回单个标量变量。按引用传递时，C 代码可以返回多个变量，包括数组。

以 MATLAB 函数 `adderRef` 为例。此函数使用外部 C 代码将两个数组相加。`coder.rref` 和 `coder.wref` 命令指示代码生成器将指针传递给数组，而不是复制它们。

```
function out = adderRef(in1, in2)
%#codegen
out = zeros(size(in1));
% the input numel(in1) is converted to integer type
% to match the cAdd function signature
coder.ceval('cAdd', coder.rref(in1), coder.rref(in2), coder.wref(out), int32(numel(in1)) );
end
```

C 代码 `cAdd.c` 使用线性索引来访问数组的元素：

```
#include <stdio.h>
#include <stdlib.h>
#include "cAdd.h"

void cAdd(const double* in1, const double* in2, double* out, int numel)
{
    int i;
    for (i=0; i<numel; i++) {
        out[i] = in1[i] + in2[i];
    }
}
```

要编译 C 代码，您必须提供带有函数签名的头文件 `cAdd.h`：

```
void cAdd(const double* in1, const double* in2, double* out, int numel);
```

通过生成 MEX 函数并将其输出与 MATLAB 中的加法运算的输出进行比较来测试 C 代码。

```
A = rand(2,2)+1;
B = rand(2,2)+10;

codegen adderRef -args {A, B} cAdd.c cAdd.h -report

if (adderRef_mex(A,B) - (A+B) == 0)
```



```
fprintf(['\n' 'adderRef was successful.']);
end
```

Code generation successful: To view the report, open('codegen/mex/adderRef/html/report.mldatx')

adderRef was successful.

## 集成使用自定义数据类型的外部代码

此示例说明如何调用使用了非 MATLAB® 定义的原生数据类型的 C 函数。

例如，如果您的 C 代码对 C 'FILE \*' 类型执行文件输入或输出，则 MATLAB 中没有对应的类型。要在 MATLAB 代码中与此数据类型进行交互，必须使用 `coder.opaque` 函数对其进行初始化。对于结构体类型，可以使用 `coder.cstructname`。

例如，假设有 MATLAB 函数 `addCTypes.m`。此函数使用其输入类型在外部代码中定义的 `coder.ceval`。函数 `coder.opaque` 在 MATLAB 中对该类型进行初始化。

```
function [out] = addCTypes(a,b)
%#codegen
% generate include statements for header files
coder.cinclude('MyStruct.h');
coder.cinclude('createStruct.h');
coder.cinclude('useStruct.h');
% initialize variables before use
in = coder.opaque('MyStruct');
out = 0;
% call C functions
in = coder.ceval('createStruct',a,b);
out = coder.ceval('useStruct',in);
end
```

`createStruct` 函数输出 C 结构体类型：

```
#include <stdio.h>
#include <stdlib.h>
#include "MyStruct.h"
#include "createStruct.h"

struct MyStruct createStruct(double a, double b) {
    struct MyStruct out;
    out.p1 = a;
    out.p2 = b;
    return out;
}
```

`useStruct` 函数对该 C 类型执行操作：

```
#include "MyStruct.h"
#include "useStruct.h"
```

```
double useStruct(struct MyStruct in) {
    return in.p1 + in.p2;
}
```

要生成代码，请将源 (.c) 文件指定为输入：

```
codegen addCTypes -args {1,2} -report createStruct.c useStruct.c
```

Code generation successful: To view the report, open('codegen/mex/addCTypes/html/report.mldatx')

## 集成使用指针、结构体和数组的外部代码

此示例说明如何将对 C 样式数组执行运算的外部代码与 MATLAB® 代码集成。外部代码计算数组数据的和。您可以自定义代码以更改输入数据或计算。

此示例说明如何合并外部代码集成功能的多个不同元素。例如，您可以：

- 使用 `coder.cstructname` 对接外部结构体类型
- 使用 `coder.opaque` 对接外部指针类型
- 使用 `coder.ceval` 执行外部代码
- 使用 `coder.ref` 按引用将数据传递到外部代码

### 浏览集成后的代码

`extSum` 函数使用外部 C 代码对 32 位整数数组执行求和运算。数组大小由用户输入控制。

```
function x = extSum(u)
%#codegen
% set bounds on input type to use static memory allocation
u = int32(u);
assert(0 < u && u < 101);
% initialize an array
temparray = int32(1):u;
% declare an external structure and use it
s = makeStruct(u);
x = callExtCode(s, temparray);
```

为了简化生成的代码，请对数组大小设置限制。限制可防止在生成的代码中使用动态内存分配。

函数 `makeStruct` 声明一种 MATLAB 结构体类型，并使用 `coder.opaque` 将其中一个字段初始化为指针类型。与此定义对应的 C 结构体包含在一个头文件中，您通过使用 `coder.cstructname` 函数中的 `HeaderFile` 参数提供该头文件。C 结构体类型提供了整数数组的简单表示形式。

```
function s = makeStruct(u)
% create structure type based on external header definition
s.numel = u;
s.vals = coder.opaque('int32_T *', 'NULL');
coder.cstructname(s, 'myArrayType', 'extern', 'HeaderFile', 'arrayCode.h');
```

在完全初始化外部结构体类型后，您可以将其作为输入传递给 `callExtCode` 函数中的外部代码。此函数对数组进行初始化，调用对数组执行的运算以返回单个输出，然后释放经过初始化的内存。

```
function x = callExtCode(s, temparray)
% declare output type
x = int32(0);
% declare external source file
coder.updateBuildInfo('addSourceFiles','arrayCode.c');
% call c code
coder.ceval('arrayInit',coder.ref(s),coder.ref(temparray));
x = coder.ceval('arraySum',coder.ref(s));
coder.ceval('arrayDest',coder.ref(s));
```

该函数使用 `coder.updateBuildInfo` 将 .c 文件提供给代码生成器。

### 生成 MEX 函数

要生成可以在 MATLAB 中运行和测试的 MEX 函数，请输入：

```
codegen extSum -args {10}
```

Code generation successful.

测试 MEX 函数。输入：

```
extSum_mex(10)
```

```
ans =
```

```
int32
```

```
55
```

包含在文件 `arrayCode.c` 和 `arrayCode.h` 中的外部 C 代码使用自定义类型定义 `int32_T`。生成的 MEX 代码将生成并使用此自定义类型定义。如果要生成使用此自定义数据类型的独立 (lib、dll 或 exe) 代码，您可以修改配置对象的 `DataTypeReplacement` 属性。请参阅“Mapping MATLAB Types to Types in Generated Code”。

### 另请参阅

`codegen` | `coder.wref` | `coder.ceval` | `coder.rref` | `coder.ref` | `coder.cinclude` | `coder.cstructname` | `coder.opaque` | `coder.reservedName`

### 详细信息

- “Configure Build for External C/C++ Code”
- “Unit Test External C Code with MATLAB Coder”



## 生成高效且可重用的代码

---

- “优化策略” (第 35-2 页)
- “内联代码” (第 35-4 页)
- “使用并行 for 循环 (parfor) 生成代码” (第 35-5 页)
- “MATLAB Coder 对生成代码进行优化” (第 35-6 页)

优化策略

MATLAB Coder 在从您的 MATLAB 代码生成 C/C++ 代码或 MEX 函数时会进行一定的优化。有关详细信息，请参阅“MATLAB Coder 对生成代码进行优化”（第 35-6 页）。

要进一步优化生成的代码，您可以：

- 调整您的 MATLAB 代码。
- 从命令行或工程设置对话框使用配置对象控制代码生成。

要优化生成的代码的执行速度，请针对下列情况根据需要执行相应操作：

条件	操作
您有 for 循环，其迭代彼此独立。	“使用并行 for 循环 (parfor) 生成代码”（第 35-5 页）  “Automatically Parallelize for Loops in Generated Code”
您的 MATLAB 代码中有可变大小数组。	“Minimize Dynamic Memory Allocation”
您的 MATLAB 代码中有多个可变大小数组。您要为较大的数组使用动态内存分配，为较小的数组使用静态内存分配。	“Set Dynamic Memory Allocation Threshold”
您要通过引用调用生成的函数。	“Avoid Data Copies of Function Inputs in Generated Code”
您正在调用 MATLAB 代码中的小函数。	“内联代码”（第 35-4 页）
您生成的代码的目标内存受限。您要内联小函数并为较大的函数生成单独的代码。	“Control Inlining to Fine-Tune Performance and Readability of Generated Code”
您不希望为仅包含常量的表达式生成代码。	“Fold Function Calls into Constants”
您的 MATLAB 代码中的循环运算不依赖于循环索引。	“Minimize Redundant Operations in Loops”
您的 MATLAB 代码中有整数运算。您事先知道在执行生成的代码期间不会发生整数溢出。	“Disable Support for Integer Overflow”
您事先知道在执行生成的代码期间不会出现 Inf 和 NaN。	“Disable Support for Nonfinite Numbers”
您有迭代次数很少的 for 循环。	“Unroll for-Loops and parfor-Loops”
您已针对目标环境优化了原有 C/C++ 代码。	“Integrate External/Custom Code”
您希望加快为基本向量和矩阵函数生成的代码的执行速度。	“Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls”
您希望加快为线性代数函数生成的代码的执行速度。	“Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”
您希望加速为快速傅里叶变换 (FFT) 函数生成的代码。	“Speed Up Fast Fourier Transforms in Generated Standalone Code by Using FFTW Library Calls”

要优化生成的代码的内存使用量，请针对下列情况根据需要执行相应操作：

条件	操作
在 MATLAB 代码中有 <code>if/else/elseif</code> 语句或 <code>switch/case/otherwise</code> 语句。您不需要在生成的代码中使用语句的某些分支。	"Prevent Code Generation for Unused Execution Paths"
您要通过引用调用生成的函数。	"Avoid Data Copies of Function Inputs in Generated Code"
您生成的代码的堆栈空间受限。	"Control Stack Space Usage"
您正在调用 MATLAB 代码中的小函数。	"内联代码" (第 35-4 页)
您生成的代码的目标内存受限。您要内联小函数并为较大的函数生成单独的代码。	"Control Inlining to Fine-Tune Performance and Readability of Generated Code"
您不希望为仅包含常量的表达式生成代码。	"Fold Function Calls into Constants"
您的 MATLAB 代码中的循环运算不依赖于循环索引。	"Minimize Redundant Operations in Loops"
您的 MATLAB 代码中有整数运算。您事先知道在执行生成的代码期间不会发生整数溢出。	"Disable Support for Integer Overflow"
您事先知道在执行生成的代码期间不会出现 <code>Inf</code> 和 <code>NaN</code> 。	"Disable Support for Nonfinite Numbers"
您的 MATLAB 代码包含大型数组变量或结构体变量。您的变量不会在生成的代码中重用。它们是保留变量。您希望查看保留大型数组或结构体的变量名称所需的额外内存是否会影响性能。	"Reuse Large Arrays and Structures"

## 内联代码

内联是一种用函数的内容（函数体）替换函数调用的方法。内联消除了函数调用的开销，但会生成更多的 C/C++ 代码。如果生成的 C/C++ 代码使用了内联，则可能有进一步优化的空间。代码生成器使用内部启发式方法来确定是否在生成的代码中内联函数。您可以使用 `coder.inline` 指令为单个函数微调这些启发式方法。有关详细信息，请参阅 `coder.inline`。

### 另请参阅

### 详细信息

- “Control Inlining to Fine-Tune Performance and Readability of Generated Code”



## 使用并行 for 循环 (parfor) 生成代码

以下示例说明如何为包含 **parfor** 循环的 MATLAB 算法生成 C 代码。

- 1 编写包含 **parfor** 循环的 MATLAB 函数。例如：

```
function a = test_parfor %#codegen
a=ones(10,256);
r=rand(10,256);
parfor i=1:10
    a(i,:)=real(fft(r(i,:)));
end
```

- 2 生成 **test\_parfor** 的 C 代码。在 MATLAB 命令行中输入：

```
codegen -config:lib test_parfor
```

因为您没有指定要使用的最大线程数，所生成的 C 代码将在可用的内核上并行执行循环迭代。

- 3 要指定最大线程数，请按如下所示重写函数 **test\_parfor**：

```
function a = test_parfor(u) %#codegen
a=ones(10,256);
r=rand(10,256);
parfor (i=1:10,u)
    a(i,:)=real(fft(r(i,:)));
end
```

- 4 生成 **test\_parfor** 的 C 代码。使用 **-args 0** 指定输入 **u** 是双精度标量。在 MATLAB 命令行中输入：

```
codegen -config:lib test_parfor -args 0
```

在生成的代码中，运行 **parfor** 循环迭代的内核数最多只能为输入 **u** 指定的内核数。如果可用的内核数小于 **u** 指定的内核数，则在调用时迭代会在可用的内核上运行。

## 另请参阅

### 详细信息

- “使用并行 for 循环 (parfor) 的算法加速” (第 33-7 页)
- “Classification of Variables in parfor-Loops”
- “Reduction Assignments in parfor-Loops”

# MATLAB Coder 对生成代码进行优化

本节内容
“常量折叠” (第 35-6 页)
“循环融合” (第 35-6 页)
“合并的连续矩阵运算” (第 35-7 页)
“排除不可达代码” (第 35-7 页)
“memcpy 调用” (第 35-7 页)
“memset 调用” (第 35-8 页)

为了改进生成代码的执行速度和内存使用量，MATLAB Coder 引入了以下优化：

## 常量折叠

代码生成器会尽可能计算 MATLAB 代码中仅包含编译时常量的表达式。在生成的代码中，它用计算结果替换这些表达式。此行为称为常量折叠。由于使用了常量折叠，生成的代码不必在执行期间计算常量。

以下示例说明在代码生成过程中经过常量折叠的 MATLAB 代码。函数 `MultiplyConstant` 将矩阵中的每个元素乘以一个标量常量。该函数使用三个编译时常量 `a`、`b` 和 `c` 的乘积来计算该常量。

```
function out=MultiplyConstant(in) %#codegen
a=pi^4;
b=1/factorial(4);
c=exp(-1);
out=in.*(a*b*c);
end
```

代码生成器计算涉及编译时常量 `a`、`b` 和 `c` 的表达式。它用生成代码中的计算结果替换这些表达式。

当表达式只涉及标量时，可能会发生常量折叠。要在其他情况下显式强制实施表达式的常量折叠，请使用 `coder.const` 函数。有关详细信息，请参阅“Fold Function Calls into Constants”。

## 控制常量折叠

您可以从命令行或工程设置对话框中控制可进行常量折叠的指令的最大数量。

- 在命令行中，为代码生成创建一个配置对象。将属性 `ConstantFoldingTimeout` 设置为所需的值。

```
cfg=coder.config('lib');
cfg.ConstantFoldingTimeout = 200;
```
- 使用 App，在工程设置对话框中的 **All Settings** 选项卡上，将字段 **Constant folding timeout** 设置为所需的值。

## 循环融合

代码生成器会尽可能将具有相同运行次数的连续循环融合到生成代码中的单个循环中。这种优化可减少循环开销。

以下代码包含连续循环，这些循环会在代码生成过程中融合。函数 `SumAndProduct` 计算数组 `Arr` 中元素的和与乘积。该函数使用两个单独的循环来计算 `y_f_sum` 和与 `y_f_prod` 乘积。

```
function [y_f_sum,y_f_prod] = SumAndProduct(Arr) %#codegen
y_f_sum = 0;
y_f_prod = 1;
for i = 1:length(Arr)
    y_f_sum = y_f_sum+Arr(i);
end
for i = 1:length(Arr)
    y_f_prod = y_f_prod*Arr(i);
end
```

从该 MATLAB 代码生成的代码在单一循环中计算和与乘积。

## 合并的连续矩阵运算

代码生成器会尽可能将 MATLAB 代码中的连续矩阵运算转换为生成代码中的单一循环运算。这种优化可减少在单独的循环中执行矩阵运算所涉及的过多循环开销。

以下示例包含发生连续矩阵运算的代码。函数 `ManipulateMatrix` 将矩阵 `Mat` 中的每个元素与一个 `factor` 相乘。然后，该函数为结果中的每个元素加一个 `shift`：

```
function Res=ManipulateMatrix(Mat,factor,shift)
    Res=Mat*factor;
    Res=Res+shift;
end
```

生成的代码将乘法和加法合并到单个循环运算中。

## 排除不可达代码

代码生成器会尽可能在代码生成中隐藏 MATLAB 代码中的不可达过程。例如，如果 `if`, `elseif`, `else` 语句的某个分支不可达，则不会为该分支生成代码。

以下示例包含不可达代码，这些代码在代码生成过程中会被排除。函数 `SaturateValue` 根据其输入 `x` 的范围返回一个值。

```
function y_b = SaturateValue(x) %#codegen
if x>0
    y_b = x;
elseif x>10 %This is redundant
    y_b = 10;
else
    y_b = -x;
end
```

`if`、`elseif`、`else` 语句的第二个分支不可达。如果变量 `x` 大于 10，显然它也大于 0。因此，会优先于第二个分支执行第一个分支。

MATLAB Coder 不会为不可达的第二个分支生成代码。

## memcpy 调用

为了优化用于复制连续数组元素的生成代码，代码生成器会尝试通过调用 `memcpy` 来替换代码。`memcpy` 调用的效率可能高于 `for` 循环或多个连续元素赋值等代码的效率。

请参阅“`memcpy` Optimization”。

## **memset 调用**

为了优化对连续数组元素进行字面常量赋值的生成代码，代码生成器会尝试通过调用 `memset` 来替换代码。`memset` 调用的效率可能高于 `for` 循环或多个连续元素赋值等代码的效率。

请参阅“`memset Optimization`”。

## 从 MATLAB 代码生成可重入 C 代码

---



## 代码生成问题排查

---

## 无法确定元胞数组的每个元素都已赋值

### 问题

您会看到以下消息之一：

Unable to determine that every element of 'y' is assigned before this line.

Unable to determine that every element of 'y' is assigned before exiting the function.

Unable to determine that every element of 'y' is assigned before exiting the recursively called function.

### 原因

对于代码生成，在使用某元胞数组元素之前，必须为其赋值。当您使用 `cell` 创建可变大小元胞数组（例如 `cell(1,n)`）时，MATLAB 会用一个空矩阵对每个元素赋值。然而，代码生成并不会对元素进行赋值。对于代码生成，在您使用 `cell` 创建可变大小元胞数组后，必须对该元胞数组的所有元素赋值，然后才能使用该元胞数组。

在首次使用元胞数组之前，代码生成器会分析您的代码，以确定是否所有元素均已赋值。当代码遵循以下模式时，代码生成器会检测到所有元素都已赋值：

```
function z = CellVarSize1D(n, j)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
end
```

以下是针对多维元胞数组的模式：

```
function z = CellAssign3D(m,n,p)
%#codegen
x = cell(m,n,p);
for i = 1:m
    for j = 1:n
        for k = 1:p
            x{i,j,k} = i+j+k;
        end
    end
end
z = x{m,n,p};
end
```

如果代码生成器检测到某些元素未赋值，代码生成将失败。有时，尽管您的代码对元胞数组的所有元素进行了赋值，代码生成也会失败，因为分析没有检测到所有元素都赋值。

以下是代码生成器无法检测到元素已赋值的示例：



- 元素在不同循环中赋值

```
...
x = cell(1,n)
for i = 1:5
    x{i} = 5;
end
for i = 6:n
    x{i} = 7;
end
...
```

- 定义循环结束值的变量与定义元胞维度的变量不同。

```
...
x = cell(1,n);
m = n;
for i = 1:m
    x{i} = 2;
end
...
```

有关详细信息，请参阅“使用 cell 定义可变大小元胞数组”（第 9-3 页）。

## 解决方法

尝试以下解决方法之一：

- “使用可识别的模式为元素赋值”（第 37-3 页）
- “使用 repmat”（第 37-3 页）
- “使用 coder.nullcopy”（第 37-4 页）

### 使用可识别的模式为元素赋值

如果可能，重写您的代码以遵循以下模式：

```
...
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
z = x{j};
...
```

### 使用 repmat

有时，您可以使用 **repmat** 来定义可变大小的元胞数组。

以如下定义可变大小元胞数组的代码为例。它将 1 赋值给奇数元素，将 2 赋值给偶数元素。

```
function z = repDefine(n, j)
%#codegen
c = cell(1,n);
for i = 1:2:n-1
    c{i} = 1;
end
for i = 2:2:n
```

```

    c{i} = 2;
end
z = c{j};

```

代码生成不允许以下代码，因为：

- 多个循环为元素赋值。
- 循环计数器不是按 1 递增。

重写代码，首先使用 `cell` 创建第一个元素为 1，第二个元素为 2 的 1×2 元胞数组。然后，使用 `repmat` 创建一个可变大小的元胞数组，其元素值在 1 和 2 之间交替。

```

function z = repVarSize(n, j)
%#codegen
c = cell(1,2);
c{1} = 1;
c{2} = 2;
c1= repmat(c,1,n);
z = c1{j};
end

```

您可以使用 `repmat` 将初始为空的可变大小元胞数组传入或传出函数。使用以下模式：

```

function x = emptyVarSizeCellArray
x = repmat({'abc'},0,0);
coder.varsize('x');
end

```

以下代码表示 `x` 是由 1x3 个字符组成的空的可变大小元胞数组，可以传入或传出函数。

### 使用 `coder.nullcopy`

最后，您还可以使用 `coder.nullcopy` 来指示代码生成器可以为元胞数组分配内存，而无需初始化内存。例如：

```

function z = nulcpyCell(n, j)
%#codegen
c = cell(1,n);
c1 = coder.nullcopy(c);
for i = 1:4
    c1{i} = 1;
end
for i = 5:n
    c1{i} = 2;
end
z = c1{j};
end

```

请慎用 `coder.nullcopy`。如果访问未初始化的内存，结果是不可预测的。

### 另请参阅

`cell` | `repmat` | `coder.nullcopy`

### 详细信息

- “代码生成的元胞数组限制” (第 9-2 页)

## 行优先数组布局

---

## 行优先和列优先数组布局

数组的元素可以存储在列优先布局或行优先布局中。对于存储在列优先布局中的数组，列的元素在内存中是连续的。在行优先布局中，行的元素是连续的。数组布局也称为顺序、格式和表示。元素的存储顺序对于集成、可用性和性能非常重要。某些算法对以特定顺序存储的数据的执行效率更高。

编程语言和环境通常为所有数据假设单一的数组布局。MATLAB 和 Fortran 默认使用列优先布局，而 C 和 C++ 使用行优先布局。使用 MATLAB Coder，您可以生成使用行优先布局或列优先布局的 C/C++ 代码。请参阅“Generate Code That Uses Row-Major Array Layout”。

### 计算机内存中的数组存储

计算机内存以一维数组的形式存储数据。例如，当您声明  $3 \times 3$  矩阵时，软件会将此矩阵存储为包含九个元素的一维数组。默认情况下，MATLAB 以列优先数组布局存储这些元素。每列的元素在内存中是连续的。

以如下矩阵 A 为例：

```
A =
    1    2    3
    4    5    6
    7    8    9
```

默认情况下，矩阵 A 在内存中的排列方式如下所示：

```
1  4  7  2  5  8  3  6  9
```

在行优先数组布局中，编程语言将行元素连续存储在内存中。在行优先布局中，数组的元素存储为：

```
1  2  3  4  5  6  7  8  9
```

N 维数组也可以存储在列优先布局或行优先布局中。在列优先布局中，第一个（最左边）维度或索引的元素在内存中是连续的。在行优先布局中，最后一个（最右边）维度或索引的元素是连续的。

### 不同数组布局之间的转换

当您在同一代码中混合行优先数据和列优先数据时，需要进行数组布局转换。例如，您可以生成包含行优先和列优先函数特化的代码。函数特化对所有输入、输出和内部数据使用一种类型的数组布局。在函数之间传递数据时，代码生成器会根据需要自动插入数组布局转换。生成的 MEX 函数的输入和输出数据也根据需要进行转换。

对于二维数据，转置运算在行优先布局和列优先布局之间转换数据。以 A 的如下转置版本为例：

```
A' =
    1    4    7
    2    5    8
    3    6    9
```

A' 的列优先布局与 A 的行优先布局相匹配。（对于复数，数组布局转换使用非共轭转置。）

### 另请参阅

`coder.columnMajor` | `coder.rowMajor` | `coder.isRowMajor` | `coder.isColumnMajor`

## 详细信息

- “Generate Code That Uses Row-Major Array Layout”
- “MATLAB 数据”
- “Generate Code That Uses N-Dimensional Indexing”



# 使用 MATLAB Coder 进行深度学习

---

- “使用 MATLAB Coder 进行深度学习的前提条件” (第 39-2 页)
- “代码生成支持的网络和层” (第 39-7 页)
- “加载预训练网络以用于代码生成” (第 39-23 页)

# 使用 MATLAB Coder 进行深度学习的前提条件

## MathWorks 产品

要使用 MATLAB Coder 为深度学习网络生成代码，您还必须安装：

- Deep Learning Toolbox™
- MATLAB Coder Interface for Deep Learning Libraries

MATLAB Online 不支持 MATLAB Coder Interface for Deep Learning Libraries。

## 第三方硬件和软件

您可以使用 MATLAB Coder 为部署到 Intel® 或 ARM® 处理器的深度学习网络生成 C++ 代码。生成的代码利用针对目标 CPU 进行了优化的深度学习库。硬件和软件要求取决于目标平台。

您还可以使用 MATLAB Coder 为深度学习网络生成泛型 C 或 C++ 代码。这种 C 或 C++ 代码不依赖于任何第三方库。有关详细信息，请参阅 “Generate Generic C/C++ Code for Deep Learning Networks” 。

**注意** 所需软件库的路径不能包含空格或特殊字符，如括号。在 Windows 操作系统中，仅当启用 8.3 文件名时才允许使用特殊字符和空格。有关 8.3 文件名的详细信息，请参考 Windows 文档。

	Intel CPU	ARM Cortex-A CPU	ARM Cortex-M CPU
硬件要求	支持 Intel Advanced Vector Extensions 2 (Intel AVX2) 指令的 Intel 处理器。	支持 NEON 扩展的 ARM Cortex-A 处理器。	ARM Cortex-M 处理器。



	Intel CPU	ARM Cortex-A CPU	ARM Cortex-M CPU
软件库	<p>Intel Math Kernel Library for Deep Neural Networks (MKL-DNN), v1.4。请参阅 <a href="https://01.org/onednn">https://01.org/onednn</a>。</p> <p>不要使用预置库，因为缺失一些必需的文件。请从源代码编译库。请参阅 GitHub® 上的编译库的说明。</p> <p>有关编译的详细信息，请参阅 MATLAB Answers™ 中的以下帖子： <a href="https://www.mathworks.com/matlabcentral/answers/447387-matlab-coder-how-do-i-build-the-intel-mkl-dnn-library-for-deep-learning-c-code-generation-and-dep">https://www.mathworks.com/matlabcentral/answers/447387-matlab-coder-how-do-i-build-the-intel-mkl-dnn-library-for-deep-learning-c-code-generation-and-dep</a></p> <p>使用说明：</p> <ul style="list-style-type: none"> <li>当生成在您的 MATLAB 主机上运行的 MEX 函数时，建议您使用 MKL-DNN 目标，而不是生成泛型 C/C++ 代码。使用 MKL-DNN 库的生成代码可能比泛型代码具有更好的性能。</li> <li>对于一些网络，MKL-DNN 库在 AVX2 计算机上的性能可能较慢。使用 AVX512 计算机，通过生成的代码充分利用 MKL-DNN 的性能。</li> </ul>	<p>针对计算机视觉和机器学习的 ARM Compute Library，版本 19.05 和 20.02.1。请参阅 <a href="https://developer.arm.com/ip-products/processors/machine-learning/compute-library">https://developer.arm.com/ip-products/processors/machine-learning/compute-library</a>。</p> <p>在 <b>coder.ARMNEONConfig</b> 配置对象中指定版本号。默认版本号是 v20.02.1。</p> <p>不要使用预置库，因为它可能与 ARM 硬件上的编译器不兼容。请从源代码编译库。请在您的主机上或直接在目标硬件上编译库。请参阅 GitHub 上的编译库的说明。</p> <p>包含库文件（如 <b>libarm_compute.so</b>）的文件夹应命名为 <b>lib</b>。如果文件夹名称为 <b>build</b>，请将文件夹重命名为 <b>lib</b>。</p> <p>有关编译的详细信息，请参阅 MATLAB Answers 中的以下帖子： <a href="https://www.mathworks.com/matlabcentral/answers/455590-matlab-coder-how-do-i-build-the-arm-compute-library-for-deep-learning-c-code-generation-and-deplo">https://www.mathworks.com/matlabcentral/answers/455590-matlab-coder-how-do-i-build-the-arm-compute-library-for-deep-learning-c-code-generation-and-deplo</a></p> <p>要在 ARM 处理器上部署以 8 位整数执行推断计算的生成代码，您必须使用 ARM Compute Library 版本 20.02.1。</p>	<p>CMSIS-NN 库版本 5.7.0。请参阅 <a href="https://developer.arm.com/tools-and-software/embedded/cmsis">https://developer.arm.com/tools-and-software/embedded/cmsis</a>。</p> <p>使用在以下位置提供的编译步骤在您的主机上编译库：</p> <ul style="list-style-type: none"> <li><a href="https://github.com/mathworks/build-steps-for-cmsisnn-library">https://github.com/mathworks/build-steps-for-cmsisnn-library</a></li> <li><a href="https://www.mathworks.com/matlabcentral/answers/1631260">https://www.mathworks.com/matlabcentral/answers/1631260</a></li> </ul>
操作系统支持	Windows、Linux® 和 macOS。	仅限 Windows 和 Linux。	仅限 Windows 和 Linux。

	Intel CPU	ARM Cortex-A CPU	ARM Cortex-M CPU
<b>支持的编译器</b>	<p>MATLAB Coder 将查找并使用支持的已安装编译器。有关受支持编译器的列表，请参阅 MathWorks 网站上的支持和兼容的编译器。</p> <p>可以使用 <code>mex -setup</code> 更改默认编译器。请参阅“更改默认编译器”。</p> <p>C++ 编译器必须支持 C++11。</p> <p>在 Windows 上，要使用 <code>codegen</code> 命令生成使用 Intel MKL-DNN 库的代码，请使用 Microsoft® Visual Studio® 2015 或更高版本。</p> <p>在 Windows 上，要生成不使用任何第三方库的泛型 C 或 C++ 代码，请使用 Microsoft Visual Studio 或 MinGW® 编译器。有关详细信息，请参阅“Generate Generic C/C++ Code for Deep Learning Networks”。</p> <p><b>注意</b> 在 Windows 上，不支持使用 MinGW 编译器生成使用 Intel MKL-DNN 库的 MEX 函数。</p>		
<b>其他</b>	<p>开源计算机视觉库 (OpenCV)，基于 ARM Cortex-A 的深度学习示例需要的版本为 v3.1.0。</p> <p>注意：这些示例需要单独的库，例如 <code>opencv_core.lib</code> 和 <code>opencv_video.lib</code>。Computer Vision Toolbox™ 附带的 OpenCV 库没有必需的库，OpenCV 安装程序也不会安装它们。因此，您必须下载 OpenCV 源代码并编译这些库。</p> <p>有关详细信息，请参考 OpenCV 文档。</p>		

## 环境变量

MATLAB Coder 使用环境变量来定位为深度学习网络生成代码所需的库。

平台	变量名称	描述
Windows	INTEL_MKLDNN	Intel MKL-DNN 库安装的根文件夹的路径。  例如：  <code>C:\Program Files\mkl-dnn</code>
	ARM_COMPUTELIB	ARM 目标硬件上 ARM Compute Library 安装的根文件夹的路径。  例如：  <code>/usr/local/arm_compute</code>  在 ARM 目标硬件上设置 ARM_COMPUTELIB。

平台	变量名称	描述
	<b>CMSISNN_PATH</b>	ARM 目标硬件上 CMSIS-NN 库安装的根文件夹的路径。 例如： <code>/usr/local/cmsis_nn</code> 在 ARM 目标硬件上设置 <b>CMSISNN_PATH</b> 。
	<b>PATH</b>	Intel MKL-DNN 库文件夹的路径。 例如： <code>C:\Program Files\mkl-dnn\lib</code>
Linux	<b>LD_LIBRARY_PATH</b>	Intel MKL-DNN 库文件夹的路径。 例如： <code>/usr/local/mkl-dnn/lib/</code>
		目标硬件上 ARM Compute Library 文件夹的路径。 例如： <code>/usr/local/arm_compute/lib/</code> 在 ARM 目标硬件上设置 <b>LD_LIBRARY_PATH</b> 。
	<b>INTEL_MKLDNN</b>	Intel MKL-DNN 库安装的根文件夹的路径。 例如： <code>/usr/local/mkl-dnn/</code>
	<b>ARM_COMPUTELIB</b>	ARM 目标硬件上 ARM Compute Library 安装的根文件夹的路径。 例如： <code>/usr/local/arm_compute/</code> 在 ARM 目标硬件上设置 <b>ARM_COMPUTELIB</b> 。
	<b>CMSISNN_PATH</b>	ARM 目标硬件上 CMSIS-NN 库安装的根文件夹的路径。 例如： <code>/usr/local/cmsis_nn</code> 在 ARM 目标硬件上设置 <b>CMSISNN_PATH</b> 。
macOS	<b>INTEL_MKLDNN</b>	Intel MKL-DNN 库安装的根文件夹的路径。 例如： <code>/usr/local/mkl-dnn</code>

平台	变量名称	描述
ARM Cortex-A 目标上基于 UNIX® 的操作系统	OPENCV_DIR	OpenCV 的编译文件夹的路径。为使用 OpenCV 的深度学习示例安装 OpenCV。  例如：  <code>/usr/local/opencv/build</code>

**注意** 要使用 MATLAB Support Package for Raspberry Pi™ Hardware 为 Raspberry Pi 生成代码，您必须以非交互方式设置环境变量。有关说明，请参阅 <https://www.mathworks.com/matlabcentral/answers/455591-matlab-coder-how-do-i-setup-the-environment-variables-on-arm-targets-to-point-to-the-arm-compute-li>

**注意** 要编译和运行使用 OpenCV 的示例，您必须在目标板上安装 OpenCV 库。对于 Linux 上的 OpenCV 安装，请确保库文件的路径和头文件的路径在系统路径上。默认情况下，库文件和头文件分别安装在一个标准位置，如 `/usr/local/lib/` 和 `/usr/local/include/opencv`。

对于目标板上的 OpenCV 安装，请按照上表所述设置 `OPENCV_DIR` 和 `PATH` 环境变量。

**注意** 通过设置控制 OpenMP 线程与物理处理单元绑定的环境变量，也许可以提高为 Intel CPU 生成的代码的性能。例如，在 Linux 平台上，将 `KMP_AFFINITY` 环境变量设置为 `scatter`。对于使用 Intel CPU 的其他平台，也许可以设置类似的环境变量来提高生成代码的性能。

另请参阅

详细信息

- “Workflow for Deep Learning Code Generation with MATLAB Coder”

## 代码生成支持的网络和层

MATLAB Coder 支持序列、有向无环图 (DAG) 和循环卷积神经网络 (CNN 或 ConvNet) 的代码生成。您可以为任何经过训练的卷积神经网络生成代码，其层支持代码生成。请参阅“支持的层”（第 39-8 页）。

### 支持的预训练网络

代码生成支持 Deep Learning Toolbox 中提供的以下预训练网络。

网络名称	描述	ARM Compute Library	Intel MKL-DNN
AlexNet	AlexNet 卷积神经网络。关于预训练的 AlexNet 模型，请参阅 <code>alexnet</code> 。	是	是
DarkNet	DarkNet-19 和 DarkNet-53 卷积神经网络。对于预训练的 DarkNet 模型，请参阅 <code>darknet19</code> 和 <code>darknet53</code> 。	是	是
DenseNet-201	DenseNet-201 卷积神经网络。对于预训练的 DenseNet-201 模型，请参阅 <code>densenet201</code> 。	是	是
EfficientNet-b0	EfficientNet-b0 卷积神经网络。关于预训练的 EfficientNet-b0 模型，请参阅 <code>efficientnetb0</code> 。	是	是
GoogLeNet	GoogLeNet 卷积神经网络。关于预训练的 GoogLeNet 模型，请参阅 <code>googlenet</code> 。	是	是
Inception-ResNet-v2	Inception-ResNet-v2 卷积神经网络。有关预训练的 Inception-ResNet-v2 模型，请参阅 <code>inceptionresnetv2</code> 。	是	是
Inception-v3	Inception-v3 卷积神经网络。有关预训练的 Inception-v3 模型，请参阅 <code>inceptionv3</code> 。	是	是
MobileNet-v2	MobileNet-v2 卷积神经网络。有关预训练的 MobileNet-v2 模型，请参阅 <code>mobilenetv2</code> 。	是	是
NASNet-Large	NASNet-Large 卷积神经网络。有关预训练的 NASNet-Large 模型，请参阅 <code>nasnetlarge</code> 。	是	是
NASNet-Mobile	NASNet-Mobile 卷积神经网络。有关预训练的 NASNet-Mobile 模型，请参阅 <code>nasnetmobile</code> 。	是	是
ResNet	ResNet-18、ResNet-50 和 ResNet-101 卷积神经网络。对于预训练的 ResNet 模型，请参阅 <code>resnet18</code> 、 <code>resnet50</code> 和 <code>resnet101</code> 。	是	是
SegNet	多类像素级分割网络。有关详细信息，请参阅 <code>segnetLayers</code> 。	否	是
SqueezeNet	小型深度神经网络。有关预训练的 SqueezeNet 模型，请参阅 <code>squeezenet</code> 。	是	是
VGG-16	VGG-16 卷积神经网络。有关预训练的 VGG-16 模型，请参阅 <code>vgg16</code> 。	是	是
VGG-19	VGG-19 卷积神经网络。有关预训练的 VGG-19 模型，请参阅 <code>vgg19</code> 。	是	是

网络名称	描述	ARM Compute Library	Intel MKL-DNN
Xception	Xception 卷积神经网络。有关预训练的 Xception 模型，请参阅 <code>xception</code> 。	是	是

## 支持的层

对于表中指定的目标深度学习库，MATLAB Coder 支持以下层的代码生成。

在安装支持包 MATLAB Coder Interface for Deep Learning Libraries 后，您可以使用 `analyzeNetworkForCodegen` 查看网络是否兼容特定深度学习库的代码生成。例如：

```
result = analyzeNetworkForCodegen(mobilenetv2, TargetLibrary = 'mklDnn')
```

**注意** 从 R2022b 开始，请使用 `analyzeNetworkForCodegen` 函数检查深度学习网络的代码生成兼容性。不推荐使用 `coder.getDeepLearningLayers`。

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<code>additionLayer</code>	相加层	是	是	是
<code>anchorBoxLayer</code>	锚框层	是	是	是
<code>averagePooling2dLayer</code>	平均池化层	是	是	是
<code>batchNormalizationLayer</code>	批量归一化层	是	是	是
<code>biLstmLayer</code>	双向 LSTM 层	是	是	是
<code>classificationLayer</code>	创建分类输出层	是	是	是
<code>clippedReluLayer</code>	裁剪修正线性单元 (ReLU) 层	是	是	是
<code>concatenationLayer</code>	串联层	是	是	是
<code>convolution2dLayer</code>	二维卷积层 <ul style="list-style-type: none"> <li>对于代码生成，<code>PaddingValue</code> 参数必须等于默认值 0。</li> </ul>	是	是	是
<code>crop2dLayer</code>	对输入应用二维裁剪的层	是	是	否
<code>CrossChannelNormalizationLayer</code>	逐通道局部响应归一化层	是	是	否

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
自定义层	<p>为您的问题定义的自定义层，具有或不具有可学习参数。</p> <p>请参阅：</p> <ul style="list-style-type: none"> <li>“定义自定义深度学习层” (Deep Learning Toolbox)</li> <li>“Define Custom Deep Learning Layer for Code Generation” (Deep Learning Toolbox)</li> <li>“代码生成支持的网络和层” (第 39-7 页)</li> </ul> <p>自定义层的输出必须为固定大小数组。</p> <p>仅泛型 C/C++ 代码生成支持串行网络中的自定义层。</p> <p>对于代码生成，自定义层必须包含 <code> %#codegen pragma</code>。</p> <p>如果满足以下条件，则可以将 <code>dlarray</code> 传递给自定义层：</p> <ul style="list-style-type: none"> <li>自定义层位于 <code>dlnetwork</code> 中。</li> <li>自定义层位于 DAG 或串行网络中，并且要么从 <code>nnet.layer.Formattable</code> 继</li> </ul>	是	是	<p>是</p> <p>仅泛型 C/C++ 代码生成支持串行网络中的自定义层。</p>

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
	<p>承，要么没有反向传播。</p> <p>对于不受支持的 <b>dlarray</b> 方法，您必须从 <b>dlarray</b> 中提取基础数据，执行计算并将数据重新构造为 <b>dlarray</b> 以便生成代码。例如，</p> <pre>function Z = predict(layer, X) if coder.target('MATLAB')     Z = doPredict(X); else     if isdlarray(X)         X1 = extractdata(X);         Z1 = doPredict(X1);         Z = dlarray(Z1);     else         Z = doPredict(X);     end end end</pre>			



层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
自定义输出层	<p>所有输出层，包括通过使用 <code>nnet.layer.ClassificationLayer</code> 或 <code>nnet.layer.RegressionLayer</code> 创建的自定义分类或回归输出层。</p> <p>有关如何定义自定义分类输出层和指定损失函数的示例，请参阅“Define Custom Classification Output Layer” (Deep Learning Toolbox)。</p> <p>有关如何定义自定义回归输出层和指定损失函数的示例，请参阅“Define Custom Regression Output Layer” (Deep Learning Toolbox)。</p>	是	是	是
<code>depthConcatenationLayer</code>	深度串联层	是	是	是
<code>depthToSpace2dLayer</code>	二维深度到空间层	是	是	是
<code>dicePixelClassificationLayer</code>	Dice 像素分类层使用广义 Dice 损失为每个图像像素或体素提供分类标签。	是	是	否
<code>dropoutLayer</code>	丢弃层	是	是	是
<code>eluLayer</code>	指数线性单元 (ELU) 层	是	是	是
<code>featureInputLayer</code>	特征输入层	是	是	是
<code>flattenLayer</code>	扁平化层	是	是	是
<code>focalLossLayer</code>	焦点损失层使用焦点损失预测对象类。	是	是	是

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<code>fullyConnectedLayer</code>	全连接层	是	是	是
<code>globalAveragePooling2dLayer</code>	空间数据的全局平均池化层	是	是	是
<code>globalMaxPooling2dLayer</code>	二维全局最大池化层	是	是	是
<code>groupedConvolution2dLayer</code>	二维分组卷积层 <ul style="list-style-type: none"> <li>对于代码生成, <code>PaddingValue</code> 参数必须等于默认值 0。</li> </ul>	是 <ul style="list-style-type: none"> <li>如果为 <code>numGroups</code> 指定整数, 则该值必须小于或等于 2。</li> </ul>	是	否
<code>groupNormalizationLayer</code>	组归一化层	是	是	是
<code>gruLayer</code>	门控循环单元 (GRU) 层	是	是	是
<code>imageInputLayer</code>	图像输入层 <ul style="list-style-type: none"> <li>代码生成不支持使用函数句柄指定的 <code>'Normalization'</code>。</li> </ul>	是	是	是
<code>leakyReluLayer</code>	泄漏修正线性单元 (ReLU) 层	是	是	是
<code>lstmLayer</code>	长短期记忆 (LSTM) 层	是	是	是
<code>lstmProjectedLayer</code>	LSTM 投影层	否	否	是

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<b>maxPooling2dLayer</b>	<p>最大池化层</p> <p>如果在核窗口的非对角线上存在相等的最大值，则 <b>maxPooling2dLayer</b> 的实现差异可能导致 MATLAB 和生成的代码之间出现轻微的数值不匹配。此问题还导致每个池化区域中最大值的索引不匹配。有关详细信息，请参阅 <b>maxPooling2dLayer</b>。</p>	是	是	是
<b>maxUnpooling2dLayer</b>	<p>最大去池化层</p> <p>如果在核窗口的非对角线上存在相等的最大值，则 <b>maxPooling2dLayer</b> 的实现差异可能导致 MATLAB 和生成的代码之间出现轻微的数值不匹配。此问题还导致每个池化区域中最大值的索引不匹配。有关详细信息，请参阅 <b>maxUnpooling2dLayer</b>。</p>	否	是	否
<b>multiplicationLayer</b>	乘法层	是	是	是
<b>pixelClassificationLayer</b>	为语义分割创建像素分类层	是	是	否
<b>rcnnBoxRegressionLayer</b>	Fast 和 Faster R-CNN 的框回归层	是	是	是
<b>rpnClassificationLayer</b>	区域提议网络 (RPN) 的分类层	是	是	否
<b>regressionLayer</b>	创建回归输出层	是	是	是
<b>reluLayer</b>	修正线性单元 (ReLU) 层	是	是	是
<b>resize2dLayer</b>	二维调整大小层	是	是	是

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
scalingLayer	执行器或评价器网络的缩放层	是	是	是
sigmoidLayer	sigmoid 层	是	是	是
sequenceFoldingLayer	序列折叠层	是	是	否
sequenceInputLayer	序列输入层 <ul style="list-style-type: none"> <li>对于向量序列输入，在代码生成期间，特征的数量必须为常量。</li> <li>代码生成不支持使用函数句柄指定的 'Normalization'。</li> </ul>	是	是	是
sequenceUnfoldingLayer	序列展开层	是	是	否
softmaxLayer	Softmax 层	是	是	是
softplusLayer	执行器或评价器网络的 Softplus 层	是	是	是
spaceToDepthLayer	空间到深度层	是	是	否
ssdMergeLayer	用于目标检测的 SSD 合并层	是	是	是
swishLayer	Swish 层	是	是	是
nnet.keras.layer.ClipLayer	在上界和下界之间裁剪输入	是	是	是
nnet.keras.layer.FlattenCStyleLayer	按 C 样式（行优先）顺序，将激活值扁平化为一维	是	是	是
nnet.keras.layer.GlobalAveragePooling2dLayer	空间数据的全局平均池化层	是	是	是
nnet.keras.layer.PreluLayer	参数化修正线性单元	是	是	是
nnet.keras.layer.SigmoidLayer	Sigmoid 激活层	是	是	是
nnet.keras.layer.TanhLayer	双曲正切激活层	是	是	是

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
nnet.keras.layer.TimeDistributedFlattenCStyleLayer	按输入层的 C 样式（即行优先）存储顺序，将输入图像序列扁平化为向量序列	是	是	是
nnet.keras.layer.ZeroPadding2dLayer	二维输入的零填充层	是	是	是
nnet.onnx.layer.ClipLayer	在上界和下界之间裁剪输入	是	是	是
nnet.onnx.layer.ElementwiseAffineLayer	对输入按元素进行缩放后再相加的层	是	是	是
nnet.onnx.layer.FlattenInto2dLayer	以 ONNX 方式扁平化 MATLAB 二维图像批量，从而生成 CB 格式的二维输出数组	是	是	是
nnet.onnx.layer.FlattenLayer	ONNX™ 网络的扁平化层	是	是	是
nnet.onnx.layer.GlobalAveragePooling2dLayer	空间数据的全局平均池化层	是	是	是
nnet.onnx.layer.IdentityLayer	实现 ONNX 恒等运算符的层	是	是	是
nnet.onnx.layer.PreluLayer	参数化修正线性单元	是	是	是
nnet.onnx.layer.SigmoidLayer	Sigmoid 激活层	是	是	是
nnet.onnx.layer.TanhLayer	双曲正切激活层	是	是	是
nnet.onnx.layer.VerifyBatchSizeLayer	验证固定批量大小	是	是	是
tanhLayer	双曲正切 (tanh) 层	是	是	是

层名称	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<b>transposedConv2dLayer</b>	转置二维卷积层  代码生成不支持输入的不对称裁剪。例如，不支持为 ' <b>Cropping</b> ' 参数指定向量 [t b l r] 来裁剪输入的顶部、底部、左侧和右侧。	是	是	否
<b>wordEmbeddingLayer</b>	单词嵌入层将单词索引映射到向量。	是	是	否
<b>yolov2OutputLayer</b>	YOLO v2 目标检测网络的输出层	是	是	否
<b>yolov2ReorgLayer</b>	YOLO v2 目标检测网络的重组层	是	是	否
<b>yolov2TransformLayer</b>	YOLO v2 目标检测网络的变换层	是	是	否

## 支持的类

类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<b>DAGNetwork</b>	用于深度学习的有向无环图 (DAG) 网络  <ul style="list-style-type: none"> <li>仅支持 <b>activations</b>、<b>predict</b> 和 <b>classify</b> 方法。</li> </ul>	是	是	是

类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<b>dlnetwork</b>	<p>用于自定义训练循环的深度学习网络</p> <ul style="list-style-type: none"> <li>• 代码生成仅支持 <b>InputNames</b> 和 <b>OutputNames</b> 属性。</li> <li>• <b>dlnetwork</b> 对象的 <b>Initialized</b> 属性必须设置为 true。</li> <li>• 您可以为具有向量或图像序列输入的 <b>dlnetwork</b> 生成代码。对于 ARM Compute, <b>dlnetwork</b> 可以有序列输入层和非序列输入层。对于 Intel MKL-DNN, 输入层必须全部为序列输入层。代码生成支持包括： <ul style="list-style-type: none"> <li>• 包含具有 'CT' 或 'CBT' 数据格式的向量序列的 <b>dlarray</b>。</li> <li>• 包含具有 'SSCT' 或 'SSCBT' 数据格式的图像序列的 <b>dlarray</b>。</li> <li>• 具有异构输入层的多输入 <b>dlnetwork</b>。对于 RNN 网络, 不支</li> </ul> </li> </ul>	是	是	是

类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
	<p>持多个输入。</p> <ul style="list-style-type: none"><li>• 代码生成仅支持 <b>predict</b> 对象函数。 <b>predict</b> 方法的 <b>dlarray</b> 输入必须为 <b>single</b> 数据类型。</li><li>• 代码生成支持 MIMO <b>dlnetworks</b>。</li><li>• 要创建用于代码生成的 <b>dlnetwork</b> 对象，请参阅“加载预训练网络以用于代码生成”（第 39-23 页）。</li></ul>			
SeriesNetwork	<p>用于深度学习的串行网络</p> <ul style="list-style-type: none"><li>• 仅支持 <b>activations</b>、<b>classify</b>、<b>predict</b>、<b>predictAndUpdateState</b>、<b>classifyAndUpdateState</b> 和 <b>resetState</b> 对象函数。</li></ul>	是	是	是



类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<b>yolov2ObjectDetector</b>	<p>使用 YOLO v2 目标检测器检测目标</p> <ul style="list-style-type: none"> <li>代码生成仅支持 <b>yolov2ObjectDetector</b> 的 <b>detect</b> 方法。</li> <li><b>detect</b> 方法的 <b>roi</b> 参数必须为代码生成常量 (<b>coder.const()</b>) 和 1×4 向量。</li> <li>仅支持 <b>detect</b> 的 <b>Threshold</b>、<b>SelectStrongest</b>、<b>MinSize</b> 和 <b>MaxSize</b> 名称-值对组。</li> </ul>	是	是	否
<b>yolov3ObjectDetector</b>	<p>使用 YOLO v3 目标检测器检测目标</p> <ul style="list-style-type: none"> <li>代码生成仅支持 <b>yolov3ObjectDetector</b> 的 <b>detect</b> 方法。</li> <li><b>detect</b> 方法的 <b>roi</b> 参数必须为代码生成常量 (<b>coder.const()</b>) 和 1×4 向量。</li> <li>仅支持 <b>detect</b> 的 <b>Threshold</b>、<b>SelectStrongest</b>、<b>MinSize</b> 和 <b>MaxSize</b> 名称-值对组。</li> </ul>	是	是	是

类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
yolov4ObjectDetector	<p>使用 YOLO v4 目标检测器检测目标</p> <ul style="list-style-type: none"><li>• <code>detect</code> 方法的 <code>roi</code> 参数必须为代码生成常量 (<code>coder.const()</code>) 和 <math>1 \times 4</math> 向量。</li><li>• 仅支持 <code>detect</code> 的 <code>Threshold</code>、<code>SelectStrongest</code>、<code>MinSize</code>、<code>MaxSize</code> 和 <code>MiniBatchSize</code> 名称-值对组。</li></ul>	是	是	是

类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<b>ssdObjectDetector</b>	<p>使用基于 SSD 的检测器来检测目标的对象。</p> <ul style="list-style-type: none"> <li>代码生成仅支持 <b>ssdObjectDetector</b> 的 <b>detect</b> 方法。</li> <li><b>detect</b> 方法的 <b>roi</b> 参数必须为 <b>codegen</b> 常量 (<b>coder.const()</b>) 和 1×4 向量。</li> <li>仅支持 <b>Threshold</b>、<b>SelectStrongest</b>、<b>MinSize</b>、<b>MaxSize</b> 和 <b>MiniBatchSize</b> 名称-值对组。所有名称-值对组都必须为编译时常量。</li> <li>输入图像的通道和批量大小必须为固定大小。</li> <li><b>labels</b> 输出以分类数组形式返回。</li> <li>在生成的代码中，输入会重新缩放为网络输入层的大小。但是，<b>detect</b> 方法返回的边界框基于原始输入大小。</li> </ul>	是	是	是

类	描述	ARM Compute Library	Intel MKL-DNN	泛型 C/C++
<code>pointPillarsObjectDetector</code> (Lidar Toolbox)	用于检测激光雷达点云中的目标的 PointPillars 网络 <ul style="list-style-type: none"><li>代码生成仅支持 <code>pointPillarsObjectDetector</code> 的 <code>detect</code> 方法。</li><li>仅支持 <code>detect</code> 方法的 <code>Threshold</code>、<code>SelectStrongest</code> 和 <code>MiniBatchSize</code> 名称-值对组。</li></ul>	是	是	否

int8 代码生成

您可以将 Deep Learning Toolbox 与 Deep Learning Toolbox 模型量化库支持包结合使用，通过将卷积层的权重、偏置和激活量化为 8 位定标整数数据类型来减少深度神经网络的内存占用。然后，您可以使用 MATLAB Coder 来生成网络的优化代码。请参阅“Generate int8 Code for Deep Learning Networks”。

另请参阅  
`coder.getDeepLearningLayers`

详细信息

- “预训练的深度学习” (Deep Learning Toolbox)
- “了解卷积神经网络” (Deep Learning Toolbox)
- “Workflow for Deep Learning Code Generation with MATLAB Coder”

## 加载预训练网络以用于代码生成

您可以为经过预训练的卷积神经网络 (CNN) 生成代码。要向代码生成器提供网络，请从经过训练的网络中加载一个 `SeriesNetwork`、`DAGNetwork`、`yolov2ObjectDetector`、`ssdObjectDetector` 或 `dlnetwork` 对象。

### 使用 `coder.loadDeepLearningNetwork` 加载网络

您可以使用 `coder.loadDeepLearningNetwork` 从任何支持代码生成的网络加载网络对象。您可以从 MAT 文件中指定网络。MAT 文件只能包含要加载的网络。

例如，假设您使用 `trainNetwork` 函数创建了一个名为 `myNet` 的经过训练的网络对象。然后，您可以通过输入 `save` 来保存工作区。这将创建一个名为 `matlab.mat` 的文件，其中包含网络对象。要加载网络对象 `myNet`，请输入：

```
net = coder.loadDeepLearningNetwork('matlab.mat');
```

您还可以通过以下方式指定网络：提供不接受输入参数并返回预训练的 `SeriesNetwork`、`DAGNetwork`、`yolov2ObjectDetector` 或 `ssdObjectDetector` 对象的函数的名称，例如：

- `alexnet`
- `densenet201`
- `googlenet`
- `inceptionv3`
- `mobilenetv2`
- `resnet18`
- `resnet50`
- `resnet101`
- `squeezenet`
- `vgg16`
- `vgg19`
- `xception`

例如，通过输入以下命令加载网络对象：

```
net = coder.loadDeepLearningNetwork('googlenet');
```

上述列表中的 Deep Learning Toolbox 函数要求您安装适用于这些函数的支持包。请参阅“预训练的深度学习神经网络” (Deep Learning Toolbox)。

### 为代码生成指定网络对象

如果您使用 `codegen` 或 App 生成代码，请使用 `coder.loadDeepLearningNetwork` 将网络对象加载到您的入口函数内。例如：

```
function out = myNet_predict(in) %#codegen
persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('matlab.mat');
```

```
end
out = predict(mynet,in);
```

对于可用作支持包函数（如 `alexnet`、`inceptionv3`、`googlenet` 和 `resnet`）的预训练网络，您可以直接指定支持包函数，例如，通过编写 `mynet = googlenet`。

接下来，为入口函数生成代码。例如：

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -args {ones(224,224,3,'single')} -config cfg myNet_predict
```

## 为代码生成指定 dlnetwork 对象

假设在 MAT 文件 `mynet.mat` 中有一个预训练的 `dlnetwork` 网络对象。要预测此网络的响应，请在 MATLAB 中创建一个入口函数，如以下代码所示。

```
function a = myDLNet_predict(in)
dlIn = dlarray(in, 'SSC');

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dIA = predict(dlnet, dlIn);

a = extractdata(dIA);

end
```

在此示例中，`myDLNet_predict` 的输入和输出属于更简单的数据类型，并且 `dlarray` 对象是在该函数中创建的。`dlarray` 对象的 `extractdata` 方法在 `dlarray dIA` 中返回数据作为 `myDLNet_predict` 的输出。输出 `a` 与 `dIA` 中的基础数据类型具有相同的数据类型。这种入口函数设计具有以下优点：

- 更容易与独立的代码生成工作流（如静态、动态库或可执行文件）集成。
- `extractdata` 函数输出的数据格式在 MATLAB 环境和生成代码中具有相同的顺序 ('SCBTU')。
- 可提高 MEX 工作流的性能。
- 可使用 MATLAB Function 模块简化 Simulink 工作流，因为 Simulink 不内生支持 `dlarray` 对象。

接下来，为入口函数生成代码。例如：

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -args {ones(224,224,3,'single')} -config cfg myDLNet_predict
```

## 另请参阅

### 函数

`codegen` | `trainNetwork` | `coder.loadDeepLearningNetwork`

## 对象

SeriesNetwork | DAGNetwork | yolov2ObjectDetector | ssdObjectDetector | dlarray |  
dlnetwork

## 详细信息

- “代码生成支持的网络和层” (第 39-7 页)
- “Code Generation for Deep Learning Networks with MKL-DNN”
- “Code Generation for Deep Learning Networks with ARM Compute Library”





## 生成 C++ 代码

---

C++ 代码生成

MATLAB Coder 使您能够生成 C 或 C++ 代码。默认情况下，代码生成器生成 C 代码。生成的 C++ 代码可以使用 C 语言中没有的功能，这些功能可以使 C++ 代码更加易读和易于使用。

生成 C++ 代码

要生成 C++ 代码，请遵循与生成 C 代码相同的整体工作流步骤。例如，请参阅“通过命令行生成 C 代码”。从命令行中或使用代码生成配置设置或者从 MATLAB Coder 中选择 C++ 语言选项。

假设您要为接受零输入的函数 `foo` 生成 C++ 代码：

- 从命令行中，使用 `-lang:c++` 设定符。此设定符提供一种快速简单的生成 C++ 代码的方法。例如，要为 `foo` 生成 C++ 静态库和 C++ 源代码，请输入：

```
codegen -config:lib -lang:c++ foo
```

- 在配置对象中，将 `TargetLang` 参数设置为 C++。例如，要生成 C++ 动态库，请输入：

```
cfg = coder.config('dll');
cfg.TargetLang = 'C++';
codegen -config cfg foo
```

- 在 App 的生成代码步骤中，选择 C++ 语言按钮。

生成的代码中支持的 C++ 语言功能

要了解利用重要的 C++ 语言功能的代码生成，请参考以下帮助主题：

目标	更多信息
为您的 MATLAB 代码中的类生成 C++ 类。	“Generate C++ Classes for MATLAB Classes”
将入口函数生成成为 C++ 类中的方法。	“Generate C++ Code with a Class Interface”
为 MATLAB 包生成 C++ 命名空间。将所有生成的代码放在指定的命名空间中。将为 MathWorks 代码生成的所有代码放在您指定的命名空间中。	“Organize Generated C++ Code into Namespaces”
在自定义 C++ 代码和生成的代码之间传递动态分配的数组。生成的 C++ 代码通过使用 <code>coder::array</code> 类模板来实现这样的数组。生成的代码提供简单的 API，您可以使用它与该模板进行交互。	“Use Dynamically Allocated C++ Arrays in Generated Function Interfaces”

这些示例说明这些功能的使用：

- “Generate C++ Classes for MATLAB Classes That Model Simple and Damped Oscillators”
- “Integrate Multiple Generated C++ Code Projects”

生成的 C 代码和 C++ 代码之间的其他区别

如果为同一个 MATLAB 函数分别生成 C 和 C++ 代码，并检查生成的源代码，可以发现在实现上的差异。这些是一些显著的差异：

- 生成的 C++ 代码可包含具有多个签名的同名重载函数或方法。C 语言不支持函数重载。

- 生成的 C++ 代码可跨不同命名空间层次结构重用相同的标识符名称。例如，相同的类型名称 `myType` 可以出现在两个不同命名空间层次结构中，顶层命名空间为 `myNamespace_1` 和 `myNamespace_2`。C 语言不支持命名空间和这种标识符名称重用。
- 在生成的 C 代码中，函数头包含为生成的 C 函数指定 `extern "C"` 标识符的 `#ifdef __cplusplus` `include` 防卫式声明。编译器和链接器在构建作为 C++ 工程一部分的 C 代码时使用这些标识符。
- 生成的 C++ 代码的 C++ 文件使用 `.cpp` 文件扩展名，头文件使用 `.h` 文件扩展名。生成的 C 代码使用 `.c` 和 `.h` 扩展。
- 生成的 C++ 代码使用一些 C++ 强制转换，例如 `static_cast`，比 C 语言中的强制转换语法更加显式。
- 生成的代码基于 C++ 和 C 的不同机制为 `Inf` 和 `NaN` 定义值。
- 生成的 C++ 代码使用“Mapping MATLAB Types to Types in Generated Code”中所述的自定义数据类型。
- 生成的 C++ 代码使用与生成的 C 代码不同的库。例如，“Change the Language Standard”中描述了 C++ 和 C 的默认语言标准。

## 另请参阅

`codegen`

## 详细信息

- “配置编译设置”（第 27-12 页）



# 仿真数据检查器

---

- “在仿真数据检查器中查看数据” (第 41-2 页)
- “将 CSV 文件中的数据导入仿真数据检查器” (第 41-10 页)
- “仿真数据检查器如何比较数据” (第 41-15 页)
- “保存和共享仿真数据检查器数据和视图” (第 41-19 页)
- “以编程方式检查和比较数据” (第 41-25 页)

## 在仿真数据检查器中查看数据

您可以使用仿真数据检查器来可视化您在整个设计过程中生成的数据。您在 Simulink 模型中记录的仿真数据会记录到仿真数据检查器中。您还可以将测试数据和其他记录的数据导入仿真数据检查器，以便与记录的仿真数据一起进行检查和分析。仿真数据检查器提供几种类型的绘图，以便您轻松创建复杂的数据可视化。

### 查看记录的数据

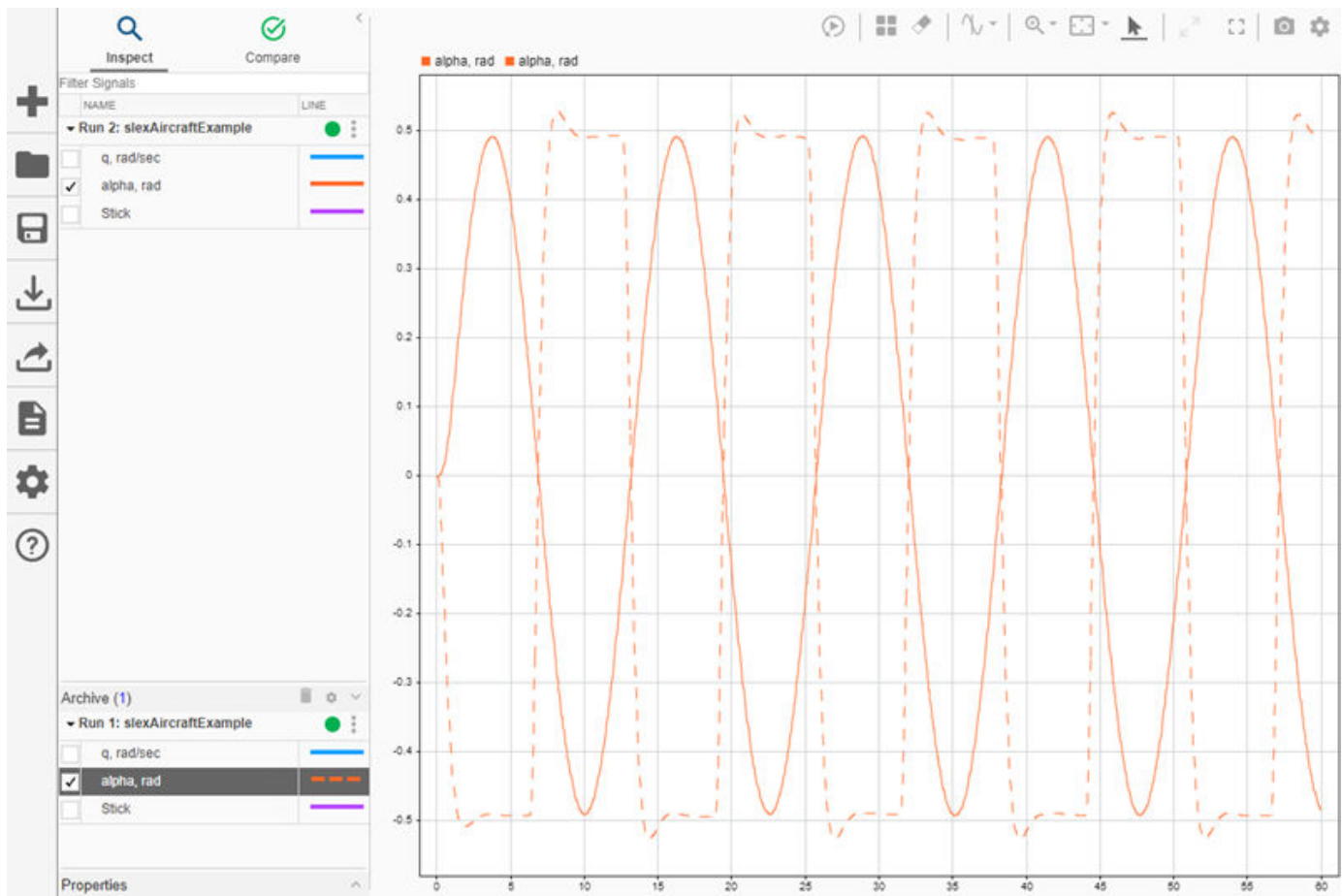
在仿真模型时，使用 **Dataset** 格式记录的信号以及输出和状态会自动记录到仿真数据检查器中。您也可以记录其他种类的仿真数据，以便数据在仿真结束时出现在仿真数据检查器中。要在仿真数据检查器中查看以 **Dataset** 以外的格式记录的状态和输出数据，请打开“配置参数”对话框，并在**数据导入/导出**窗格中选择**在仿真数据检查器中记录所记录的工作区数据**参数。

---

**注意** 当您使用 **Structure** 或 **Array** 格式记录状态和输出时，您还必须记录数据记录到仿真数据检查器的时间。

---

仿真数据检查器在**检查**窗格的表中显示可用数据。要绘制信号，请选中信号旁边的复选框。您可以修改布局并添加不同可视化来分析仿真数据。有关详细信息，请参阅“Create Plots Using the Simulation Data Inspector” (Simulink)。



仿真数据检查器使用存档管理传入的仿真数据。默认情况下，当您开始新仿真时，上一次运行会移至存档中。您可以从存档中绘制信号，也可以将关注的运行拖回工作区域中。

## 从工作区或文件导入数据

您可以从基础工作区或文件导入数据，以单独查看或与仿真数据一起查看。仿真数据检查器支持从工作区导入数据的所有内置数据类型和许多数据格式。一般情况下，无论何种格式，采样值都必须与采样时间成对出现。在基于导入的工作区数据创建的一个运行中，仿真数据检查器允许每个信号最多有 8000 个信道。

您也可以从下列类型的文件中导入数据：

- MAT 文件
- CSV 文件 - 如“将 CSV 文件中的数据导入仿真数据检查器” (Simulink) 中所示的格式化数据。
- Microsoft Excel® 文件 - 如“Microsoft Excel Import, Export, and Logging Format” (Simulink) 中所述的格式化数据。
- MDF 文件 - Linux 和 Windows 操作系统支持 MDF 文件导入。MDF 文件必须具有 .mdf、.mf4、.mf3、.data 或 .dat 文件扩展名，并且只包含整数和浮点数据类型的数据。
- ULG 文件 - 飞行日志数据导入需要 UAV Toolbox 许可证。

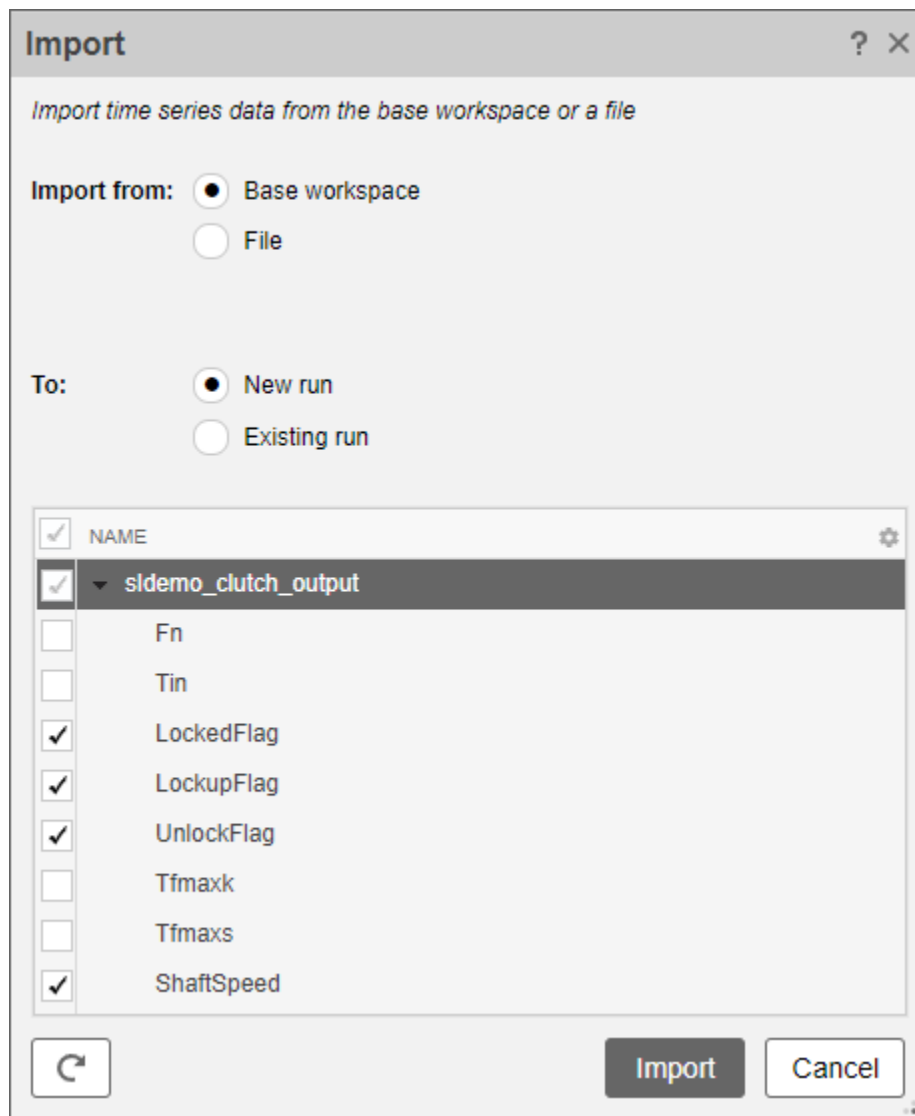
要从工作区或以仿真数据检查器不支持的数据或文件格式保存的文件中导入数据，您可以编写自己的工作区数据或文件读取器以使用 `io.reader` 类导入数据。您还可以编写自定义读取器来代替内置读取器用于支持的文件类型。有关示例，请参阅：

- “Import Data Using a Custom File Reader” (Simulink)
- “Import Workspace Variables Using a Custom Data Reader” (Simulink)



要导入数据，请在仿真数据检查器中选择**导入**按钮。

在“导入”对话框中，您可以选择从工作区或文件中导入数据。选项下方的表显示可用于导入的数据。如果您在表中没有看到您的工作区变量或文件内容，这意味着仿真数据检查器没有支持该数据的内置或注册的读取器。您可以使用复选框选择要导入的数据，并可以选择是将该数据导入现有运行中还是新运行中。



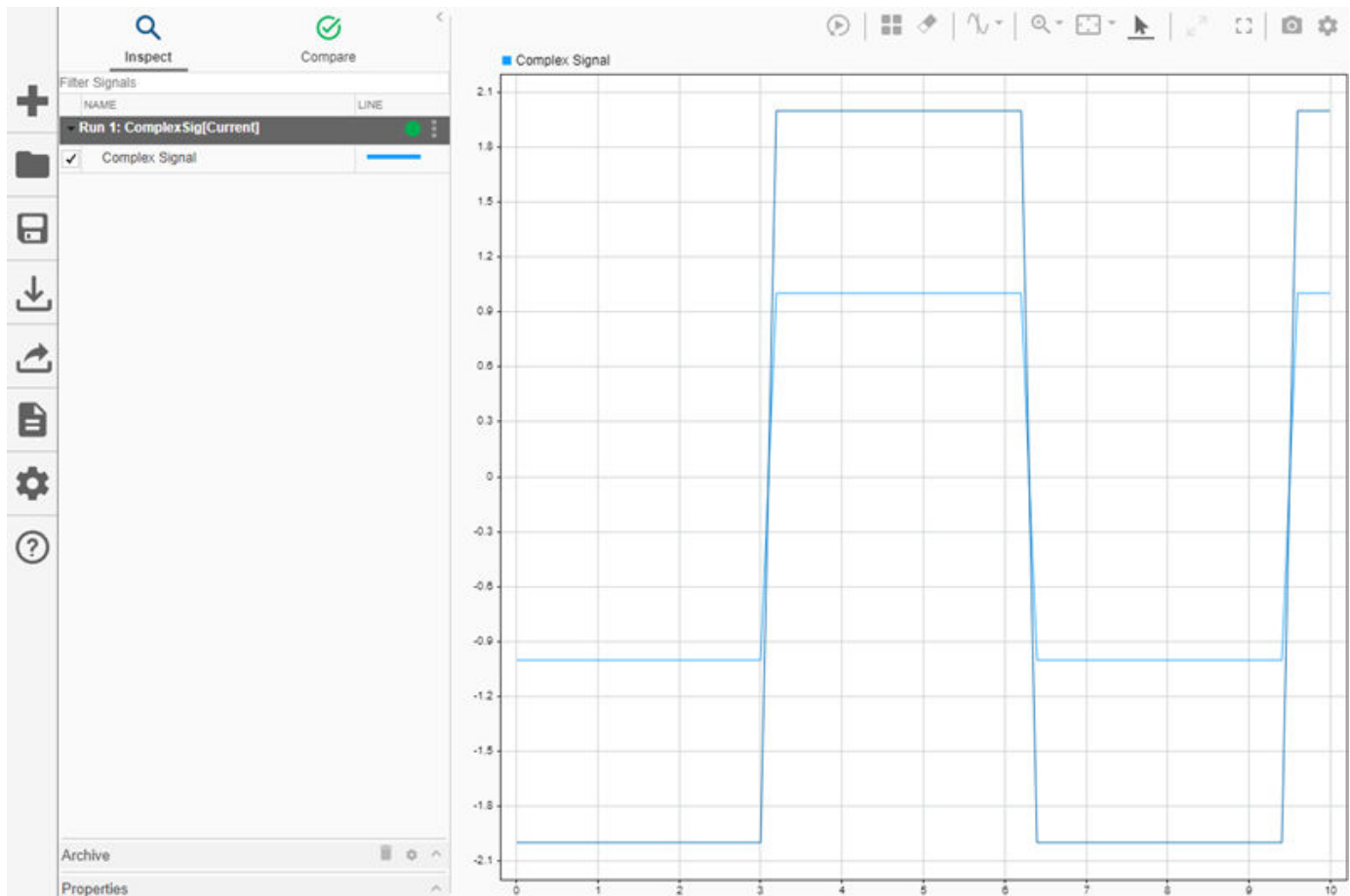
当您导入数据到新运行中时，该运行始终显示在工作区中。您可以手动将导入的运行移至存档中。



## 查看复数数据

要在仿真数据检查器中查看复数数据，请导入数据或将信号记录到仿真数据检查器中。对于模型中的信号，您可以使用仿真数据检查器中的**检测属性**中的**属性**窗格来控制如何可视化复信号。要访问某信号的**检测属性**，请右键点击该信号的记录标记，然后选择**属性**。

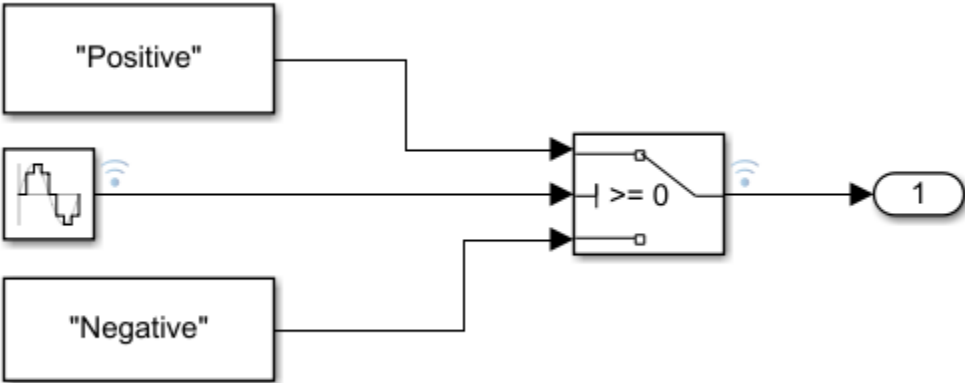
您可以将**复信号格式**指定为“幅值”、“幅值-相位”、“相位”或“实部-虚部”。如果您为**复信号格式**选择“幅值-相位”或“实部-虚部”，则当您为信号选中此复选框时，仿真数据检查器将同时绘制该信号的实部和虚部。对于“实部-虚部”格式的信号，**线条颜色**指定信号的实部的颜色，而虚部则以该**线条颜色**的不同深度显示。例如，下方图中的 **Rectangular QAM Modular Baseband** 信号以浅蓝色显示信号的实部（与**线条颜色**参数匹配），虚部则显示为较深的蓝色。



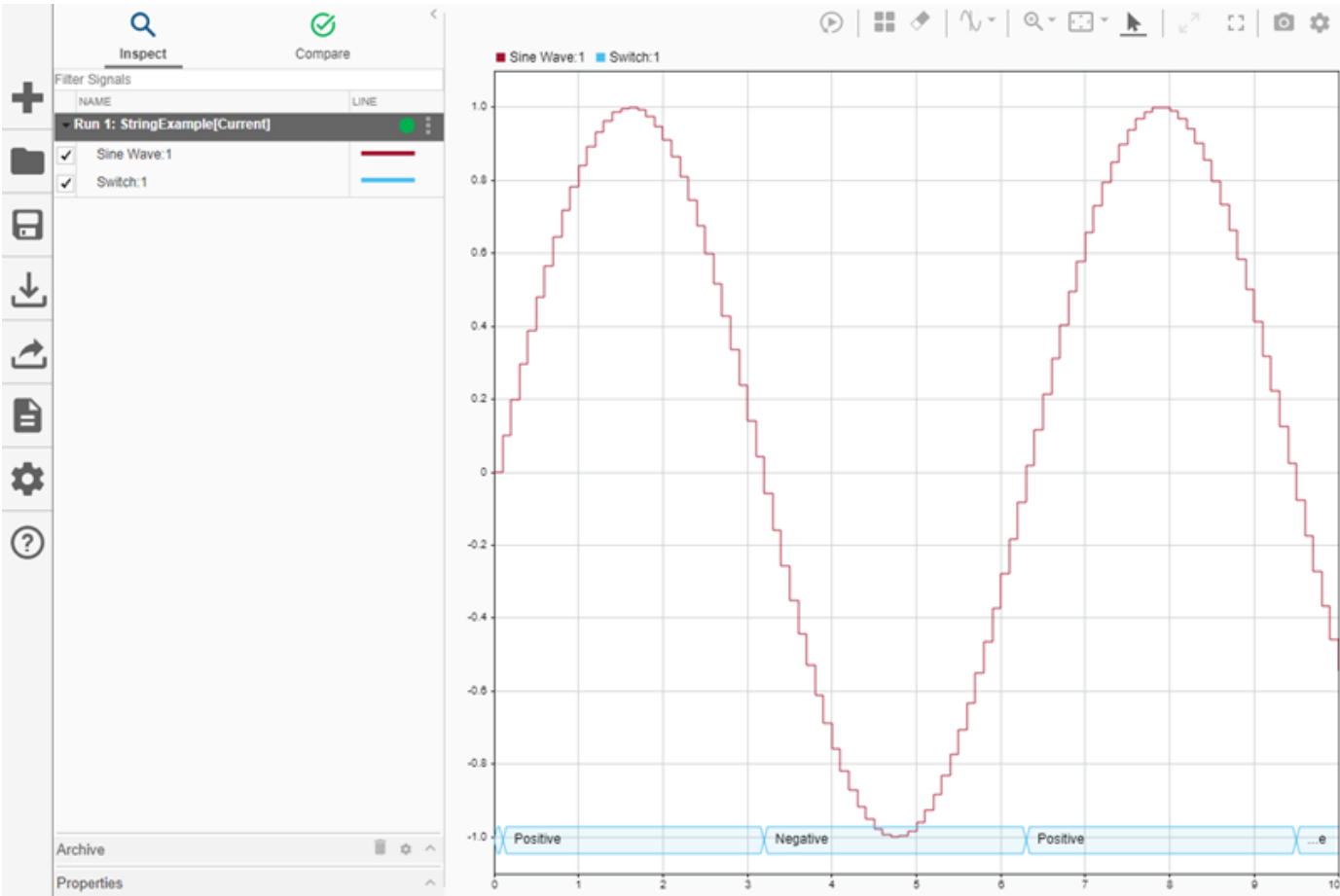
对于“幅值-相位”格式的信号，**线条颜色**指定幅值分量的颜色，相位则以该**线条颜色**的不同深度显示。

## 查看字符串数据

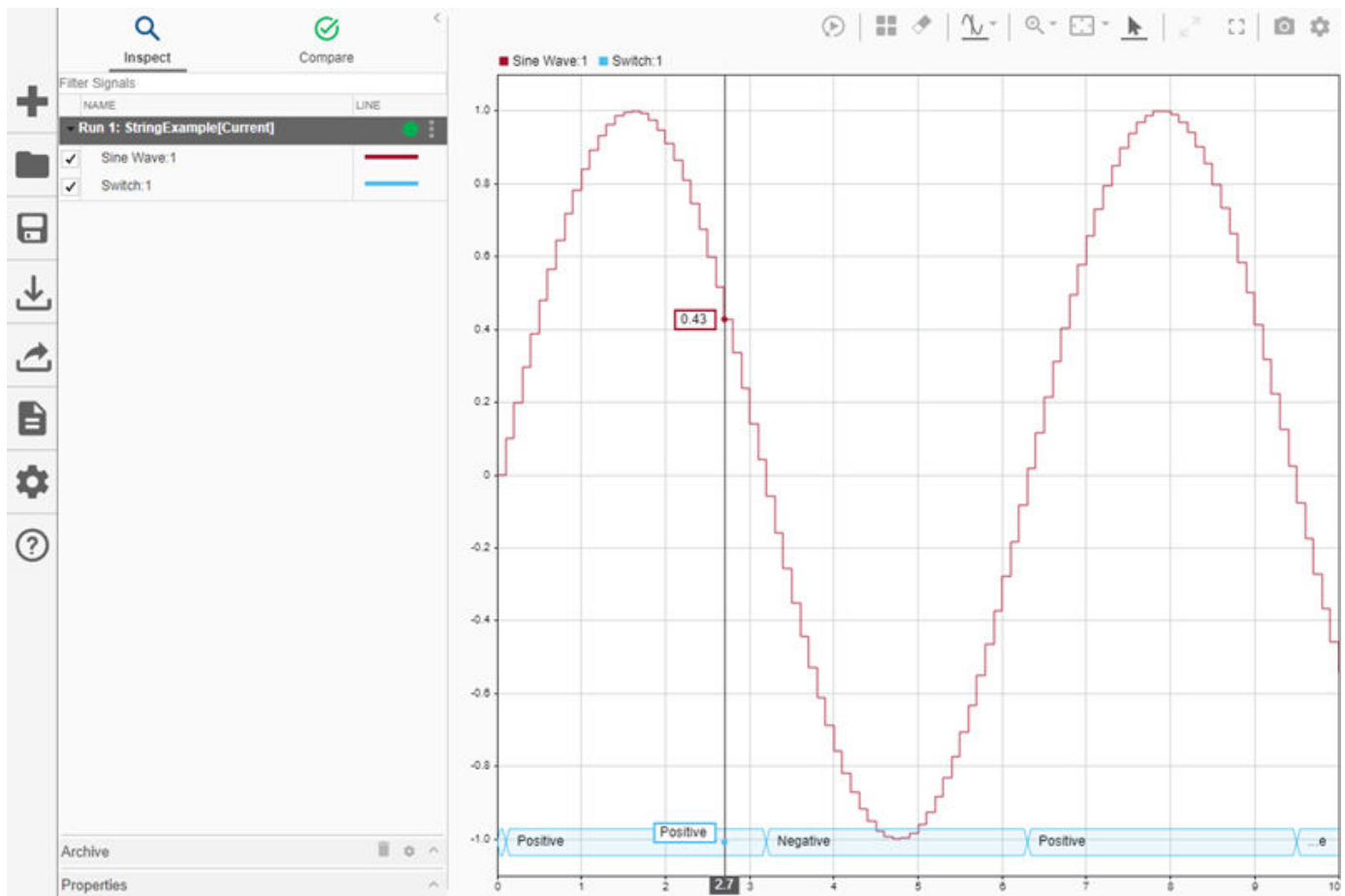
您可以在仿真数据检查器中使用信号数据来记录和查看字符串数据。假设有以下简单模型。正弦波模块的值控制开关是否向输出发送读取 **Positive** 或 **Negative** 的字符串。



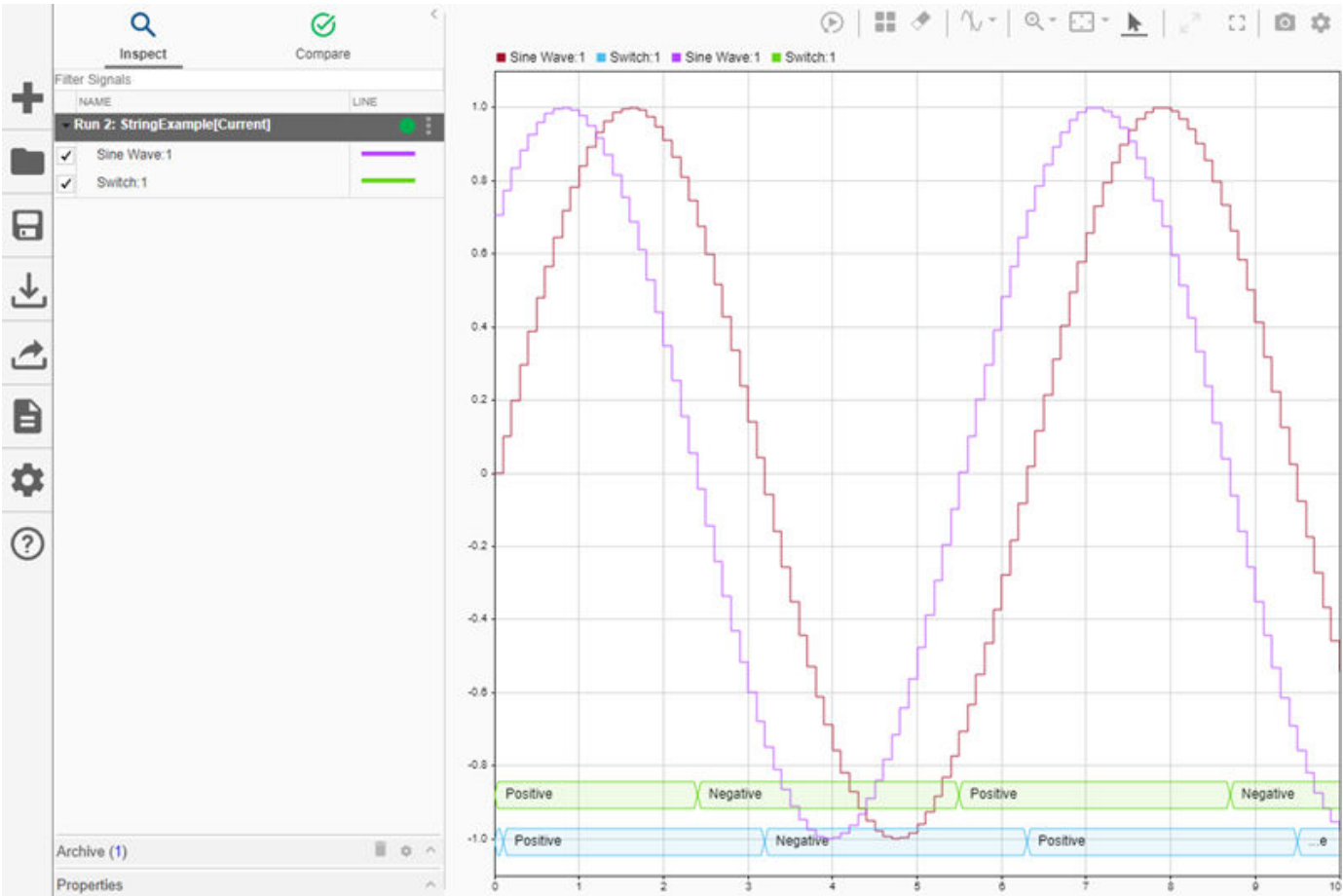
下图显示模型的仿真结果。字符串信号显示在图形查看区域的底部。信号值显示在条带内，字符串信号值中的转换用交叉线进行标记。



您可以使用游标来检查字符串信号值与正弦信号值的对应关系。



当您在绘图上绘制多个字符串信号时，信号会按照它们仿真或导入的顺序堆叠，最新的信号位于最上层。例如，您可能想了解更改控制开关的正弦波的相位会有什么影响。

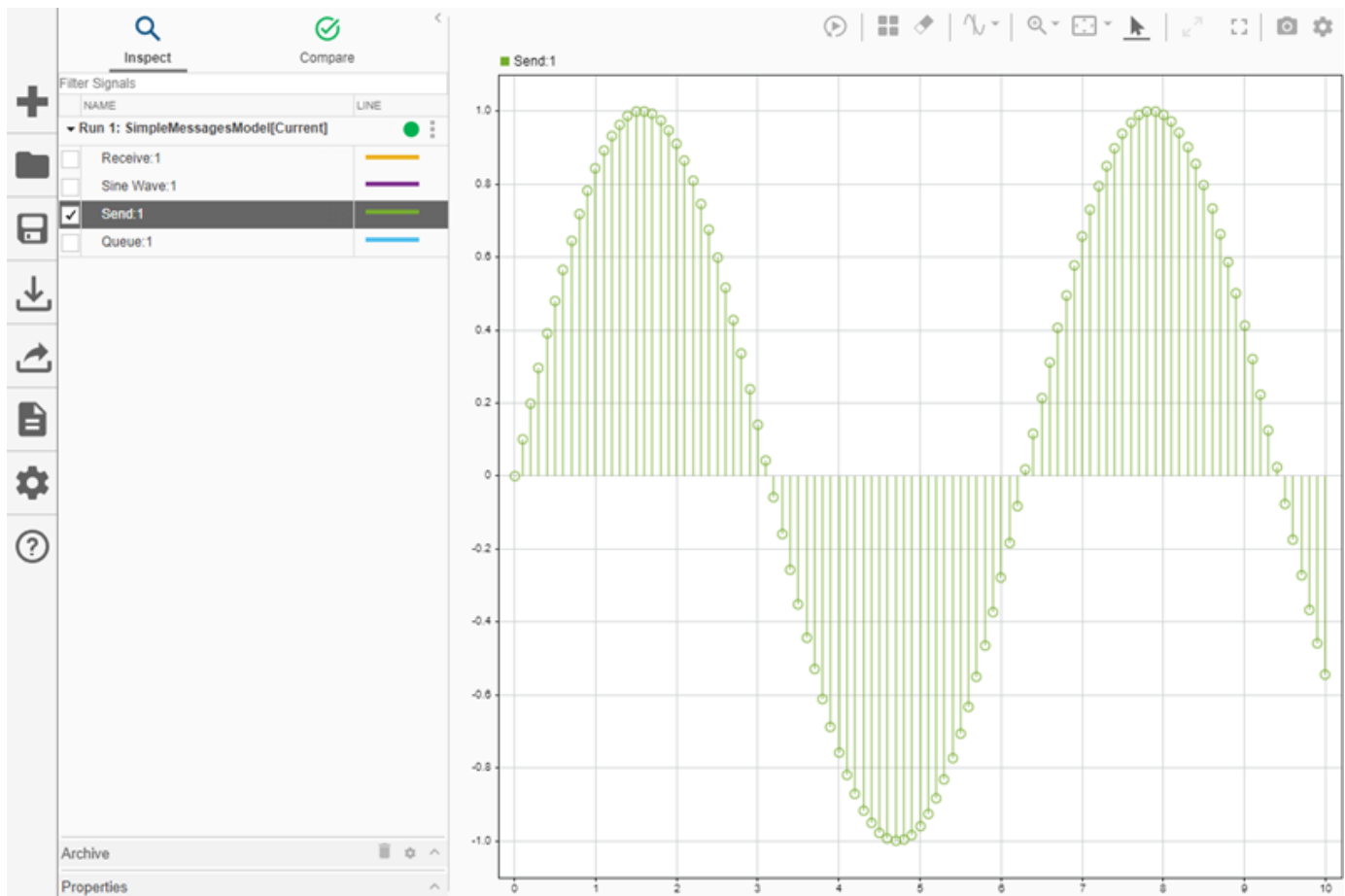


查看基于帧的数据

以帧为单位（而不是逐点）处理数据可在某些应用场景实现所需的性能提升。要在仿真数据检查器中查看基于帧的数据，必须在**检测属性**中将信号指定为基于帧。要访问信号的**检测属性**对话框，请右键点击信号的记录标记并选择**属性**。要将信号指定为基于帧，请为**输入处理**选择**列作为通道(基于帧)**。

查看基于事件的数据

您可以将事件数据记录到或导入仿真数据检查器。要查看记录的基于事件的数据，请选中 **Send: 1** 旁边的复选框。仿真数据检查器将数据显示为针状图，其中每个针状代表给定采样时间内发生的事件数量。



## 另请参阅

### 详细信息

- 检查仿真数据 (Simulink)
- 比较仿真数据 (Simulink)
- 共享仿真数据检查器数据和视图 (第 41-19 页)
- 决定如何可视化数据 (Simulink)
- 所记录数据的数据集转换 (Simulink)

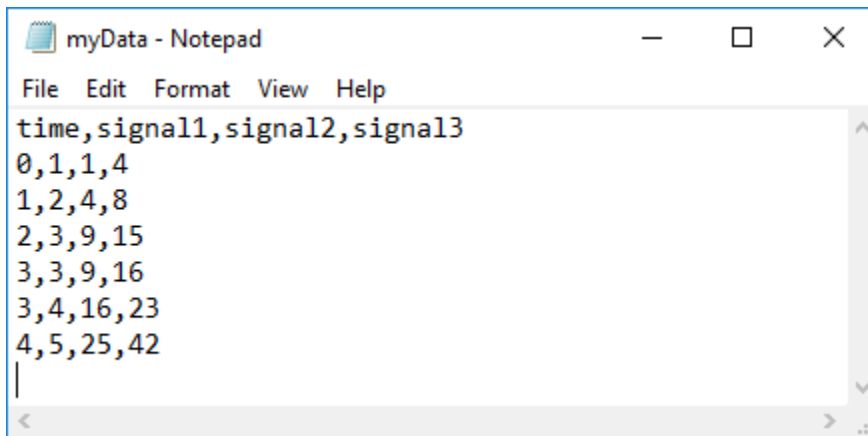
## 将 CSV 文件中的数据导入仿真数据检查器

要将数据从 CSV 文件导入仿真数据检查器，请格式化该 CSV 文件中的数据。然后，您可以使用仿真数据检查器 UI 或 `Simulink.sdi.createRun` 函数导入数据。

**提示** 当您要从 CSV 文件导入数据时，如果其中的数据格式与本主题中的设定不同，您可以使用 `io.reader` 类为仿真数据检查器编写您自己的文件读取器。

### 基本文件格式

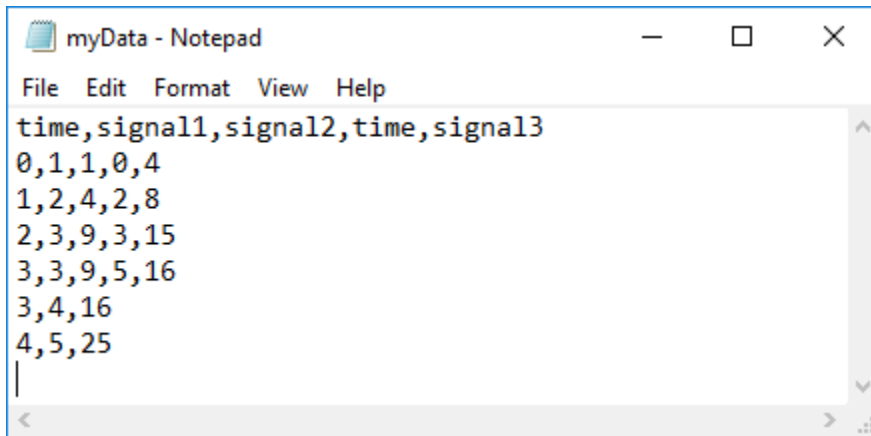
在最简单的格式中，CSV 文件的第一行是标题，列出文件中信号的名称。第一列是时间。时间列的名称必须为 `time`，并且时间值必须单调递增。信号名称以下的各行列出对应于每个时间步的信号值。



导入操作不支持包含 `Inf` 或 `NaN` 值的时间数据或包含 `Inf` 值的信号数据。空值或 `NaN` 信号值呈现为缺失数据。支持所有内置数据类型。

### 多个时间向量

当数据包含具有不同时间向量的信号时，文件可以包含多个时间向量。每个时间列必须命名为 `time`。时间列指定其右边各信号列的采样时间，直到下一个时间向量列为止。例如，第一个时间列定义 `signal1` 和 `signal2` 的时间，第二个时间列定义 `signal3` 的时间步。



信号列必须具有与相关联的时间向量相同的数据点数。

## 信号元数据

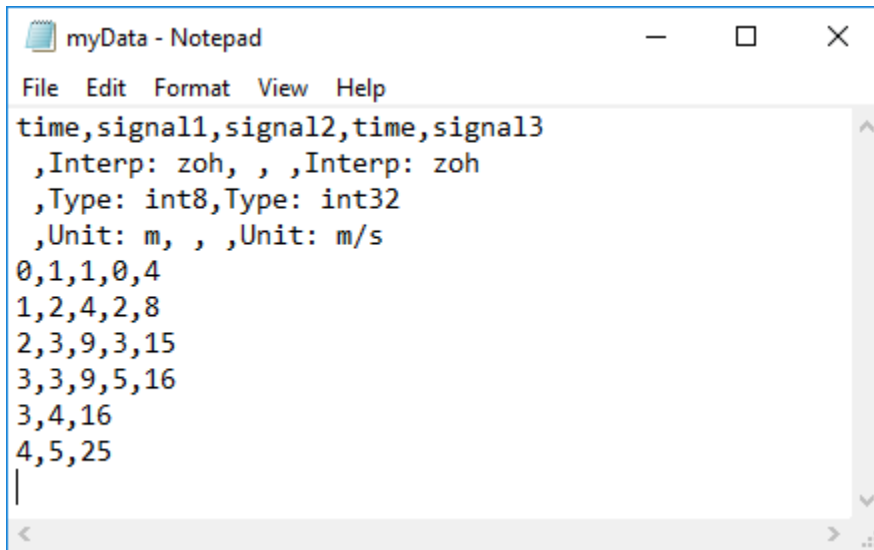
您可以在 CSV 文件中指定信号元数据，以指示信号数据类型、单位、插值方法、模块路径和端口索引。在信号名称和信号数据之间的行中列出每个信号的元数据。根据下表对元数据进行标记。

信号属性	标签	值
数据类型	<b>Type:</b>	内置数据类型。
单位	<b>Unit:</b>	支持的单位。例如，Unit: m/s 指定单位“米/秒”。  有关支持的设备列表，请在 MATLAB 命令行窗口中输入 <b>showunitslist</b> 。
插值方法	<b>Interp:</b>	<b>linear</b> 、 <b>zoh</b> （零阶保持）或 <b>none</b> 。
模块路径	<b>BlockPath:</b>	生成信号的模块的路径。
端口索引	<b>PortIndex:</b>	整数。

您也可以导入数据类型由枚举类定义的信号。请使用 **Enum:** 标签并将值指定为枚举类的名称，而不是使用 **Type:** 标签。枚举类的定义必须保存在 MATLAB 路径中。

当导入的文件未指定信号元数据时，仿真数据检查器会采用双精度数据类型和线性插值。您可以将插值方法指定为 **linear**、**zoh**（零阶保持）或 **none**。如果未在文件中指定信号的单位，则可以在导入文件后，在仿真数据检查器中为信号指定单位。

您可以为每个信号指定任意元数据组合。对元数据指定不足的信号，在相应位置保留一个空元胞即可。



```
myData - Notepad
File Edit Format View Help
time,signal1,signal2,time,signal3
,Interp: zoh, , ,Interp: zoh
,Type: int8,Type: int32
,Unit: m, , ,Unit: m/s
0,1,1,0,4
1,2,4,2,8
2,3,9,3,15
3,3,9,5,16
3,4,16
4,5,25
|
```

## 从 CSV 文件导入数据

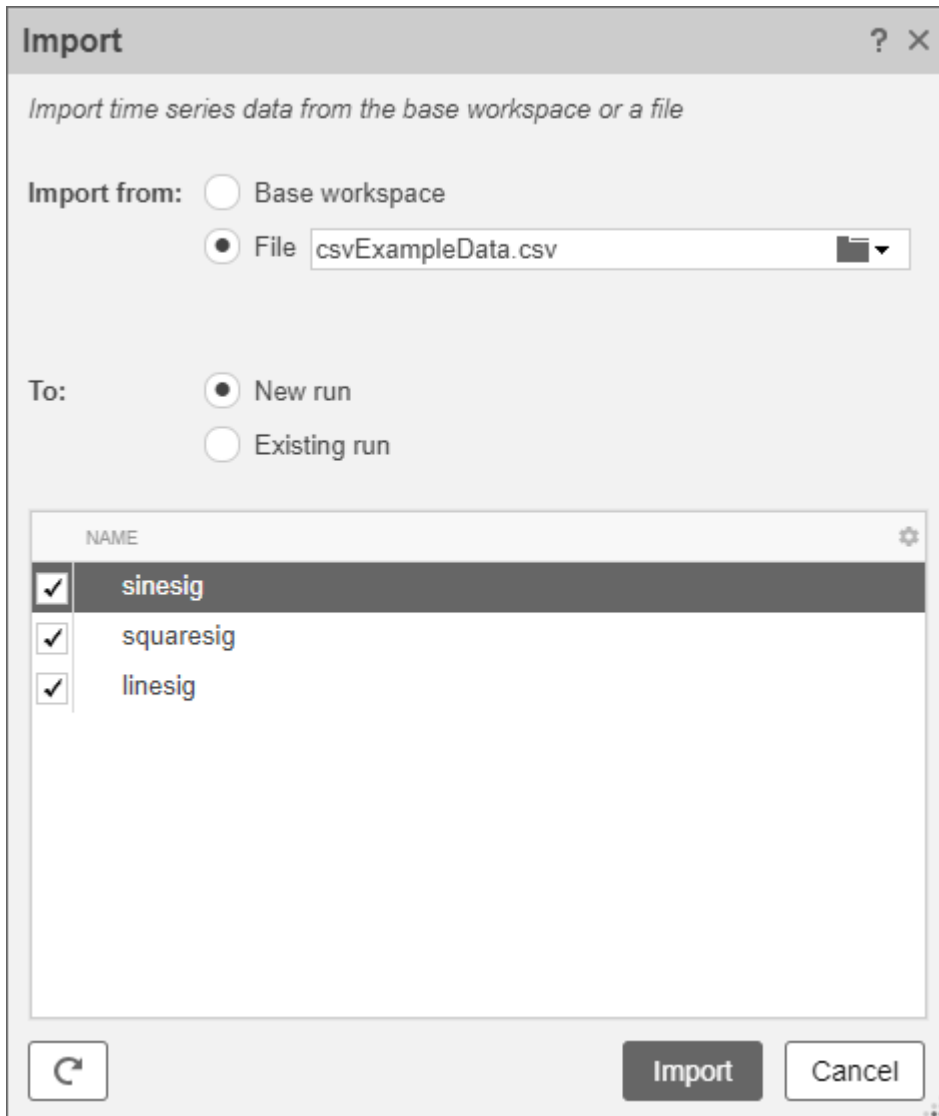
您可以使用仿真数据检查器 UI 或使用 `Simulink.sdi.createRun` 函数从 CSV 文件导入数据。

要使用 UI 导入数据，请使用 `Simulink.sdi.view` 函数或 Simulink™ 工具条中的**数据检查器**按钮打开仿真数据检查器。然后，点击**导入**按钮。



在“导入”对话框中，选择从文件导入数据的选项，并在文件系统中导航以选择文件。选择文件后，可用于导入的数据会显示在表中。您可以选择导入哪些信号，以及是将它们导入新运行还是现有运行中。此示例将所有可用信号导入新运行中。选择选项后，点击**导入**按钮。





当您使用 UI 将数据导入新运行时，新运行名称在运行编号后会跟有 **Imported\_Data**。

以编程方式导入数据时，您可以指定导入的运行的名称。

```
csvRunID = Simulink.sdi.createRun('CSV File Run','file','csvExampleData.csv');
```

## 另请参阅

### 函数

`Simulink.sdi.createRun`

## 详细信息

- “在仿真数据检查器中查看数据” (Simulink)
- “Microsoft Excel Import, Export, and Logging Format” (Simulink)

- “Import Data Using a Custom File Reader” (Simulink)

## 仿真数据检查器如何比较数据

您可以定制仿真数据检查器比较过程，以多种方式满足您的需求。在比较各运行时，仿真数据检查器会执行以下操作：

- 1 根据**对齐**设置，对齐**基线**运行和**比较项**运行中的信号对组。

仿真数据检查器不会比较无法对齐的信号。

- 2 根据指定的**同步方法**同步对齐的信号对组。

根据指定的**插值方法**，对同步中添加的时间点值进行插值。

- 3 计算信号对组的差值。

- 4 将差值结果与指定的容差进行比较。

在比较运行完成后，比较的结果将显示在导航窗格中。

状态	比较结果
	差值在指定的容差范围内。
	差值违反指定的容差。
	信号未与来自 <b>比较项</b> 运行的信号对齐。

当您比较具有不同时间区间的信号时，仿真数据检查器会比较其重叠区间上的信号。

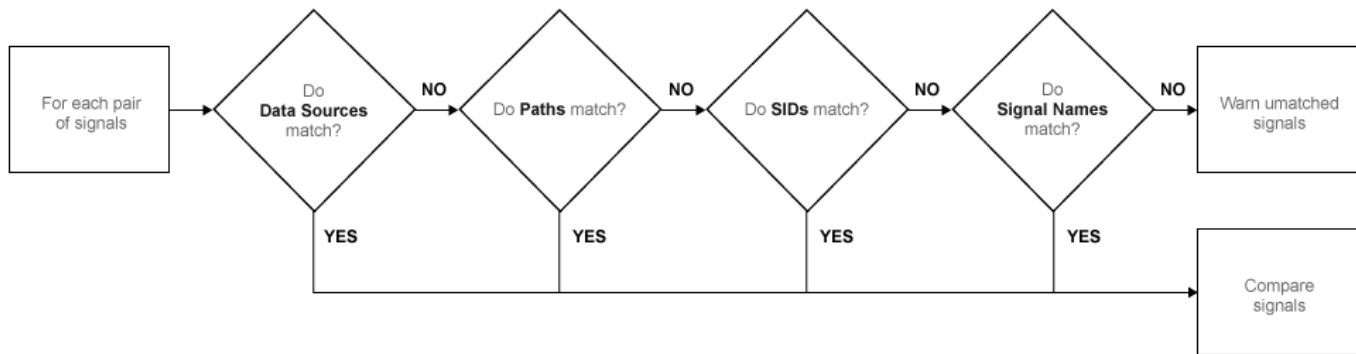
## 信号对齐

在对齐步骤中，仿真数据检查器决定来自**比较项**运行的哪个信号与**基线**运行中的给定信号配对。当您使用仿真数据检查器比较信号时，可以通过选择**基线**和**比较项**信号来完成对齐步骤。

仿真数据检查器使用信号的数据源、路径、SID 和信号名称属性的组合来对齐信号。

属性	描述
数据源	从工作区导入的数据在 MATLAB 工作区中的变量路径
路径	模型中数据源的模块路径
SID	自动分配的 Simulink 标识符
信号名称	模型中信号的名称

使用默认对齐设置时，仿真数据检查器会根据此流程图在各运行之间对齐信号。

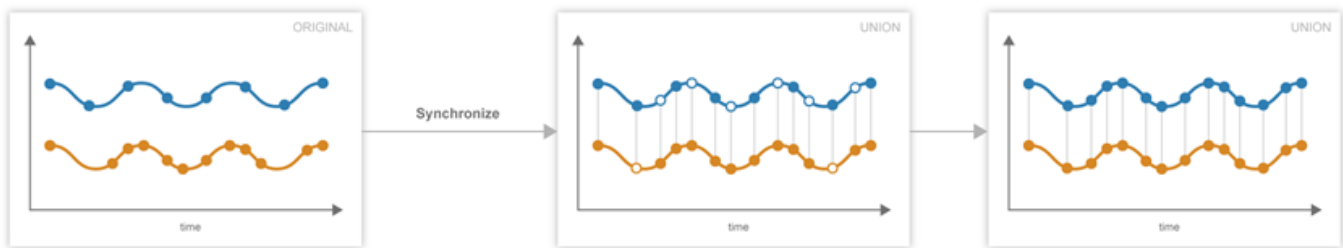


您可以在仿真数据检查器的**预设项**中为用于对齐的每个信号属性指定优先级。**对齐方式**字段指定用于对齐信号的最高优先级属性。优先级随着每个后续的**然后依据**字段依次下降。您必须在**对齐方式**字段中指定主要对齐属性，但您可以将任何数量的**然后依据**字段留空。

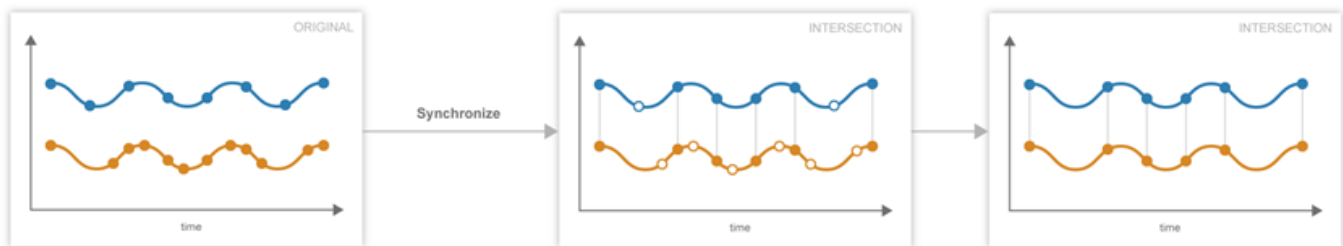
## 同步

通常，您要比较的信号包含的时间点集合不会完全相同。仿真数据检查器比较中的同步步骤解决信号的时间向量的差异。您可以选择 **union** 或 **intersection** 作为同步方法。

当您指定 **union** 同步时，仿真数据检查器会构建一个时间向量，其中包括两个信号之间的每个采样时间。对于两个信号中最初都不存在的每个采样时间，仿真数据检查器会对其进行插值。示意图中的第二个图显示联合同步过程，其中仿真数据检查器识别要添加到每个信号中的采样，由非实心圆表示。最终绘图显示仿真数据检查器对添加的时间点进行插值后的信号。仿真数据检查器使用最终绘图中的信号计算差值，以便计算的差值信号包含信号之间的所有数据点。



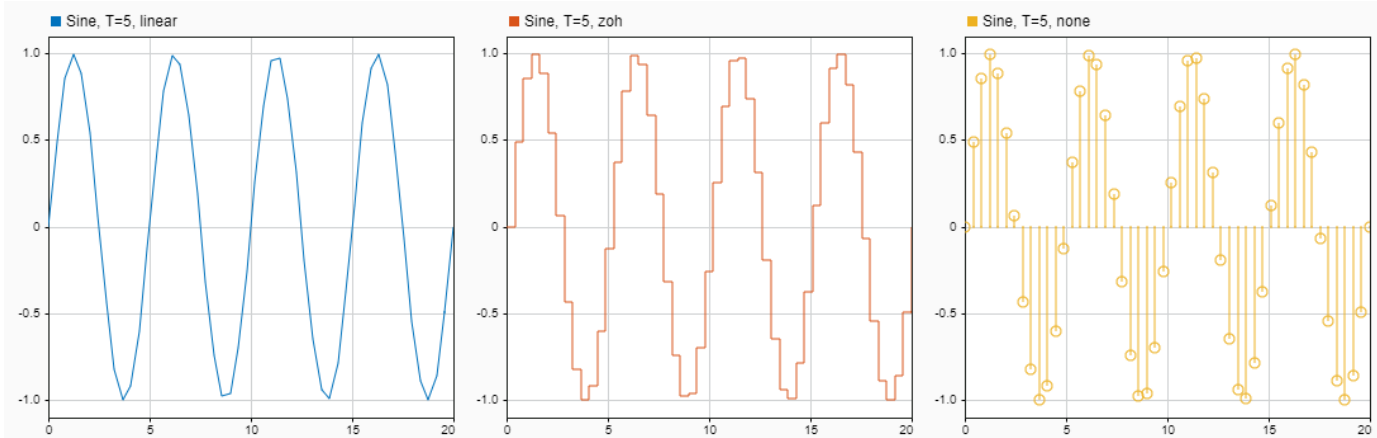
当您指定 **intersection** 同步时，仿真数据检查器在比较中仅使用两个信号中都存在的采样时间。在第二个绘图中，仿真数据检查器识别没有对应采样进行比较的采样，显示为非实心圆。最终图显示用于比较的信号，不包含第二个图中识别出的那些采样。



选择哪个同步选项涉及速度和准确度之间的权衡。**union** 同步所需的插值需要耗费一定的时间，但会提供更准确的结果。当您使用 **intersection** 同步时，比较会很快完成，因为仿真数据检查器会计算较少数据点的差异，而且不会进行插值。但是，使用 **intersection** 同步时，会丢弃一些数据，准确度有所降低。

## 插值

信号的插值属性确定仿真数据检查器如何显示信号以及在同步中如何计算其他数据值。您可以选择使用零阶保持 (zoh) 或线性逼近对数据进行插值。也可以指定不进行插值。



当您为**插值方法**指定 **zoh** 或 **none** 时，仿真数据检查器将对插值的采样时间复制上一个采样的数据。当您指定 **linear** 插值时，仿真数据检查器使用插值点两侧的采样来线性逼近所插的值。通常，离散信号使用 **zoh** 插值，连续信号使用 **linear** 插值。您可以在信号属性中为信号指定**插值方法**。

## 容差设定

仿真数据检查器允许您指定信号容差的范围和值。您可以使用绝对、相对和时间容差值的任意组合来定义容差带，并且可以指定所指定的容差是应用于单个信号还是应用于一次运行中的所有信号。

### 容差范围

在仿真数据检查器中，您可以为数据全局指定容差，也可以为单个信号指定容差。全局容差值会应用于**覆盖全局容差**未设置为 **yes** 的一次运行中的所有信号。您可以在**比较视图**的图形查看区域顶部为数据指定全局容差值。要指定信号特定的容差值，请编辑信号属性，并确保**覆盖全局容差**属性设置为 **yes**。

### 容差计算

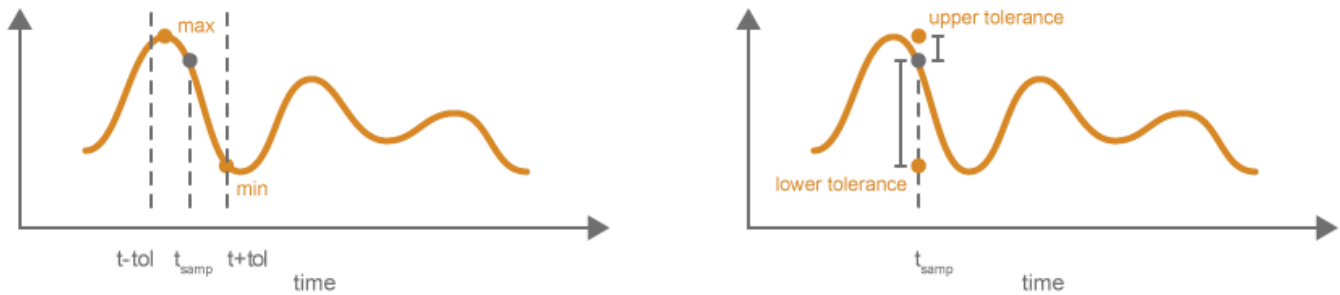
在仿真数据检查器中，您可以使用绝对、相对和时间容差值的组合为运行或信号指定容差带。当您使用多种类型的容差为运行或信号指定容差时，每个容差可能在每个点上产生不同的容差结果。这时仿真数据检查器会通过为每个数据点选择最宽松的容差结果来计算整体容差带。

当仅使用绝对和相对容差属性定义容差时，仿真数据检查器会简单地将每个点的容差计算为最大值。

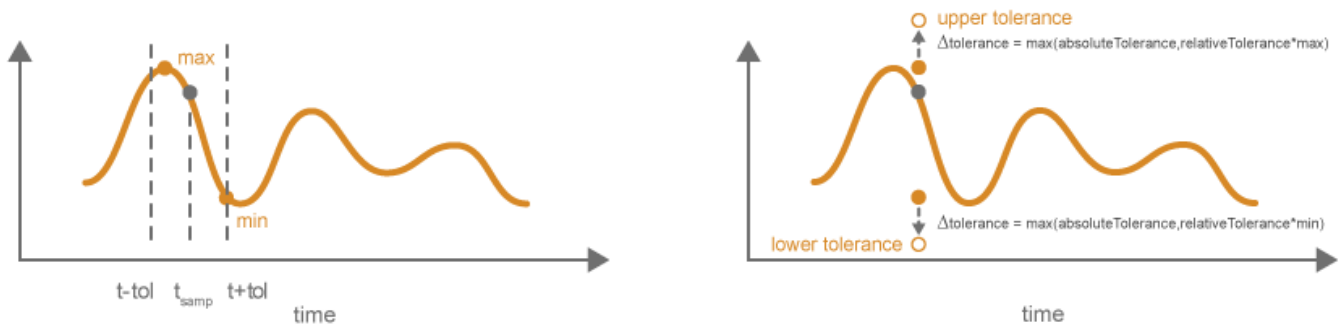
$$\text{tolerance} = \max(\text{absoluteTolerance}, \text{relativeTolerance} * \text{abs}(\text{baselineData}));$$

容差带的上界通过将 **tolerance** 加到**基线**信号上形成。同样，仿真数据检查器通过从**基线**信号中减去 **tolerance** 来计算容差带的下界。

当您指定时间容差时，仿真数据检查器会首先计算每个采样在其时间区间内的时间容差，该时间区间定义为  $[(t_{\text{samp}} - \text{tol}), (t_{\text{samp}} + \text{tol})]$ 。仿真数据检查器通过选择每个采样的时间区间容差内的最小值点来构建容差带的下界。同样，它还会选择每个采样的时间区间容差内的最大值点定义为其容差上界。



如果除了使用时间容差，您还使用绝对或相对容差指定容差带，则仿真数据检查器将首先应用时间容差，然后将绝对和相对容差应用于用时间容差选择的最大值和最小值点。



$$\text{upperTolerance} = \text{max} + \max(\text{absoluteTolerance}, \text{relativeTolerance} * \text{max})$$

$$\text{lowerTolerance} = \text{min} - \max(\text{absoluteTolerance}, \text{relativeTolerance} * \text{min})$$

## 限制

仿真数据检查器不支持比较以下项：

- `int64` 或 `uint64` 数据类型的信号。
- 可变大小信号。

## 另请参阅

## 相关示例

- “Compare Simulation Data” (Simulink)

## 保存和共享仿真数据检查器数据和视图

在仿真数据检查器中检查、分析或比较您的数据后，您可以与其他人共享您的结果。仿真数据检查器根据您的需要提供几个共享和保存您的数据和结果的选项。使用仿真数据检查器，您可以：

- 将您的数据和布局修改保存在仿真数据检查器会话中。
- 在仿真数据检查器视图中共享您的布局修改。
- 共享您在仿真数据检查器中创建的绘图的图像和图窗。
- 创建仿真数据检查器报告。
- 将数据导出到工作区。
- 将数据导出到文件。

### 保存和加载仿真数据检查器会话

如果要在仿真数据检查器中随配置的视图保存或共享数据，请在仿真数据检查器会话中保存数据和设置。您可以将会话保存为 MAT 文件或 MLDATX 文件。默认格式是 MLDATX。保存仿真数据检查器会话时，会话文件包含：

- **检查**窗格中的所有运行、数据和属性，包括哪个运行是当前运行，哪些运行在存档中。
- 在**检查**窗格中绘制所选信号的显示。
- 子图布局、线型和颜色选择。

---

**注意** 比较结果和全局容差不会保存在仿真数据检查器会话中。

---

要保存仿真数据检查器会话，请执行以下操作：

- 1 将鼠标悬停在左侧栏上的保存图标上。然后，点击**另存为**。




- 2 命名文件。
- 3 浏览到您要保存会话的位置，然后点击**保存**。

对于大型数据集，图形查看区域右下角会叠加显示保存操作的进度状态信息，并允许您取消保存操作。

左侧栏上仿真数据检查器预设项菜单的**保存**选项卡允许您配置与 MLDATX 文件保存操作相关的选项。您可以对用于保存操作的内存量设置低至 50MB 的限制。您还可以选择三个**压缩**选项之一：

- 默认选项“无”在保存操作期间不应用压缩。
- “普通”创建最小的文件大小。
- “最快”创建的文件大小比选择“无”时要小，但比“普通”所需的保存时间更短。



要加载仿真数据检查器会话，请点击左侧栏上的打开图标 。然后，浏览以选择要打开的 MLDATX 文件，并点击**打开**。

您也可以双击 MLDATX 文件。MATLAB 和仿真数据检查器将打开（如果它们尚未打开）。

当仿真数据检查器已包含仿真运行时，如果您打开一个会话，该会话中的所有运行都将移至存档中。视图会更新以显示根据会话文件绘制的信号。您可以根据需要在工作区域和存档之间拖动各次仿真运行。


当仿真数据检查器不包含运行时，如果您打开一个会话，仿真数据检查器会将运行放入工作区域中并按照文件中的指定进行存档。

## 共享仿真数据检查器视图


当您希望以相同方式可视化不同数据集时，您可以保存视图。视图可以在不保存数据的情况下保存仿真数据检查器的布局 and 外观特性。具体来说，视图保存：

- 绘图可视化类型、布局、轴范围、链接特性和归一化坐标区
- 绘图中信号的位置，包括存档中绘制的信号
- 在**检查**窗格中显示的信号分组和列
- 信号颜色和线型

要保存视图，请执行以下操作：

- 1 点击“可视化和布局” 。
- 2 在**保存的视图**中，点击**保存当前视图**。
- 3 在对话框中，指定视图的名称，并浏览到要保存 MLDATX 文件的位置。
- 4 点击**保存**。


要加载视图，请执行以下操作：

- 1 点击“可视化和布局” 。
- 2 在**保存的视图**中，点击**打开保存的视图**。
- 3 浏览到您要加载的视图，然后点击**打开**。

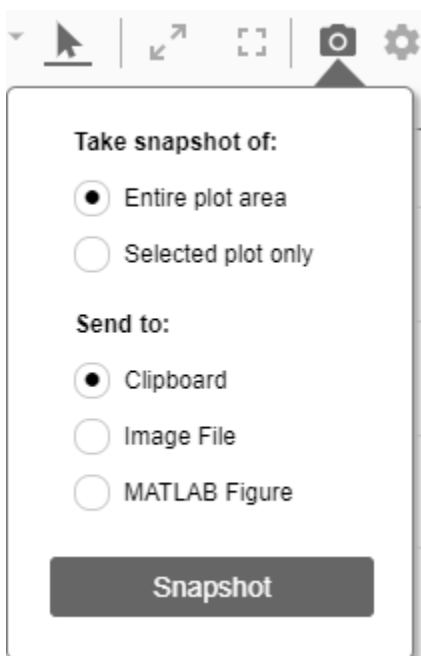
## 共享仿真数据检查器绘图

使用快照功能共享您在仿真数据检查器中生成的绘图。您可以将绘图导出到剪贴板，以作为图像文件粘贴到文档或粘贴到 MATLAB 图窗中。您可以选择捕获整个绘图区域，包括绘图区域中的所有子图，或者仅捕获选定的子图。



点击工具栏上的相机图标  以访问快照菜单。使用单选按钮选择要共享的区域以及共享绘图的方式。作出选择后，点击**快照**导出绘图。





如果您创建一个图像，请在文件浏览器中选择要保存图像的位置。

您可以使用 `Simulink.sdi.snapshot` 以编程方式在仿真数据检查器中创建绘图的快照。

## 创建仿真数据检查器报告

要快速生成结果的文档，请创建仿真数据检查器报告。您可以在**检查**或**比较**窗格中创建数据的报告。该报告是一个 HTML 文件，其中包含活动窗格中所有信号和绘图的有关信息。该报告包括在导航窗格中的信号表中显示的所有信号信息。有关配置该表的详细信息，请参阅“Inspect Metadata” (Simulink)。

要生成仿真数据检查器报告，请执行以下操作：

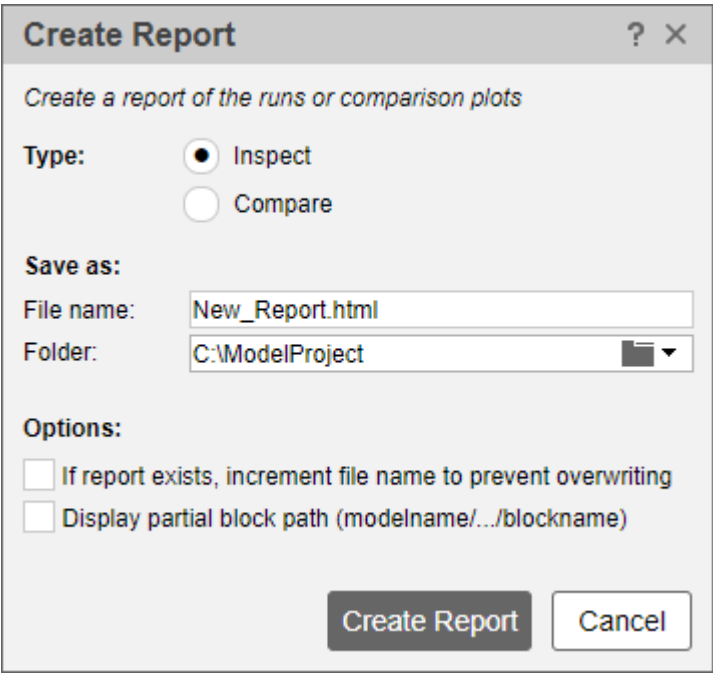
1



点击左侧栏上的创建报告图标。

2 指定要创建的报告的类型。

- 选择**检查**以包含**检查**窗格中的绘图和信号。
- 选择**比较**以包含**比较**窗格中的数据和绘图。生成**比较运行**报告时，您可以选择**仅报告不匹配的信号**或**报告所有信号**。如果选择**仅报告不匹配的信号**，报告仅显示不在指定容差范围内的信号比较。



- 3 为报告指定**文件名**，并导航到您要保存报告的**文件夹**。
- 4 点击**创建报告**。

生成的报告会自动在默认浏览器中打开。

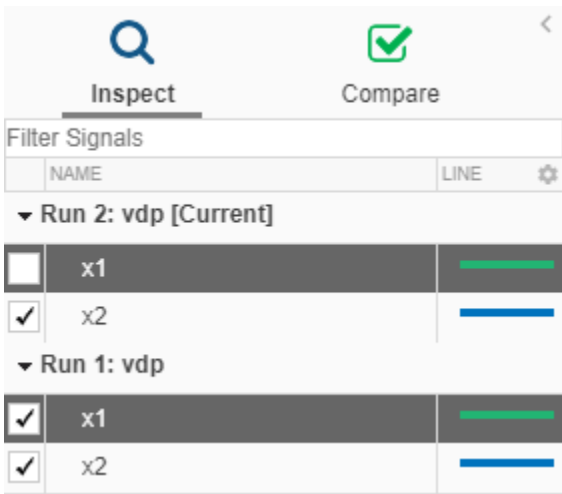
将数据导出到工作区或文件

您可以使用仿真数据检查器将数据导出到基础工作区、MAT 文件或 Microsoft Excel 文件。您可以导出选定的运行和信号、工作区域中的运行、**检查**窗格中的所有运行，包括**存档**。



当您导出选定的运行和信号时，请先选择要导出的数据，再点击导出按钮。

仅导出选定的运行和信号。在本示例中，仅导出 Run 1 和 Run 2 的 x1 信号。绘图数据的复选框选择状态不影响信号是否导出。



当您将单个信号导出到工作区或 MAT 文件时，该信号会导出到 `timeseries` 对象。导出到工作区或 MAT 文件的一次运行或多个信号的数据存储为 `Simulink.SimulationData.Dataset` 对象。

要将数据导出到文件，请在**导出**对话框中选择**文件**选项。您可以指定文件名并浏览到要保存导出的文件的位置。将数据导出为 MAT 文件时，单个导出的信号的数据存储为 `timeseries` 对象，运行或多个信号的数据存储为 `Simulink.SimulationData.Dataset` 对象。在将数据导出为 Microsoft Excel 文件时，数据将使用“Microsoft Excel Import, Export, and Logging Format” (Simulink)中所述的格式存储。

要导出为 Microsoft Excel 文件，请从下拉列表中选择 XLSX 扩展名。在将数据导出为 Microsoft Excel 文件时，可以为导出文件中的数据格式指定其他选项。如果您提供的文件名已存在，您可以选择覆盖整个文件或仅覆盖包含与导出数据对应的工作表。您还可以选择要包括哪些元数据，以及具有相同时间数据的信号是否共享导出文件中的时间列。

## 将视频信号导出到 MP4 文件

您可以使用仿真数据检查器将包含 RGB 或单色视频数据的二维或三维信号导出到 MP4 文件。例如，当您在仿真中记录视频信号时，可以将数据导出到 MP4 文件，并使用视频播放器查看视频。要将视频信号导出到 MP4 文件，请执行下列步骤：

1 选择要导出的信号。

2



点击工具栏中左侧的“导出”，或右键点击信号并选择**导出**。

3 在“导出”对话框中，选择将**所选运行和信号**导出到文件。

4 指定文件名和要保存该文件的位置的路径。

5 从列表中选择“**MP4 视频文件**”，然后点击**导出**。

要使用导出到 MP4 文件的选项，需满足以下条件：

- 一次只能导出一个信号。
- 所选信号必须为二维或三维信号，并包含 RGB 或单色视频数据。
- 所选信号必须在仿真数据检查器中表示为具有多维采样值的单个信号。

在导出信号数据之前，您可能需要转换信号表示。有关详细信息，请参阅“Analyze Multidimensional Signal Data” (Simulink)。

- 信号值的数据类型必须为 **double**、**single** 或 **uint8**。

不支持在 Linux 操作系统上将视频信号导出到 MP4 文件。

### 另请参阅

#### 函数

`Simulink.sdi.saveView`

### 相关示例

- “在仿真数据检查器中查看数据” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)

## 以编程方式检查和比较数据

您可以使用仿真数据检查器 API 从 MATLAB 命令行利用仿真数据检查器的功能。

仿真数据检查器对运行和信号中的数据进行组织，对每个运行和信号赋予一个唯一数值标识。一些仿真数据检查器 API 函数使用运行和信号 ID 来引用数据，而不是接受运行或信号本身作为输入。要访问工作区中的运行 ID，您可以使用 `Simulink.sdi.getAllRunIDs` 或 `Simulink.sdi.getRunIDByIndex`。您可以使用 `getSignalIDByIndex` 方法通过 `Simulink.sdi.Run` 对象访问信号 ID。

`Simulink.sdi.Run` 和 `Simulink.sdi.Signal` 类提供对数据的访问，并允许您查看和修改运行和信号元数据。您可以使用类似 `Simulink.sdi.setSubPlotLayout`、`Simulink.sdi.setRunNamingRule` 和 `Simulink.sdi.setMarkersOn` 的函数来修改仿真数据检查器预设项。要还原仿真数据检查器的默认设置，请使用 `Simulink.sdi.clearPreferences`。

### 创建运行并查看数据

此示例说明如何创建一次运行，向其中添加数据，然后在仿真数据检查器中查看数据。

#### 为运行创建数据

创建 `timeseries` 对象以包含正弦信号和余弦信号的数据。为每个 `timeseries` 对象指定一个描述性名称。

```
time = linspace(0,20,100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';
```

#### 创建运行并添加数据

使用 `Simulink.sdi.view` 函数打开仿真数据检查器。

```
Simulink.sdi.view
```

要从工作区将数据导入仿真数据检查器，请使用 `Simulink.sdi.Run.create` 函数创建一个 `Simulink.sdi.Run` 对象。使用 `Run` 对象的 `Name` 和 `Description` 属性将有关运行的信息添加到其元数据中。

```
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';
```

使用 `add` 函数将您在工作区中创建的数据添加到空运行中。

```
add(sinusoidsRun,'vars',sine_ts,cos_ts);
```

### 在仿真数据检查器中绘制数据

使用 `getSignalByIndex` 函数访问包含信号数据的 `Simulink.sdi.Signal` 对象。您可以使用 `Simulink.sdi.Signal` 对象属性来指定信号的线型和颜色，并将其绘制在仿真数据检查器中。为每个信号指定 `LineColor` 和 `LineDashed` 属性。

```
sine_sig = getSignalByIndex(sinusoidsRun,1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-';
```

```
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.LineDashed = '-';
```

使用 `Simulink.sdi.setSubPlotLayout` 函数在仿真数据检查器绘图区域中配置  $2 \times 1$  子图布局。然后使用 `plotOnSubplot` 函数在顶部子图上绘制正弦信号，在下部子图上绘制余弦信号。

```
Simulink.sdi.setSubPlotLayout(2,1);
```

```
plotOnSubPlot(sine_sig,1,1,true);
plotOnSubPlot(cos_sig,2,1,true);
```

### 关闭仿真数据检查器并保存您的数据

检查完绘制的信号数据后，可以关闭仿真数据检查器并将会话保存到 `MLDATX` 文件中。

```
Simulink.sdi.close('sinusoids.mldatx')
```

## 比较同一运行中的两个信号

您可以使用仿真数据检查器编程接口比较一次运行中的信号。此示例比较飞机纵向控制器的输入和输出信号。

首先，加载包含数据的会话。

```
Simulink.sdi.load('AircraftExample.mldatx');
```

使用 `Simulink.sdi.Run.getLatest` 函数访问最后一次运行的数据。

```
aircraftRun = Simulink.sdi.Run.getLatest;
```

然后，您可以使用 `Simulink.sdi.getSignalsByName` 函数访问表示控制器输入的 `Stick` 信号和表示输出的 `alpha, rad` 信号。

```
stick = getSignalsByName(aircraftRun,'Stick');
alpha = getSignalsByName(aircraftRun,'alpha, rad');
```

在比较信号之前，您可以指定用于比较的容差值。比较使用在比较中为基线信号指定的容差值，因此请在 `Stick` 信号上设置绝对容差值 `0.1`。

```
stick.AbsTol = 0.1;
```

现在，使用 `Simulink.sdi.compareSignals` 函数比较信号。`Stick` 信号是基线，`alpha, rad` 信号是要与基线进行比较的信号。

```
comparisonResults = Simulink.sdi.compareSignals(stick.ID,alpha.ID);
match = comparisonResults.Status
```

```
match =
    ComparisonSignalStatus enumeration

    OutOfTolerance
```

比较结果超出容差范围。您可以使用 `Simulink.sdi.view` 函数打开仿真数据检查器来查看和分析比较结果。

## 使用全局容差比较各运行

您可以指定在比较两个仿真运行时使用的全局容差值。全局容差值应用于运行中的所有信号。此示例说明如何为运行比较指定全局容差值，以及如何分析和保存比较结果。

首先，加载包含要比较的数据的会话文件。该会话文件包含一个飞机纵向控制器的四次仿真的数据。此示例比较使用不同输入滤波器时间常量的两次运行的数据。

```
Simulink.sdi.load('AircraftExample.mldatx');
```

要访问要比较的运行数据，请使用 `Simulink.sdi.getAllRunIDs` (Simulink) 函数获取对应于最后两次仿真运行的运行 ID。

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

使用 `Simulink.sdi.compareRuns` (Simulink) 函数来比较运行。指定全局相对容差值为 **0.2**，全局时间容差值为 **0.5**。

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

检查返回的 `Simulink.sdi.DiffRunResult` 对象的 `Summary` 属性，查看信号是在容差值内还是超出容差。

```
runResult.Summary
```

```
ans = struct with fields:
    OutOfTolerance: 0
    WithinTolerance: 3
    Unaligned: 0
    UnitsMismatch: 0
    Empty: 0
    Canceled: 0
    EmptySynced: 0
    DataTypeMismatch: 0
    TimeMismatch: 0
    StartStopMismatch: 0
    Unsupported: 0
```

所有三个信号比较结果都在指定的全局容差内。

您可以使用 `saveResult` (Simulink) 函数将比较结果保存到一个 MLDATX 文件中。

```
saveResult(runResult,'InputFilterComparison');
```

## 使用信号容差分析仿真数据

您可以通过编程方式指定信号容差值，以便在使用仿真数据检查器进行比较时使用。在此示例中，您将比较通过仿真飞机纵向飞行控制系统模型收集的数据。每次仿真使用不同的输入滤波器时间常量值，并记录输入和输出信号。通过使用仿真数据检查器和信号容差比较结果，可以分析时间常量变化的影响。

首先，加载包含仿真数据的会话文件。

```
Simulink.sdi.load('AircraftExample.mldatx');
```

该会话文件包含四次运行。在此示例中，您将比较文件中前两次运行的数据。访问从文件加载的前两次运行的 `Simulink.sdi.Run` 对象。

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDs1 = runIDs(end-3);
runIDs2 = runIDs(end-2);
```

现在，在不指定任何容差的情况下比较这两次运行。

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDs1,runIDs2);
```

使用 `getResultByIndex` 函数访问 `q` 和 `alpha` 信号的比较结果。

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

检查每个信号结果的 `Status`，查看比较结果是否在我们的容差范围内。

```
qResult.Status
```

```
ans =
    ComparisonSignalStatus enumeration
    OutOfTolerance
```

```
alphaResult.Status
```

```
ans =
    ComparisonSignalStatus enumeration
    OutOfTolerance
```

该比较对所有容差使用值 0，因此 `OutOfTolerance` 结果意味着信号不同。

通过为信号指定容差值，可以进一步分析时间常量的影响。通过设置与所比较信号对应的 `Simulink.sdi.Signal` 对象的属性来指定容差。这些比较使用为基线信号指定的容差。此示例指定时间容差和绝对容差。

要指定容差，首先请访问来自基线运行的 `Signal` 对象。

```
runTs1 = Simulink.sdi.getRun(runIDs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```



使用 `AbsTol` 和 `TimeTol` 属性，为 `q` 信号指定绝对容差 0.1 和时间容差 0.6。

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

为 `alpha` 信号指定绝对容差 0.2 和时间容差 0.8。

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

再次比较结果。访问比较结果，并检查每个信号的 `Status` 属性。

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);
```

```
qResult2.Status
```

```
ans =
    ComparisonSignalStatus enumeration
    WithinTolerance
```

```
alphaResult2.Status
```

```
ans =
    ComparisonSignalStatus enumeration
    WithinTolerance
```

## 另请参阅

### 仿真数据检查器

## 相关示例

- “Compare Simulation Data” (Simulink)
- “仿真数据检查器如何比较数据” (Simulink)
- “Create Plots Using the Simulation Data Inspector” (Simulink)

