对OrcaFlex的Python接口的介绍

大卫-赫弗南
www.orcina.com/

2016年4月

1 概述

OrcaFlex（www.orcina.com/SoftwareProducts/OrcaFlex/），由Orcina有限公司开发。(www.orcina.com/)，是一个主要用于海洋工程的有限元分析工具。OrcaFlex有一个强大的图形用户界面（GUI），使模型开发等任务非常有效。该程序还提供了一些自动使用的方法。这些方法包括从基于Excel的自动化设施到低级别的编程接口，称为OrcFxAPI。

与一些不同的编程语言的接口是可用的。在撰写本报告时，这些语言是C, C++, C#, Delphi, Matlab和Python。本文件对OrcaFlex的Python接口进行了基本介绍。

1.1 OrcFxAPI，OrcaFlex编程接口

编程接口，OrcFxAPI，是作为一个Windows DLL实现的，它通过输出的函数暴露其能力。这些函数被全面地记录下来，但直接调用它们可能是相当可怕的。通常情况下，这需要使用像C或C++这样的编程语言来完成，这些语言的学习曲线相当陡峭。此外，OrcFxAPI DLL所暴露的原始功能使用起来也很繁琐。程序员必须照顾到内存分配和去分配，并编写错误检查代码。这些方面的开发有些令人厌烦，特别是对于非专业的程序员来说，犯编程错误（内存泄漏、错误处理缺失等）是非常常见的。

为了让OrcFxAPI的消费者生活得更轻松，我们提供了来自C#、Matlab和Python的高级接口。这些都是建立在低级别的OrcFxAPI接口之上的，但将程序员从上面讨论的所有繁琐的方面屏蔽掉。内存分配由高层接口以对程序员透明的方式进行。高级接口通过将低级错误代码转换为有意义的异常来处理错误。

所以，一般来说，如果你的需求能被其中一个高级接口满足，那么它应该比直接使用低级接口更有成效。在本文的其余部分，我们只考虑 Python 接口。然而，所讨论的整体概念也同样适用于其他的高层接口，并考虑到语言语法之间的任何差异。

1.2 蟒蛇

Python（www.python.org/）是一种高级的、通用的编程语言。该语言很成熟，可以追溯到1991年，并被广泛用于许多不同的应用领域。Python是开放源码和免费的。有大量的扩展包，可用于执行不

同的任务，如数值分析、信号处理、并行处理、GUI编程、图像处理、数据库编程等。Python通常用于科学编程，而且往往有很好的文档。

开发OrcaFlex的Python接口是我的一个个人项目，动机是我想学习Python。在向我们开发团队的同事介绍了接口的早期版本后，我们很快采用Python作为编写OrcaFlex测试代码的首选语言。现在OrcaFlex自动测试套件完全是用Python编写的，这证明了Python界面的多功能性。

1.3 Python版本

有两种不兼容的 Python 语言，Python 2 和 Python 3。它们之间与本文有关的主要区别是打印语句或函数。在 Python 2 中 `print` 是一个语句。

```
print 'Hello'.
```

在Python 3中，`print`是一个函数。

```
print('Hello')
```

这确实意味着为 Python 2 编写的代码可能无法在 Python 3 上执行，反之亦然。本文以Python 3为目标，因此使用`print`作为函数。如果你试图在Python 2上执行下面的任何代码样本，那么它们很可能会表现得不正确，因此我们建议在执行本文的代码时使用Python 3。

请注意，OrcaFlex本身对Python 2和Python 3都有完全的支持。

1.4 本文件中代码的表述

本文件中的源代码是在一个盒子里呈现的，像这样。

```
print('Some output')
```

有时我们会完整地介绍整个节目，其他时候我们会在节目中穿插解释性的文字。

```
print('Some more output')
```

在整个程序呈现后，输出被显示出来，由输出左边的垂直线表示。

```
print('The end, there is no more')
```

```
一些产出
一些更多的输出
结束了，没有了
```

本文件中介绍的Python源文件可以单独获得。

1.5 第一个方案

在纠结于细节之前，让我们至少写一个简单的程序来演示可以用OrcaFlex的Python接口做什么。我们要做的第一件事是导入OrcFxAPI模块，它提供了OrcaFlex的接口。

```
输入OrcFxAPI
```

我们创建一个模型类的实例，它代表整个OrcaFlex模型。

```
model = OrcFxAPI.Model()
```

在这一点上，加载一个现有的OrcaFlex基础模型是很常见的。但相反，我们展示了模型可以在代码中建立。虽然你可能不希望以这种方式建立大型复杂的模型，但你可以使用这种能力来加载现有的模型并进行修改。在这里，我们创建了一个船只对象和一个线状对象。

```
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
```

我们可以为这些对象设置一些数据。我们将把线的末端A连接到容器上，偏移该连接，同时设置末端B的位置。

```
line.EndAConnection = vessel.Name
line.EndAX = 45.0
line.EndAZ = -5.0
line.EndBX = 110.0
line.EndBZ = -30.0
# 跳过位置的Y值设置，保留默认值为0
```

有使用OrcaFlex批处理脚本语言的自动化经验的OrcaFlex用户会认识到这些名称是生产线的连接和位置的数据名称。

我们现在可以进行计算了。我们用默认的阶段持续时间和环境条件运行一个模拟。

```
model.RunSimulation()
```

最后，让我们提取并输出一些结果--线的最大和最小的顶部张力。

```
张力 = line.TimeHistory('有效张力',
    objectExtra=OrcFxAPI.oeEndA)
print('最大张力=', max(tension))
```

```
最大张力=91.157585144 最小张力
=9.47248744965
```

2 OrcaFlex对象模型

OrcaFlex的对象和它们的数据有一个结构和层次，可以映射到Python接口上。因此，为了使用Python接口，理解这个结构是很重要的。为了帮助说明，图1显示了OrcaFlex模型浏览器和由第1.5节的代码创建的模型。

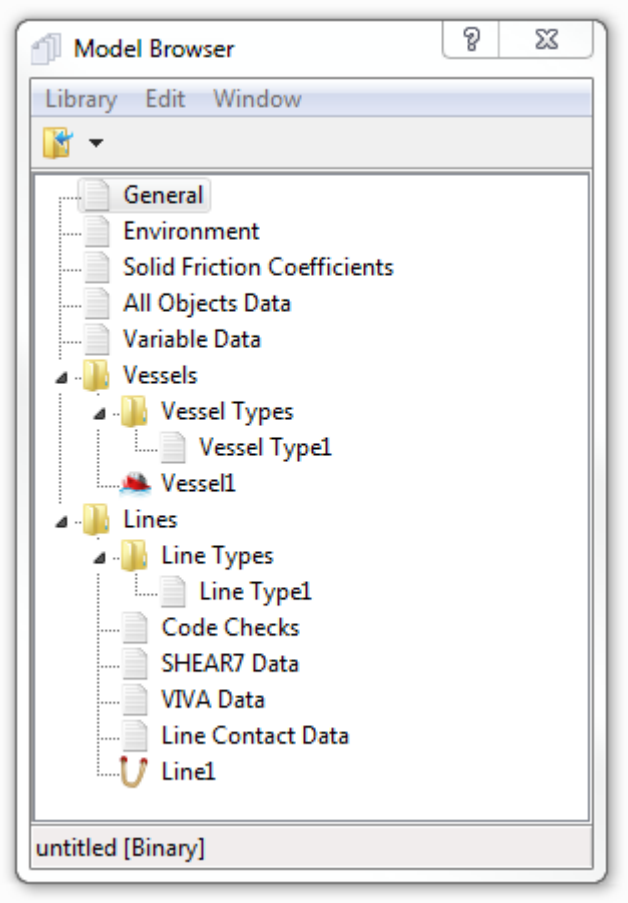


图1：OrcaFlex模型浏览器显示对象的层次结构

2.1 枚举对象

我们现在可以重新创建该对象结构，并使用模型的对象属性将其输出。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
for obj in model.objects:
    print(obj)
```

```
<一般数据: '一般'>
<环境数据: 'Environment'>。
<固体摩擦系数:'固体摩擦系数'>
<线路联系数据: '线路联系数据'>。
```

<代码检查：'代码检查'>
<SHEAR7数据：'SHEAR7数据'>。
<VIVA数据：'VIVA数据'>。

```
<容器: 'Vessel1'>
<线: '线1'>
<线型。'线型1'>
<船舶类型。'Vessel Type1'>
```

传递给CreateObject的参数指定了被创建的对象的*类型*。这里我们创建了一个容器和一条线。请注意，OrcaFlex也会自动创建新创建的容器和线条对象所需要的容器类型和线条类型对象。

模型浏览器所显示的层次结构并没有被Python模型所反映，它呈现的是一个扁平的Python对象的元组。这些对象可以被认为是分为两个主要类别。

- 自动创建的对象，始终存在，并且有适用于整个模型的数据。一般、环境、实体摩擦系数、线接触、代码检查、SHEAR7和VIVA对象都属于这个类别。这些对象不能被销毁，也不能被重命名。
- 由用户创建的对象，如船和线对象。模型可以包含任何数量的此类对象，它们既可以被创建也可以被销毁。它们的名字是由用户选择的。

2.2 销毁物体

我们已经看到了如何使用CreateObject方法来创建对象。它们是通过调用DestroyObject来销毁的。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)

# 通过向DestroyObject方法传递引用来销毁对象 ...
model.DestroyObject(line)

# ......如果我们没有可用的引用，则通过传递对象名称，例如自动创建的线型
model.DestroyObject('Line Type1')
```

2.3 用名称来指代对象

对象也可以通过名称来引用。这在操作已经在GUI中建立并随后由Python接口加载的模型时被广泛使用。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)

# 设置该行的名称和一些章节数据
line.Name = 'Riser'
line.Length = 87.0, 45.0
line.TargetSegmentLength = 5.0, 2.0

# 硬编码的名称
print('Riser length =', model['Riser'].Length)
```

```
# 从部分数据中读取线型名称 lineTypeName =
line.LineType[0] lineType =
model[lineTypeName]
print(lineTypeName, 'EA =', lineType.EA)
```

```
立管长度 = (87.0, 45.0)
立管段长度= (5.0, 2.0)  线路类型1
EA=700000.0
```

可以直接访问两个最常用的自动创建的对象，而不必通过名称来查找它们。

```
输入OrcFxAPI

model = OrcFxAPI.Model()

# 我们可以通过名字访问一般数据和环境数据
print(model['General'].ImplicitConstantTimeStep)
print(model['Environment'].WaveHeight)

# 但 这 样 做 更 方 便
print(model.general.ImplicitConstantTimeStep)
```

```
0.1
7.0
0.1
7.0
```

2.4 选择一个特定类型的对象

想对某一特定类型的所有对象进行操作是很常见的，例如所有的线、所有的船等等。对象的类型属性允许我们确定特定对象的OrcaFlex类型。这个属性是一个整数值，用于识别对象的类型。这个类型属性应该与预定义的值进行比较，如otVessel、otLine、otLineType等。这些是我们在创建对象时使用的相同的预定义值。

```
输入OrcFxAPI

model                    =
OrcFxAPI.Model()  # 创 建
两 个 容 器 ...  for  i  in
range(2):
    model.CreateObject(OrcFxAPI.otVessel)。
# ......和三行
for i in range(3):
    model.CreateObject(OrcFxAPI.otLine)

# 循环，对所有行进行操作
for obj in model.objects:
    如果obj.type == OrcFxAPI.otLine:
        print('(1)', obj.name)
```

```
    如果obj.type == OrcFxAPI.otVessel:
        vessels.append(obj)
print('(2)', vessels)

# 列表理解，更简洁地做同样的事情
vessels = [obj for obj in model.objects if obj.type
    == OrcFxAPI.otVessel]
print('(3)', vessels)

# 类型属性返回一个整数值
print('(4)', vessels[0].type, OrcFxAPI.otVessel)

# 不要将类型属性与内置的type()函数相混淆
```

```
(1) 线路1
(1) 2号线
(1) 3号线
(2) [<容器：'Vessel1'>, <容器：'Vessel2'>]
(3) [<容器：'Vessel1'>, <容器：'Vessel2'>]
(4) 5 5
(5) <类 'OrcFxAPI.OrcaFlexVesselObject'>。
```

3 OrcaFlex数据模型

我们已经看到了一些访问OrcaFlex输入数据的简单例子。现在我们将更深入地研究这个话题。

如前所述，数据是由名称识别的--与OrcaFlex批处理脚本使用的名称相同。这些数据名称没有列在实现接口的OrcFxAPI Python模块中，而是只在OrcFxAPI DLL内部持有。因此，数据访问是一个*动态*过程。你要求为一个给定的名称提供数据，指定为一个字符串，OrcFxAPI在其内部的数据项名称列表中寻找这个名称。

3.1 非索引的数据访问

数据可以通过GetData方法读取，并通过SetData方法修改。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
print(model.environment.GetData('WaveHeight', -1))
model.environment.SetData('WaveHeight', -1, 2.5)。
print(model.environment.GetData('WaveHeight', -1))
```

```
7.0
2.5
```

上面使用的-1指定了索引。这对于构成OrcaFlex中表格一部分的数据项是需要的。但是我们在这里访问的数据项，即波浪高度，并没有出现在一个表中。请注意，在图2中，数据出现在一个有单一可编辑行的控件中。这就是我们所说的

*非索引的*数据项。还值得指出的是，用于识别数据的名称出现在图2的弹出提示中。这些数据名称也可以通过在数据表格上点击右键，选择*数据名称*菜单项，或按F7键来检查。

Wave Data:

Direction (deg)	Height (m)	Period (s)	Wave Origin X (m)	Wave Origin Y (m)	Wave Time Origin (s)	Wave Type
180.00	7.00	8.00	0.00	0.00	0.000	Stokes' 5th

Allowable values: Cannot be negative
Data name: WaveHeight

图2：波高数据项，非索引数据的一个例子

GetData和SetData方法仍然需要传递一个索引，对于索引的和非索引数据项。对于非索引的数据项，索引被忽略，任何东西都可以被传递。按照惯例，我们用-1来向代码的读者表示这个数据项是没有索引的。如果由于某种原因，我们犯了一个错误，传递了一个有索引的数据项的名称，那么-1不可能是一个有效的索引，所以会产生一个错误。因此，使用-1的好处是可以保护我们免受一种潜在的编程错误的影响。

你会注意到，这种形式的数据访问比我们到目前为止所看到的更加冗长。我们可以用更简洁的语法来编写前面的程序。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
print(model.environment.WaveHeight)
model.environment.WaveHeight = 2.5
print(model.environment.WaveHeight)
```

```
7.0
2.5
```

这种语法利用了 Python 语言的*动态*特性。通常情况下，environment.WaveHeight表示WaveHeight是环境对象的一个定义属性。然而，有可能覆盖Python对象的属性查找，这就是这种简洁的语法的实现方式。当一个属性没有被识别时，该对象会询问OrcFxAPI DLL是否识别该属性的名字作为一个数据项。如果名称被识别，那么代码将被转化为对GetData或SetData的调用。

总的来说，这第二种变体比直接调用GetData或SetData更受欢迎。这条规则的主要例外是当数据名称只有在运行时才知道，而不是在编写代码时才知道。这种情况相当罕见，可能最常发生在我们内部的OrcaFlex测试代码中。

3.2 指数化数据访问

索引数据项是指那些在表格中呈现的、有多于一行可编辑的数据。我们将使用阶段持续时间数据（见图3）进行说明。

索引数据可以以预期的方式使用GetData或SetData进行访问，通过提供一个索引而不是我们用于非索引数据的-1的假值。使用数据名作为属性的简明语法通过Python索引操作符[]提供了索引。该接口还允许读取或修改行数，插入或删除行，以及读取或修改整个数据列。

索引总是基于零，这意味着第一个项目的索引是0。这个选择是为了与Python使用的惯例相匹配。选择基于零或基于一的索引提供了

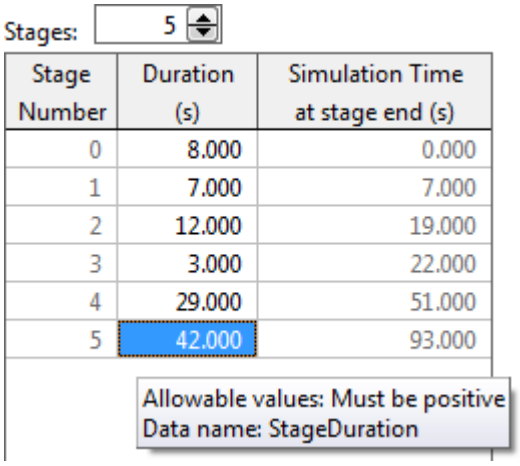


图3：阶段持续时间数据项，是索引数据的一个例子

有可能造成混乱。并非所有的OrcaFlex自动化方法都做出同样的选择。我们选择了在个案的基础上进行选择。因此，Python、C#和高级C++接口都使用
基于零的索引。另一方面，OrcaFlex的批处理脚本和文本数据文件使用基于一的索引，Matlab和低级别的C++接口也是如此。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
general = model.general

# 两种获得计数的方法，我们倾向于第一种方案
print('(1) Stage count =', len(general.StageDuration))
print('(2) Stage count =', general.GetDataRowCount('StageDuration'))

# 修改计数
general.SetDataRowCount('StageDuration', 4)。
print('(3) Stage count =', len(general.StageDuration))

# 获取和设置整列数据
print('(4) Stage duration =', general.StageDuration)
general.StageDuration = 8, 7, 12, 3, 29, 42 # 设置计数和数值
print('(5) Stage duration =', general.StageDuration)

# 按索引获取项目，注意Python对负数索引的约定 print('(6)积累时间=',
general.StageDuration[0]) print('(7)最后阶段时间=',
general.StageDuration[-1])

# 删除和插入行
general.StageDuration.DeleteRow(1)
general.StageDuration.InsertRow(3)
general.StageDuration[3] = 25
print('(8) Stage duration =', general.StageDuration)
```

```
# 遍历这些值
for index, value in enumerate(general.StageDuration):
    print('(9) Stage', index, 'duration =', value)
```

(1) 舞台 数 = 2
(2) 舞台 数 = 2
(3) 舞台 数 = 4
(4) 阶段持续时间 = (8.0, 16.0, 16.0, 16.0)
(5) 阶段持续时间 = (8.0, 7.0, 12.0, 3.0, 29.0, 42.0)

```
(6) 积累时间=8.0
                                .0
(7) 最后阶段持续时间=42        .0, 3.0, 25.0, 29.0, 42.0)
(8) 阶段持续时间= (8.0, 12
(9)  舞台  0  持续时间  = 8.0
(9)  舞台  1  持续时间  = 12.0
(9)  舞台  2  持续时间  = 3.0
(9)  舞台  3  持续时间  = 25.0
(9)  舞台  4  持续时间  = 29.0
```

值得详细说明的是对负指数的解释，如StageDuration[-1]中看到的。负指数被解释为从数组的末端开始计算。所以索引-1是最后一项，-2是倒数第二项，以此类推。请注意，当你使用索引操作符[]时，这个约定适用，但在调用GetData和SetData时则不适用。

作为一项规则，在OrcaFlex中以具有多个可编辑行的控件呈现的数据是索引数据。然而，有几个明显的例外。线型和团块型数据以两种模式呈现。这些数据可以在*全部*模式或*单独*模式下查看。当以*全部*模式查看时，数据显示在一个表格中，每个线条类型或丛生类型有一行。因此，虽然这些数据可能看起来是索引数据，但它们实际上是非索引数据。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
lineType1                                        =
model.CreateObject(OrcFxAPI.otLineType)   lineType2
=           model.CreateObject(OrcFxAPI.otLineType)
print(lineType1.Name)
```

```
线路类型1
线路类型2
```

3.3 选择

有些输入数据需要另一种技术来访问它们--选择。我们将用波浪列车数据来证明这一点（图4），但这些概念适用于其他需要选择的数据形式。

波浪列车组框中的数据决定有多少个波浪列车，以及它们的名称。与波浪有关的其他数据适用于*所选的*波浪列车。在图中，有三个波浪列车，有点匪夷所思地命名为*Wave1*、*Wave2*和*Wave3*。突出显示的波浪列车，即*Wave2*，是选定的波浪列车，任何与波浪列车有关的数据访问都是指该选定波浪列车的数据。

为了以编程方式访问波浪列车的具体数据，我们必须首先选择一个波浪列车。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
environment = model.environment

# 定义两个波段，名称要有信息量
environment.WaveName = 'Swell', 'Wind generated'.

# 选择漩涡波列，并设置其数据 environment.SelectedWave =
'Swell' environment.WaveType = 'Single Airy'
environment.WaveDirection = 140.0
```

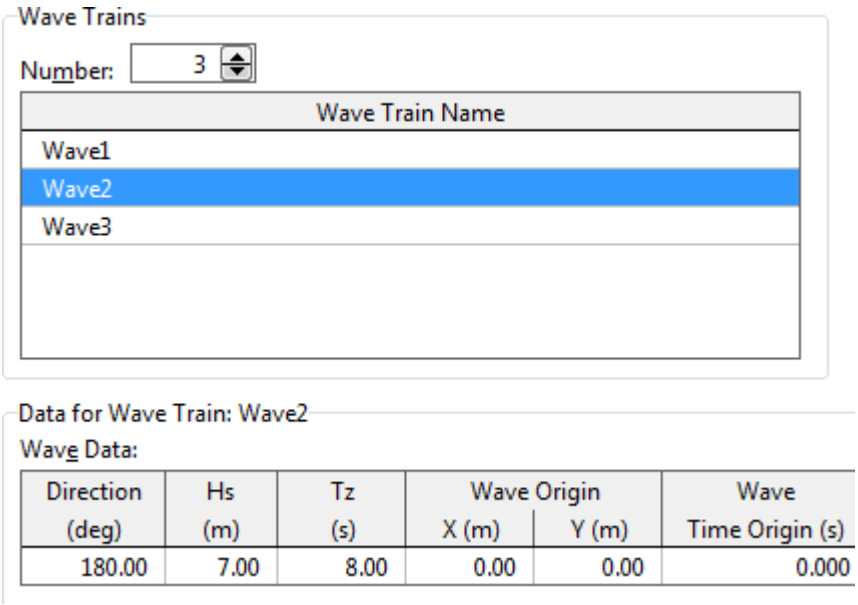


图4：波浪列车数据

```
环境.WaveHeight = 4.5
环境.WavePeriod = 17.0

# 选择风产生的波浪列车，并设置其数据 environment.SelectedWave
= 'Wind generated' environment.WaveType = 'JONSWAP'
environment.WaveDirection = 75.0
环境.WaveHs = 3.0
```

同样，熟悉批处理过程的OrcaFlex用户应该熟悉选择的概念。在OrcaFlex批处理脚本中，选择波段使用选择命令。

```
...
选择环境
  选择波浪'风产生' 波浪类型=JONSWAP 波
    浪方向=75.0
...
```

数据名称SelectedWave不容易从OrcaFlex发现。在程序中没有可见的数据项有这个名称。那么，你应该如何发现这个名字，以及其他选择数据的名字？批处理脚本的文档（www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/Help/Redirector.htm?BatchProcessing,Examplesofsettingdata.htm）描述了选择批脚本命令的各种形式，用于需要选择的数据。比如说。

```
...
选择环境 选择电流 电流1
    //为当前的这个项目设置一些数据
...
选择 "SHEAR7数据"。
  选择SHEAR7SNCurve 曲线2
```

```
...
选择 "线路联系数据"。
  选择穿透器位置数据集Locations1
    //为这个穿透器的位置设置一些数据，数据集
```

这些选择命令的形式都是选择数据类型名称。批量脚本解释器将这些命令转化为 SelectedDataType = Name形式的赋值。这意味着存在一个形式为 SelectedDataType 的数据名。一旦你理解了这个规则，你就可以从你的 Python 代码中使用它。

例如，对于当前数据集，上面的批处理脚本选择命令是Select Current Current1。所以这意味着DataType是Current，所以相应的选择数据名称是SelectedCurrent。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
environment = model.environment
environment.MultipleCurrentDataCanBeDefined = 'Yes'。

# 定义两个当前的数据集，其名称相当无用
environment.CurrentName = 'Current1', 'Current2' 。

# 选择第一个电流数据集，并设置速度 environment.SelectedCurrent =
'Current1' environment.RefCurrentSpeed = 0.3

# 选择第二个电流数据集，并设置速度 environment.SelectedCurrent =
'Current2' environment.RefCurrentSpeed = 0.5
```

3.4 特殊值、默认值和无限值

OrcaFlex中的一些数据项可以接受我们所说的*特殊值*。例如，线端连接刚度可以有*无限的*值。使用 OrcinaInfinity() 来获得代表这个特殊值的浮点值。特殊值的完整列表是。

- OrcinaDefaultReal() 的浮点值显示为'~'。
- 显示为'~'的整数值的OrcinaDefaultWord。
- OrcinaInfinity() 为显示为'*Infinity*'的浮点值。
- OrcinaUndefinedReal() 的浮点值显示为'?
- OrcinaNullReal() 的浮点值显示为'*N/A*'。

4 保存和加载文件

虽然有可能建立完整的模型，进行分析并提取结果，但更常见的是加载现有的输入文件，保存修改后的输入文件，加载现有的仿真文件，等等。例如，人们可能使用OrcaFlex的批处理功能或分布式OrcaFlex来生成仿真文件，但随后使用Python代码来对这些仿真文件进行后处理。

4.1 负载

要加载OrcaFlex文件，请使用LoadData和LoadSimulation方法。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
model.LoadData('model.dat') # 二进制数据文件
model.LoadData('model.yml') # 文本数据文件，YAML
model.LoadData('model.sim') # 从一个模拟文件中加载数据
```

因为创建一个模型并立即加载一个文件是很常见的，所以模型的构造函数
类接受一个文件名参数。

```
输入OrcFxAPI

model = OrcFxAPI.Model('model.dat') # 二进制数据文件 model =
OrcFxAPI.Model('model.yml') # 文本数据文件，YAML model =
```

4.2 节约

要保存文件，请使用SaveData和SaveSimulation方法。

```
输入OrcFxAPI

model = OrcFxAPI.Model('model.sim')
model.SaveData('model.dat') # 二进制数据文件
model.SaveData('model.yml') # 文本数据文件，YAML
```

保存数据时，文件扩展名决定了是保存二进制数据文件还是文本数据文件。

5 进行分析

OrcaFlex提供的主要分析方法是静态分析和动态分析。虽然OrcFxAPI确实支持其他形式的分析，如模态分析、疲劳分析等，但我们在这里只涉及静态和动态分析。更多的细节可以在文档中找到
。

静态分析是由CalculateStatics方法调用的。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)

model.CalculateStatics()
print('(1) Model state =', model.state)
```

对于动态分析，调用RunSimulation。

```
model.RunSimulation()
print('(2) Model state =', model.state)
```

调用重置方法，使模型回到重置状态。

```
model.Reset()
print('(3) Model state =', model.state)
```

请注意，你可以从重置状态调用RunSimulation，而不需要明确地执行静态计算。静态计算首先由OrcFxAPI隐含地执行，然后再执行动态计算。

```
model.RunSimulation()
print('(4) Model state =', model.state)
```

注意，修改数据也将导致模型被重置。

```
line.Length = 120.0,
print('(5) Model state =', model.state)
```

```
(1) 模型状态=InStaticState
(2) 模型状态 = 仿真停止
(3) 模型状态=重置
(4) 模型状态 = 仿真停止
(5) 模型状态=重置
```

这些进行分析的功能对于复杂的模型来说可能需要相当长的时间来完成。正因为如此，这些函数提供了报告进度和取消的设施。为了保持论述的相对简单，我们在这里省略了对这些细节的讨论。

6 提取结果

现在我们知道了如何加载、建立和修改模型，以及如何进行分析，剩下的就是学习如何查询结果了。OrcaFlex中的所有结果都可以从OrcFxAPI中获得。

我们将首先看一下时间历史的结果。这些结果是通过调用TimeHistory方法，通过名称指定所需的结果变量。该方法有如下签名。

```
TimeHistory(varNames, period=None, objectExtra=None)
```

6.1 指定模拟期

我们将首先看一下指定模拟周期的period参数。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
```

```
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

# 没有指定周期，默认为整个模拟。
X = vessel.TimeHistory('X')
print('(1) 没有指定周期', min(X), max(X))

# 指定一个时期，整个模拟，相当于之前的调用
X = vessel.TimeHistory('X', OrcFxAPI.pnWholeSimulation)
print('(2)整个模拟', min(X), max(X)

# 积累，阶段0
X = vessel.TimeHistory('X', 0).
print('(3) build-up', min(X), max(X))

# 第一阶段
X = vessel.TimeHistory('X', 1).
print('(4) stage 1', min(X), max(X))

# 最新的浪潮
X = vessel.TimeHistory('X', OrcFxAPI.pnLatestWave)
print('(5) latest wave', min(X), max(X))

# 由时间指定的周期
```

```
(1) 未指定期限 -0.220513388515 0.235916048288
(2) 整个模拟 -0.220513388515 0.235916048288
(3) 积累 -0.121314510703 0.190882235765
(4) 第一阶段 -0.220513388515 0.235916048288
(5) 最新波 -0.220513388515 0.235916048288
(6) 1.0s到2.0s -0.220513388515 -0.182141140103
(7) 1.0s到2.0s [-0.21700118 -0.21944398 -0.22051339 -0.22021785
-0.21857354 -0.21560377
 -0.21133871 -0.20581487 -0.1990746 -0.19116575 -0.18214114]
```

TimeHistory方法返回一个元组，或者如果numpy（www.numpy.org/）模块可用，则返回一个numpy数组对象。

6.2 抽查时间

要求与时间历史结果相对应的样本时间是很常见的。我们可以为一般对象的*时间*变量请求一个时间历史。但是，使用模型对象的SampleTimes方法更符合习惯。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

period = OrcFxAPI.SpecifiedPeriod(1.0, 2.0)
times = model.SampleTimes(period)
X = vessel.TimeHistory('X', period)
for t, x in zip(times, X):
    print('t={:4.1f}s, X={:7.4f}m'.format(t, x))
```

```
t=1.0s,  X=-0.2170m
t=1.1s,  X=-0.2194m
t=1.2s,  X=-0.2205m
t=1.3s,  X=-0.2202m
t=1.4s,  X=-0.2186m
t=1.5s,  X=-0.2156m
t=    1.6s,    X=-
0.2113m  t=  1.7s,
X=-0.2058m       t=
1.8s,    X=-0.1991m
t=    1.9s,    X=-
0.1912m  t=  2.0s,
X=-0.1821m
```

为了方便起见，对象也提供了一个SampleTimes方法，所以在上面的程序中，下面两行是等价的。

```
times = model.SampleTimes(period)
times = vessel.SampleTimes(period)
```

6.3 提取多个变量的结果

TimeHistory方法允许传递多个变量名，并以列堆叠的形式返回时间历史。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

period = OrcFxAPI.SpecifiedPeriod(1.0, 2.0)
varNames = 'X', 'Y', 'Z'
pos = vessel.TimeHistory(varNames, period)
print(pos)
```

```
[[-0.21700118  0.         -0.14714514]
 [-0.21944398  0.         -0.19857015]
 [-0.22051339  0.         -0.24879079]
 [-0.22021785  0.         -0.29748428]
 [-0.21857354  0.         -0.34433767]
 [-0.21560377  0.         -0.38905022]
 [-0.21133871  0.         -0.43133539]
 [-0.20581487  0.         -0.4709231 ]
 [-0.1990746   0.         -0.50756156]
 [-0.19116575  0.         -0.54101902]
 [-0.18214114  0.         -0.57108533]]
```

这个数组的第一行包含第一个采样时间的容器的X、Y和Z坐标。第二行包含第二个采样时间的坐标，以此类推。

这个例子可以扩展到展示使用numpy与OrcFxAPI可以实现的简洁的代码。我们将添加第二个船只，并展示如何计算两个船只之间的距离。

```
import numpy
import OrcFxAPI
```

```
model = OrcFxAPI.Model()
model.environment.WaveType = 'JONSWAP' # 使用不规则波浪
model.environment.WaveDirection = 130.0
vessel1 = model.CreateObject(OrcFxAPI.otVessel)
vessel1.InitialX, vessel1.InitialY = 0.0, 0.0
vessel2 = model.CreateObject(OrcFxAPI.otVessel)
vessel2.InitialX, vessel2.InitialY = 80.0, -25.0
model.RunSimulation()

period = OrcFxAPI.SpecifiedPeriod(1.0, 2.0)
varNames = 'X', 'Y', 'Z'
pos1 = vessel1.TimeHistory(varNames, period)
pos2 = vessel2.TimeHistory(varNames, period)
distance = numpy.linalg.norm(pos1-pos2, axis=1)
times = model.SampleTimes(period)
for t, d in zip(times, distance):
    print('t={:4.1f}s, d={:7.3f}m'.format(t, d) )
```

```
t=  1.0s,  d=
82.132m  t=  1.1s,
d=  82.074m  t=
1.2s,  d=  82.021m
t=  1.3s,  d=
81.973m  t=  1.4s,
d=  81.930m  t=
1.5s,  d=  81.892m
t=  1.6s,  d=
81.861m  t=  1.7s,
d=  81.835m  t=
1.8s,  d=  81.816m
t=  1.9s,  d=
81.804m  t=  2.0s,
d= 81.798m
```

6.4 额外的对象

到目前为止，我们已经考虑了相对简单的结果。但许多结果需要更多的信息来指定。这种附加信息的最常见形式如下。

- 对于许多环境结果，你必须指定一个空间位置，在这个位置上报告结果。
- 对于直线结果，你必须指定报告结果的直线上的点。这可以是*A端*、*B端*、一个弧长或一个节点编号。对于某些结果，你可能还需要指定径向和周向位置。对于间隙结果，你可以指定另一条线的名称，间隙将从该另一条线上报告。
- 对于6D浮标和船只，平移位置、速度和加速度的结果是在物体的特定位置报告的。

这些额外的信息在`TimeHistory`方法的`objectExtra`参数中提供。这个参数接受`ObjectExtra`类的一个实例。虽然你可以直接实例化和填充这样一个实例，但我们提供了辅助函数来简化这一任务。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
environment = model.environment
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
line.EndAConnection = vessel.Name
```

```
line.EndAX, line.EndAY, line.EndAZ = 40.0, 0.0, 5.0
line.EndBX, line.EndBY, line.EndBZ = 70.0, 0.0, -25.0
model.RunSimulation()

期间 = OrcFxAPI.pnLatestWave

#在5号位置的海平面高度，0，-10
tmp = environment.TimeHistory('Elevation', period,
    OrcFxAPI.oeEnvironment(5.0, 0.0, -10.0) )

# 容器在本地容器轴线上的指定点的速度
tmp = vessel.TimeHistory('Velocity',
    period, OrcFxAPI.oeVessel(-10.0, 0.0,
    3.0) )

#终点A和终点B的线上结果 ...
tmp = line.TimeHistory('Effective Tension', period,
OrcFxAPI.oeEndA) tmp = line.TimeHistory('Effective Tension',
period, OrcFxAPI.oeEndB)

# ... 在指定的弧长或节点数上
tmp = line.TimeHistory('Effective Tension', period,
    OrcFxAPI.oeArcLength(15.0) )
tmp = line.TimeHistory('X', period, OrcFxAPI.oeNodeNum(3))

# 如果你希望省略周期并默认为整个模拟，你# 可以使用位置参数，为周期传递None
...
tmp = line.TimeHistory('Effective Tension', None, OrcFxAPI.OeEndA)
# ... 或者你可以使用命名参数
tmp = line.TimeHistory('Effective Tension',
    objectExtra=OrcFxAPI.oeEndA)
```

我们只涵盖了这些辅助函数中的少数。在撰写本文时，完整的清单是：。

- oeEnvironment
- oeBuoy
- oeWing
- oeVessel
- oeConstraint（从10.1版开始）。
- oeSupport
- oeWinch
- 烯线
- oeNodeNum
- oeArcLength
- 终端A
- oeEndB
- oeTouchdown

更多的细节可以在文档中找到。

6.5 静止状态的结果

静态的结果是通过`StaticResult`方法获得的。该方法不需要指定一个周期，原因希望是显而易见的。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

# 当模型处于静态状态时，静态结果可用 ...
print('(1)', line.StaticResult('有效张力', OrcFxAPI.oeEndA))

# ......和动态模拟之后
```

```
(1) 40.3575448081
(2) 40.3575439453
```

请注意，数值之间的微小差异是因为在进行了动力学运算之后，静态状态的结果是由第一个日志采样计算出来的。默认情况下，日志是以单精度采样的。当处于静态状态时，结果是以双精度计算的。

就像时间历史结果一样，如果结果变量不需要对象的额外参数，可以省略。同样，和时间历史结果一样，多个变量的静态状态结果可以通过传递多个变量名来获得。

作为旁证，我们注意到，通过传递pnStaticState作为周期，也可以从TimeHistory方法中获得静态状态结果。注意，StaticResult返回一个标量值，而TimeHistory返回一个数组。当TimeHistory传入周期pnStaticState时，该数组的长度为1，因此下面两行代码是等价的。

```
X = obj.StaticResult(varNames, objectExtra)
Y = obj.TimeHistory(varNames, OrcFxAPI.pnStaticState, objectExtra)[0] 。
```

6.6 范围图

OrcaFlex范围图显示了沿直线的弧长*范围的*结果。这意味着它们只适用于直线对象。这些结果由RangeGraph方法返回，其签名如下。

```
RangeGraph(varName, period=None, objectExtra=None,
    arclengthRange=None, stormDurationHours=None)
```

6.6.1 动态范围图

第一个例子产生一个张力的动态范围图，在最近的波段上。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
line.EndAConnection = vessel.Name
line.EndAX, line.EndAY, line.EndAZ = 40.0, 0.0, 5.0
line.EndBX, line.EndBY, line.EndBZ = 70.0, 0.0, -25.0
```

```
model.RunSimulation()

期间 = OrcFxAPI.pnLatestWave
rg = line.RangeGraph('有效张力', period)
print('{:>5}{:>7}{:>7}{:>7}'.format('z', 'min', 'max', 'mean'))
for z, minTe, maxTe, meanTe in zip(rg.X, rg.Min, rg.Max, rg.Mean) 。
    print('{:5.1f} {:7.3f} {:7.3f} {:7.3f}'.format(
```

```
    z       闽      最大    意味着
  0.0  27.014  84.328  56.673
  5.0  20.501  76.252  48.963
 15.0  14.065  64.224  39.931
 25.0  10.734  51.943  31.981
 35.0   7.636  39.708  24.040
 45.0   4.635  27.689  16.171
 55.0   1.469  16.565   8.618
 65.0  -0.962   9.419   3.733
 75.0   4.609  14.163   8.944
 85.0  12.204  21.799  16.618
 95.0  19.821  29.814  24.466
100.0  23.570  33.900  28.401
```

这里我们指定了周期，但如果省略了它，则选择默认周期。当动态结果可用时，就是整个模拟，如果只有静态结果可用，就是静态状态。期限的规定与时间历史结果的规定完全相同。

RangeGraph方法返回一个包含多个数组的对象，代表OrcaFlex范围图上的不同曲线。上面的代码产生的输出模仿了从OrcaFlex GUI的图形值中获得的输出。

除了上面列出的四个属性外，范围图对象还包含StdDev、上限和下限曲线。请注意，后两者并非对所有结果变量都可用。如果不可用，这些属性将包含无。这些上限和下限曲线，如果可用的话，包含变量的限制。例如，如果指定了一个允许的张力，那么它将被报告为张力范围图的上限曲线。同样地，压缩极限被报告为张力范围图的下曲线。

包含弧长值的x曲线，其名称有点令人困惑。最初选择这个名字是因为这些值在OrcaFlex的图形的X轴上呈现。现在，这些弧长值可以在X轴或Y轴上显示。为了加剧这种混乱，我们习惯上用z来指代弧长。这只是界面上的一个历史性的怪异现象，你必须习惯。

6.6.2 静态范围图

静态的范围图是类似的。它们在使用pnStaticState周期值进行动态分析后可用，但更常见的是在模型处于静态状态时提取这种结果。在这种情况下，如果你愿意，你可以省略周期参数而依赖默认值。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

rg = line.RangeGraph('有效张力')
```

```
for z, Te in zip(rg.X, rg.Mean):
    print('{:5.1f} {:7.3f}'.format(z, Te))
```

```
    z       特
  0.0  40.358
  5.0  36.393
 15.0  28.546
 25.0  20.802
 35.0  13.332
 45.0   7.087
 55.0   7.087
 65.0  13.332
 75.0  20.802
 85.0  28.546
 95.0  36.393
100.0  40.358
```

6.6.3 用于范围图的objectExtra

有人可能会问，为什么我们需要向RangeGraph方法传递一个ObjectExtra对象。之前我们看到ObjectExtra被用来指定沿线的单一弧长。然而，有些结果取决于径向位置、圆周位置、间隙线名称等等。对于这些结果，我们需要指定这些附加信息。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

inner = line.RangeGraph('von Mises Stress', objectExtra=
    OrcFxAPI.oeLine(RadialPos=OrcFxAPI.rpInner, Theta=0.0)
outer = line.RangeGraph('von Mises Stress', objectExtra=
    OrcFxAPI.oeLine(RadialPos=OrcFxAPI.rpOuter, Theta=0.0)
print('{:>5}{:>7}{:>7}'.format('z', '内', '外'))
for z, vmsInner, vmsOuter in zip(inner.X, inner.Mean, outer.Mean):
     print('{:5.1f} {:7.2f} {:7.2f}'.format(z, vmsInner, vmsOuter)
```

```
    z      内部      外部
  0.0   856.81   856.81
  5.0   852.93   864.32
 15.0   943.20   896.01
 25.0  1199.85  1077.63
 35.0  1788.60  1852.51
 45.0  2755.77  3311.63
 55.0  2755.77  3311.63
 65.0  1788.60  1852.51
 75.0  1199.85  1077.63
 85.0   943.20   896.01
 95.0   852.93   864.32
100.0   856.81   856.81
```

6.6.4 用于范围图的arclengthRange

arclengthRange参数可以用来限制报告的弧长范围。虽然这可以通过手动操作来完成，即获取整条线的输出，并删除不需要的

部分，使用arclengthRange参数允许OrcFxAPI引擎优化结果的推导。因此，除了允许更简单的代码外，这可以带来显著的性能优势。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

rg = line.RangeGraph('有效张力',
    arclengthRange=OrcFxAPI.arSpecifiedArclengths(15.0, 45.0))
print('{:>5}{:>7}'.format('z', 'Te'))
for z, Te in zip(rg.X, rg.Mean):
```

```
    z      特
 15.0  28.546
 25.0  20.802
 35.0  13.332
 45.0   7.087
```

这里我们通过指定最小和最大弧长值来定义弧长范围。通过OrcinaDefaultReal()表示行的开始或结束。

```
#从线的起点到弧长45.....。
tmp = OrcFxAPI.arSpecifiedArclengths(
    OrcFxAPI.OrcinaDefaultReal( ), 45.0)

# .....但使用零肯定是一个更清晰的方式来做到这一点
tmp = OrcFxAPI.arSpecifiedArclengths(0.0, 45.0)

#从弧长45到线的末端.....
tmp = OrcFxAPI.arSpecifiedArclengths(
    45.0, OrcFxAPI.OrcinaDefaultReal( ))

#       这可以说是比
```

作为按弧长指定的替代方法，你可以使用arSpecifiedSections来指定最小和最大截面指数。

最后，arEntireLine()可以用来明确指定整条线的结果。但这很少被使用，因为完全省略该参数更为简单。

6.6.5 适用于范围图的风暴持续时间(stormDurationHours)

stormDurationHours参数只用于频域动力学，并指定用于计算MPM的风暴持续时间（小时）。

6.7 链接统计

链接统计结果报告多个结果变量的统计数据。调用LinkedStatistics获得一个随后可以被查询的结果对象。

```
LinkedStatistics(varNames, period=None, objectExtra=None)
```

现在，周期和objectExtra参数应该不需要进一步解释。这个结果类型的新颖之处在于LinkedStatistics返回的对象，以及如何查询它。

输入OrcFxAPI

```
model = OrcFxAPI.Model()
model.general.StageDuration = 10.0, 200.0
model.environment.WaveType = 'JONSWAP' # 使用不规则波浪
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
line.EndAConnection = vessel.Name
line.EndAX, line.EndAY, line.EndAZ = 40.0, 0.0, 5.0
line.EndBX, line.EndBY, line.EndBZ = 70.0, 0.0, -25.0
model.RunSimulation()

varNames = '有效张力', '弯矩', '曲率' period = 1
objectExtra = OrcFxAPI.oeArcLength(25.0)
stats = line.LinkedStatistics(varNames, period, objectExtra)

#查询张力和弯矩
query = stats.Query('有效张力', '弯矩')
print(' (1) max tension =', query.ValueAtMax)
print(' (2)最大张力的时间=', query.TimeOfMax)
print(' (3) bend moment at this time =', query.LinkedValueAtMax)
print(' (4) min tension =', query.ValueAtMin)
print(' (5) time of min tension =', query.TimeOfMin)
print(' (6) bend moment at this time =', query.LinkedValueAtMin)

#查询张力和曲率
query = stats.Query('Effective Tension', 'Curvature')
print(' (7) time of max tension =', query.TimeOfMax)
print(' (8) 此时的曲率=', query.LinkedValueAtMax) print(' (9) 最
小张力的时间=', query.TimeOfMin) print(' (10) 此时的曲率=',
query.LinkedValueAtMin)

#还可以提取时间序列的统计数据
tss = stats.TimeSeriesStatistics('有效张力')
print('(11) mean =', tss.Mean)
print('(12) stddev =', tss.StdDev)
print('(13) m0 =', tss.m0)
print('(14) m2 =', tss.m2)
print('(15) m4 =', tss.m4)
print('(16) Tz =', tss.Tz)
print('(17) Tc =', tss.Tc)
print('(18) Bandwidth =', tss.Bandwidth)
```

```
(1)      最大张力 = 50.07413101196289
(2)    最大的时间 张力=164.5
(3)         弯矩 此时=0.2841897608585272
(4)      最小张力 = 10.75954818725586
(5)    分钟的时间 张力=167.8
(6)         弯矩 此时=0.6568732168296456
(7)    最大的时间 张力=164.5
(8) 此时的曲率=0.0023682480071543933
```

(9) 最小张力的时间=167.8

(10) 此时的曲率=0.00547394347358038

(11) 平均=32.163606738043335

(12) stddev = 5.348383715390616

(13) m0 = 28.605208367055532

```
(14) m2 = 0.4578602459846008
(15) m4 = 0.012873914746648361
(16) Tz = 7.904166666666668
(17) Tc = 5.9636363636364
(18) 带宽=0.656308392214        2144
```

6.8 时间序列统计

我们在上一节看到了如何从一个链接的统计对象中返回时间序列的统计数据。

TimeSeriesStatistics方法直接返回同样的信息。

```
TimeSeriesStatistics(varNames, period=None, objectExtra=None)
```

6.9 雨流半周期

后处理自动化通常用于定制的疲劳分析、裂纹扩展分析等。这些形式的分析取决于Rainflow循环计数算法。OrcaFlex通过RainflowHalfCycles方法提供了一个周期计数实现。

```
RainflowHalfCycles(varNames, period=None, objectExtra=None)
```

为了简短的例子，我们将循环计算一些容器的结果，但在实际使用中，我们希望看到线型结构结果被循环计算。

```
输入OrcFxAPI

model = OrcFxAPI.Model()
model.general.StageDuration = 10.0, 50.0
model.environment.WaveType = 'JONSWAP' # 使用不规则波浪 vessel =
model.CreateObject(OrcFxAPI.otVessel) model.RunSimulation()

期限=1
halfCycleRanges = vessel.RainflowHalfCycles('X', period)
for halfCycleRange in halfCycleRanges:
    print(halfCycleRange)
```

```
0.159379020333
0.159379020333
0.916084051132
0.916084051132
1.54313793778
2.05583393574
2.63803243637
2.86304700375
```

在设计自动化界面时，我们试图使功能尽可能地灵活可用。例如，目前正在考虑的功能是对一个特定的OrcaFlex结果变量进行周期计数。但是，如果你希望提取一个或多个时间历史，对这些时间历史进行一些后处理，以创建一个衍生结果，然后对该衍生结果进行循环计数，那该怎么办？

我们预计到了这种可能性，并提供了另一种访问该功能的方法。这也是通过一个名为

RainflowHalfCycles的函数，但这个函数是在模块范围内。

雨流半周期(数值)

我们通过对一个阻尼谐波振荡器进行循环计数来说明这个模块范围内函数的使用。

```
输入numpy
import matplotlib.pyplot as pyplot
import OrcFxAPI

def f(x, a, omega, gamma):
    # 带阻尼的谐波振荡器
    返回 numpy.exp(-gamma*x) * a * numpy.cos(omega*x)

# 1000个等距的0和100之间的值
X = numpy.linspace(0.0, 100.0, num=1000)
# 在这些值上评估f
值=[f(x, 10.0, 0.4, 0.02) for x in X]

# 绘制阻尼振荡器
pyplot.plot(values)
pyplot.show()

#计算和输出雨水的半周期 halfCycleRanges =
```

```
3.29894708985
3.85996361396
4.51657039367
5.28460374211
6.18355215086
7.2351081279
8.46583018246
9.9059245794
11.590851258
13.5625335167
15.8692692175
```

6.10 其他结果类型

我们在这里只介绍了最常用的结果类型。还有更多可用的结果类型--关于其他可用结果的细节，请参考文档。

7 与其他Python模块互动

正如一开始提到的，通过Python实现OrcaFlex自动化的好处之一是有大量的扩展包，可以简化某些任务的编码。我们已经在上面的代码中看到并使用了一些这样的包，但现在我们将强调一些最常用于OrcaFlex自动化的包。这个列表并不意味着是全面的，而是为了让大家了解可用的东西。

本节中使用的少数软件包是基本 Python 安装的一部分。pip软件包管理器可以用来安装你的安装中可

能缺少的任何扩展包。

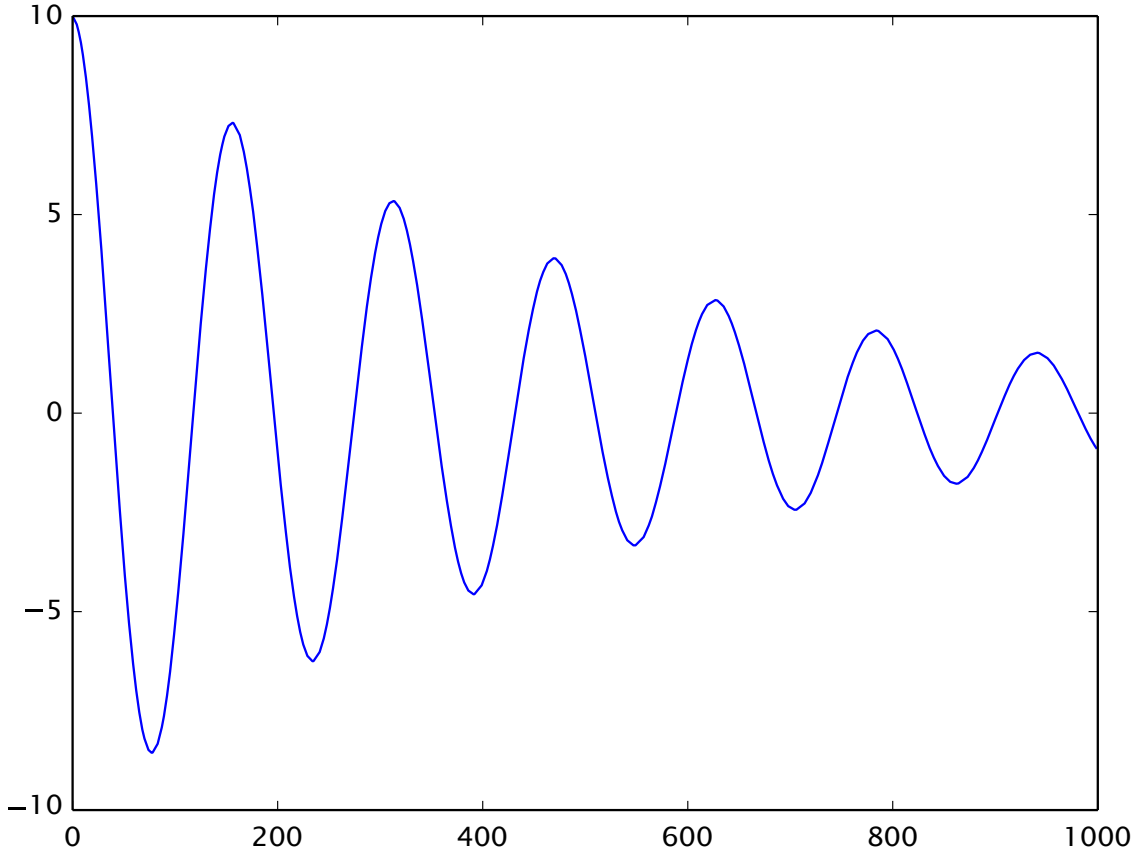


图5：阻尼谐波振荡器

7.1 用xlrd、xlsxwriter和openpyxl读取和写入Excel文件

虽然有一些OrcaFlex用户狂热地避开了Excel，但作为OrcaFlex工作流程的一部分，能够读和写Excel工作簿往往是非常有用的。有许多扩展包可用于与Excel互操作，值得一提的是xlrd、xlsxwriter和openpyxl，我们都使用过，可以推荐。这些包都不需要安装Excel，而是直接对Excel文件进行操作。

我们将从一个使用openpyxl的简单例子开始。

```
输入openpyxl

waveHeights = [0.5, 1.0, 2.0, 3.0, 4.0, 5.5] 。
wavePeriods = [6.0, 7.0, 8.0, 10.0] 。

book = openpyxl.Workbook()
sheet = book.active
sheet.title = 'Data' 。
boldFont = openpyxl.style.Font(bold=True)
centredAlignment = openpyxl.style.Alignment(horizontal='center')
sheet['A1'] = 'Height'(高度)
sheet['A1'].font = boldFont
sheet['A1'].alignment = centredAlignment
sheet['B1'] = 'Period'
sheet['B1'].font = boldFont
sheet['B1'].alignment = centredAlignment

行 = 2
for H in waveHeights:
    for T in wavePeriods:
        sheet.cell(row=row,  column=1).value  =
        H  sheet.cell(row=row,  column=2).value
        = T row += 1
```

在这里，我们已经创建了一个简单的数据表格。现在我们展示如何读取这些数据并使用它来创建OrcaFlex模型。

```
import OrcFxAPI
import openpyxl

book = openpyxl.load_workbook('scratch/LoadCaseData.xlsx')
sheet = book['Data']
model = OrcFxAPI.Model()

行 = 2
while sheet.cell(row=row, column=1).value。
    H = sheet.cell(row=row,
    column=1).value T =
    sheet.cell(row=row, column=2).value
    model.environment.WaveHeight = H
    model.environment.WavePeriod = T

    id = 行 - 1
    fileName = 'scratch/case{0:03d},H={1:.1f},T={2:.1f}.dat'.format(
        id, H, T)
    model.SaveData(fileName)。
```

这并不是一个特别现实的例子，但它确实希望能让人看到这个包所提供的可能性。

另一个用例是收集从OrcaFlex输出的结果，在Excel中呈现。只是为了好玩，我们将展示如何做到这一点，并制作Excel图表。

```
import OrcFxAPI
import openpyxl

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('有效张力')

book = openpyxl.Workbook()
sheet = book.active
sheet.append(['Arclength', 'Te'] )
for data in zip(Te.X, Te.Mean):
    sheet.append(data)

chart = openpyxl.chart.ScatterChart()
chart.title = '静态下的有效张力' chart.x_axis.title =
'Arclength (m)' chart.y_axis.title = 'Te (kN)' 。

N = len(Te.X)
x     =     openpyxl.chart.Reference(sheet,     min_col=1,     min_row=2,
max_row=N+1)    y    =    openpyxl.chart.Reference(sheet,    min_col=2,
min_row=2, max_row=N+1) series = openpyxl.chart.Series(y, x)
chart.series.append(series)
sheet.add_chart(chart,
'D2')

book.save('scratch/chart_embedded.xlsx')
```

openpyxl中对图表支持的一个限制是，图表必须位于工作表中。xlsxwriter软件包能够在图表表中生成图表。让我们用这个包重新创建前面的例子。请注意，xlsxwriter的语法与openpyxl的语法不同。这两个软件包在网上都有很好的文档，如果你需要自己编写这样的代码，你将需要利用它。

```
import OrcFxAPI
import xlsxwriter

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('有效张力')

book =
xlsxwriter.Workbook('scratch/chart_sheet.xlsx') sheet
= book.add_worksheet('ChartData')
sheet.write_row('A1', ['Arclength', 'Te'])
sheet.write_column('A2', Te.X)
sheet.write_column('B2', Te.Mean)

chart = book.add_chart({'type': 'scatter',
                        '子类型': '直' })
```

```
    '名称'：'有效张力'。
    'categories': '=ChartData!$A$2:$A${0}'.format(len(Te.X) + 1),
    'values': '=ChartData!$B$2:$B${0}'.format(len(Te.X) + 1) 。
})
chart.set_title({'name': 'Effective Tension in static state'})
chart.set_x_axis({'name': 'Arclength (m)' 。
                  'min':Te.X[0],
                  '最大'。Te.X[-1]})
chart.set_y_axis({'name': 'Te (kn)'})
chart.set_legend({'none': True})
chartsheet = book.add_chartsheet('有效张力图') chartsheet.set_chart(图表)

book.close()
```

7.2 用json、yaml或h5py进行序列化

有时有必要将数据结构保存到文件中。例如，你可能有一个工作流程，一个程序提取结果，而下一个程序对这些结果进行后处理。事实上，你可能有多个后处理步骤，通过将任务分解成小块，你可以选择做一些后处理，但不一定是全部。OrcaFlex后计算动作将是执行结果提取的有效方式。

以这种方式拆分整个后处理任务，可以获得更大的灵活性。这种方法确实需要有保存和加载你所操作的数据的能力。你可能会考虑使用文本文件或Excel文件来实现这一目的，这可能是一种合理的方法。然而，当数据比较复杂，有更多的结构时，这种平面格式就很难操作了。像JSON、YAML、HDF5等结构化的文件格式可以非常有效。

我们将从JSON（http://www.json.org/）开始，它是由标准的Python包json提供的。我们将提取一条线的范围图，并将信息保存为JSON格式的文本文件。

```
import json
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('有效张力')

数据 = {
    'title':'静态下的有效张力',
    'x_axis_title':'Arclength (m) ',
    'y_axis_title':'Te (kN)'。
    'x_axis_values':Te.X.tolist(),
    'y_axis_values':Te.Mean.tolist()
}
with open('scratch/serialization.json', 'w') as f:
```

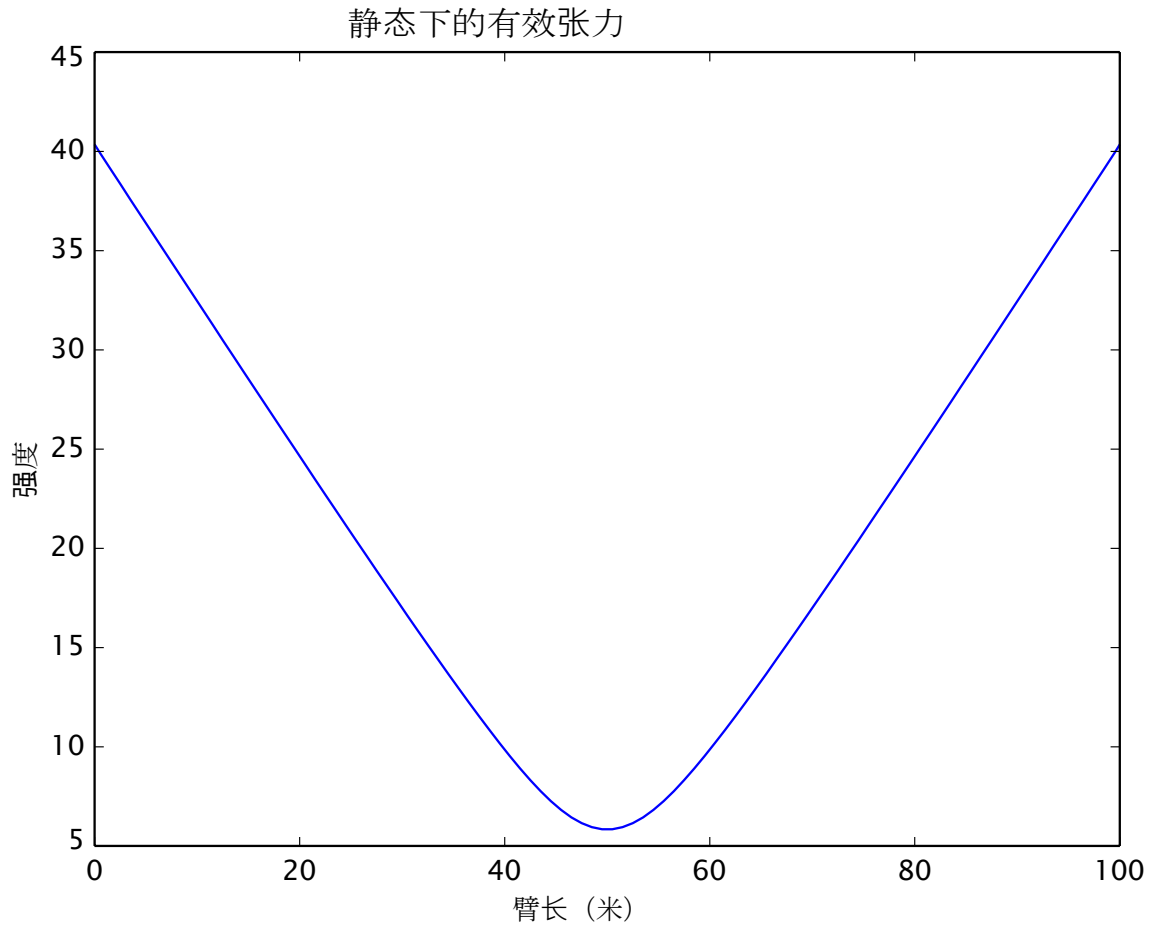
注意，我们使用了numpy数组对象的tolist方法，将它们转换为普通的Python列表对象。这是因为numpy数组对象是不能被JSON序列化的。

我们可以用一个读取文件的程序来跟进，并绘制其中的数据。这第二个程序没有使用OrcFxAPI，它能够从上一步创建的文件中获得所有需要的信息。

```
输入json
输入matplotlib.pyplot作为pyplot

with open('scratch/serialization.json', 'r') as f:
    data = json.load(f)

pyplot.plot(data['x_axis_values'], data['y_axis_values'])
pyplot.title(data['title'])
pyplot.xlabel(data['x_axis_title'])
pyplot.ylabel(data['y_axis_title'] )
```



我们可以用接近相同的代码执行相同的任务，但这次使用YAML（http://yaml.org/）文件格式。对于任何涉足过OrcaFlex文本数据文件的OrcaFlex用户来说，YAML都是熟悉的。尽管YAML和JSON非常相似（事实上，JSON文件往往可以成为有效的YAML文件），但YAML格式对人的眼睛来说更容易阅读。

```
import yaml
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('有效张力')

数据 = {
```

```
    'title':'静态下的有效张力',
    'x_axis_title':'Arclength (m)',
    'y_axis_title':'Te (kN)'。
    'x_axis_values':Te.X.tolist(),
    'y_axis_values':Te.Mean.tolist()
}
with open('scratch/serialization.yaml', 'w') as f:
    yaml.dump(data, f)
```

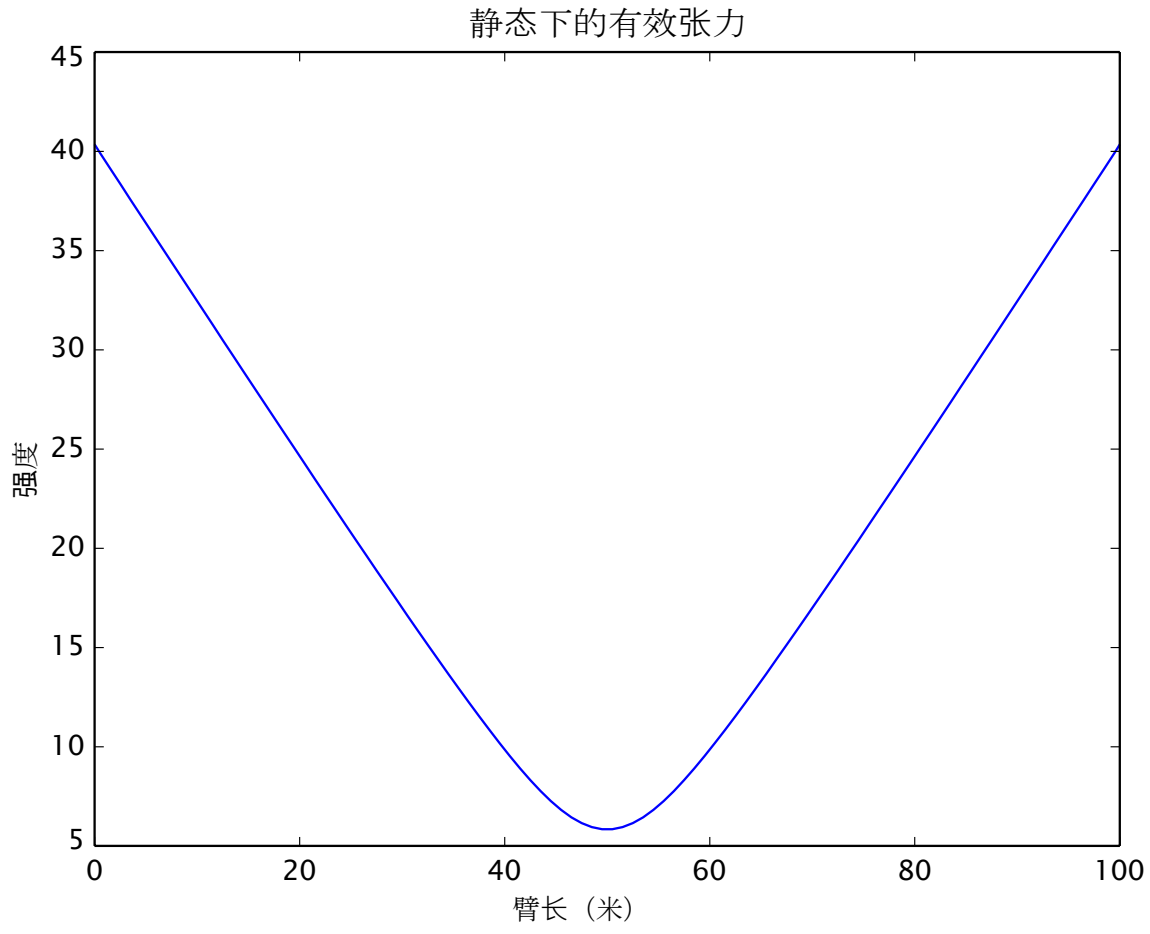
我们再次在numpy数组对象上使用tolist。尽管numpy数组对象是YAML可序列化的，但其输出的可读性比Python列表对象的输出要差很多。

再一次，我们使用近乎相同的代码来读取文件。

```
输入yaml
输入matplotlib.pyplot作为pyplot

with open('scratch/serialization.yaml', 'r') as f:
    data = yaml.load(f)

pyplot.plot(data['x_axis_values'], data['y_axis_values'])
pyplot.title(data['title'])
pyplot.xlabel(data['x_axis_title'])
pyplot.ylabel(data['y_axis_title'] )
```



然而，另一种可能性是HDF5（

https://en.wikipedia.org/wiki/Hierarchical_Data_Format）格式。这是一种二进制

在科学数据应用中广泛使用的格式。当处理非常大量的数据时，HDF5可能提供比JSON或YAML更有效的存储和访问。然而，它在使用上无疑比JSON或YAML更复杂。我们将在h5py包的帮助下，用HDF5实现我们现在熟悉的例子。

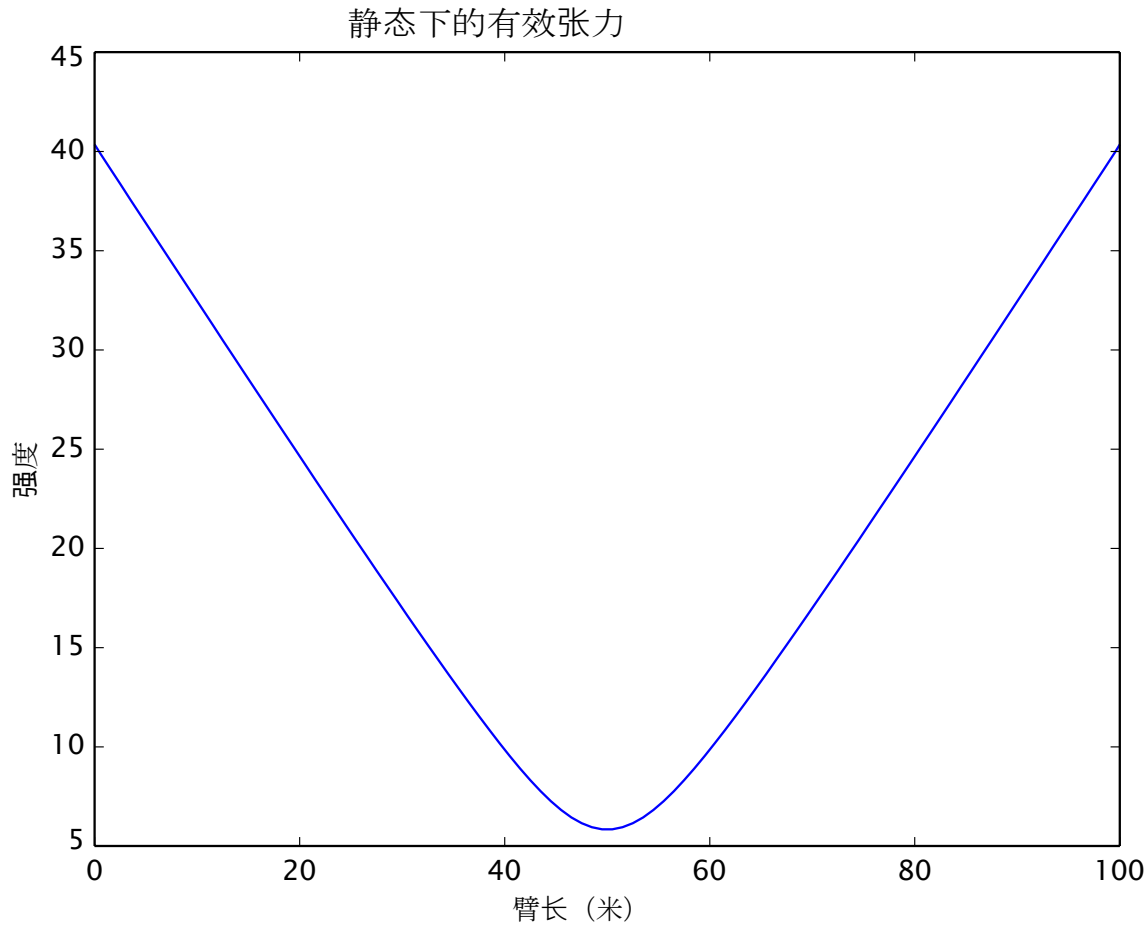
```
import h5py
import numpy
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('有效张力')

f = h5py.File('scratch/serialization.hdf5', 'w')
dataset = f.create_dataset('values', data=numpy.column_stack([Te.X,
Te.Mean]))
dataset.attrs['title'] = '静态下的有效张力'
dataset.attrs['x_axis_title'] = 'Arclength (m)'
dataset.attrs['y_axis_title' = 'Te (kN)'
f.close()
```

```
输入h5py
输入matplotlib.pyplot作为pyplot

f = h5py.File('scratch/serialization.hdf5',
'r') dataset = f['values']
pyplot.plot(dataset[...,0], dataset[...,1])
pyplot.title(dataset.attrs['title'])
pyplot.xlabel(dataset.attrs['x_axis_title'])
pyplot.ylabel(dataset.attrs['y_axis_title')
f.close()
```



现在我们要尝试一个稍微复杂的例子，并利用HDF5的分层性质。让我们试着提取多个范围图。

```
import h5py
import numpy
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()

varNames = [
    有效张力"、"弯曲力矩"、
    "倾角"、"深度
]
varUnits = {}。
for details in
    line.varDetails(OrcFxAPI.rtRangeGraph): varUnits[
    details.VarName] = details.VarUnits

f = h5py.File('scratch/serialization_advanced.hdf5', 'w')
for varName in varNames:
    rg = line.RangeGraph(varName)
    dataset = f.create_dataset(varName, data=numpy.column_stack([rg.X,
rg.Mean]))
    dataset.attrs['x_axis_title'] = 'Arclength
        ({0})'.format(varUnits['X'])
    dataset.attrs['y axis title'] = \\
```

```
        '{0} ({1}) '.format(varName, varUnits[varName])

f.close()
```

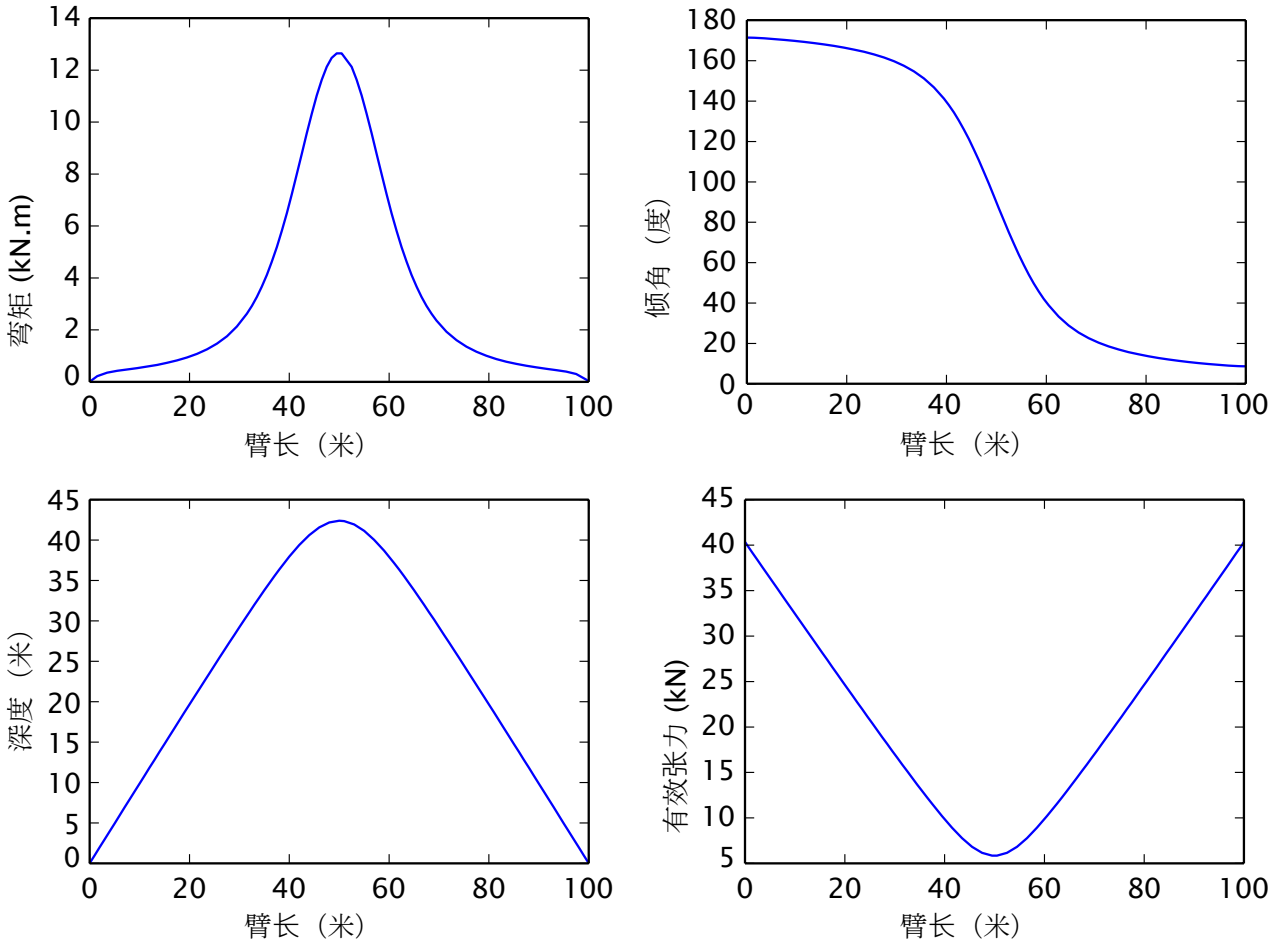
接下来我们加载HDF5文件，并列举其数据集，然后绘制每个数据集。

```
import h5py
import math
输入matplotlib.pyplot作为pyplot

f = h5py.File('scratch/serialization_advanced.hdf5', 'r')
count = len(f)
ncols = int(math.sqrt(count))
nrows = math.ceil(count / ncols)
i = 1
for varName, dataset in f.items():
    pyplot.subplot(nrows, ncols, i)
    pyplot.plot(dataset[...,0], dataset[...,1])
    pyplot.xlabel(dataset.attrs['x_axis_title'])
    pyplot.ylabel(dataset.attrs['y_axis_title')
    i+= 1

f.close()

pyplot.tight_layout() # 避免标签重叠
pyplot.show()
```



这个例子同样可以用json或yaml来实现，但请注意，h5py包能够直接接受numpy数组对象。因为HDF5专注于科学应用，它是为处理数字数据而准备的。

这仅仅是对HDF5所能进行的工作的一个初步了解。如果你想了解更多，可以在网上找到很多资源。事实上，h5py甚至不是处理HDF5的唯一选择--PyTables（http://www.pytables.org/）和pandas（http://pandas.pydata.org/）是广泛使用的替代品。

7.3 用matplotlib作图

我们已经看到了一些使用 matplotlib 包进行绘图的例子。我们没有再创造一些例子，而是参考了前面出现的那些例子，并注意到matplotlib是Python绘图的事实上的标准。

7.4 使用numpy的线性代数

numpy包在Python的科学编程中被广泛使用。尽管它不是一个内置包，但它的使用非常广泛，我们建议你在安装Python时都要安装它。事实上，许多针对科学应用的 Python 发行版都包括 numpy。

正如已经提到的，如果numpy可用，那么OrcaFlex的Python接口将利用它并为非标量数据返回numpy数组对象。在第6.3节中，我们看到了一个程序的例子，该程序使用numpy包提供的设施对这些numpy数组对象进行后处理。

7.5 用scipy进行插值

从形式上看，SciPy是科学计算工具的堆栈。该网站（http://www.scipy.org/about.html）列出了以下核心软件包，其中一些我们已经遇到过了。

- Python，一种通用的编程语言。它是解释型和动态类型的，非常适用于交互式工作和快速原型设计，同时又足够强大，可以编写大型应用程序。
- NumPy，是数值计算的基本包。它定义了数字数组和矩阵类型以及对它们的基本操作。
- SciPy库，是数字算法和特定领域工具箱的集合，包括信号处理、优化、统计和更多。
- Matplotlib是一个成熟和流行的绘图包，它提供了出版质量的2D绘图以及初级的3D绘图。
- pandas，提供高性能、易于使用的数据结构。
- SymPy，用于符号数学和计算机代数。
- IPython，一个丰富的互动界面，让你快速处理数据和测试想法。IPython笔记本在你的网络浏览器中工作，让你以一种容易复制的形式记录你的计算。
- nose，一个用于测试Python代码的框架。

我们所说的scipy只是指这个堆栈中的一个包，即SciPy库，它是一个非常丰富的数值算法的来源。为了选取一个例子，我们将演示如何

使用scipy进行内插。我们将创建一个OrcaFlex可变弯曲刚度数据源，它表示一个针对曲率的弯曲力矩表。

```
输入numpy
import scipy.interpolate
import OrcFxAPI

model = OrcFxAPI.Model()
stiffness = model.CreateObject(OrcFxAPI.otBendingStiffness)
stiffness.IndependentValue = 0.0, 0.012, 0.021, 0.074, 0.28
stiffness.DependentValue = 0.0, 1200.0, 1450.0, 1700.0, 1850.0
```

现在我们将使用变量数据源的数据创建一个scipy插值对象。我们没有指定一个插值方法，所以使用默认的线性插值方法。这与OrcaFlex中用于可变弯曲刚度数据的插值方法一致。

```
interp = scipy.interpolate.interp1d(
    stiffness.IndependentValue,
    stiffness.DependentValue)
)
```

最后，我们可以进行插值来计算弯矩，我们在10个等距的曲率值上进行插值计算。

```
for C in numpy.linspace(interp.x[0], interp.x[-1],
    num=10):M = interp(C)
    print('C =', C, 'M =', M)
```

```
C = 0.0 M = 0.0
C = 0.0311111111111 M = 1497.6939203354298
C = 0.0622222222222 M = 1644.4444444444446
C = 0.0933333333333 M = 1714.0776699029127
C = 0.124444444444 M = 1736.7313915857605
C = 0.155555555556 M = 1759.3851132686084
C = 0.186666666667 M = 1782.0388349514562
C = 0.217777777778 M = 1804.6925566343043
C = 0.248888888889 M = 1827.3462783171522
C = 0.28 M = 1850.0
```

这是我们在OrcaFlex测试代码中非常频繁使用的模式。

7.6 用scipy找根

scipy的另一个可能的应用是寻根。这就是寻找$f(x) = 0$形式的方程的解决方案，其中x是未知的。为了给这个非常抽象的数学定义一些背景，让我们考虑一个带有浮力部分的OrcaFlex立管模型。

```
import scipy.optimation
import OrcFxAPI

model = OrcFxAPI.Model()

riserType = model.CreateObject(OrcFxAPI.otLineType)
riserType.Name = 'Riser type' 。
```

```
buoyedType = model.CreateObject(OrcFxAPI.otLineType)
buoyedType.Name = 'Buoyed type' 。
riser = model.CreateObject(OrcFxAPI.otLine)
riser.Name = 'Riser' 。

riser.LineType = riserType.Name, buoyedType.Name, riserType.Name
riser.Length = 90.0, 80.0, 100.0
riser.TargetSegmentLength = 2.0, 2.0, 2.0
riser.EndBX = riser.EndAX + 160.0
riser.EndBConnection = 'Anchored'
riser.EndBHeightAboveSeabed = 0.0

buoyedType.WizardCalculation = 'Line with Floats'
buoyedType.FloatBaseLineType = riserType.Name
buoyedType.FloatDiameter = 1.1
buoyedType.FloatLength = 2.0
buoyedType.FloatPitch = 8.0
buoyedType.InvokeWizard () 。
```

我们所建立的模型如图6所示。

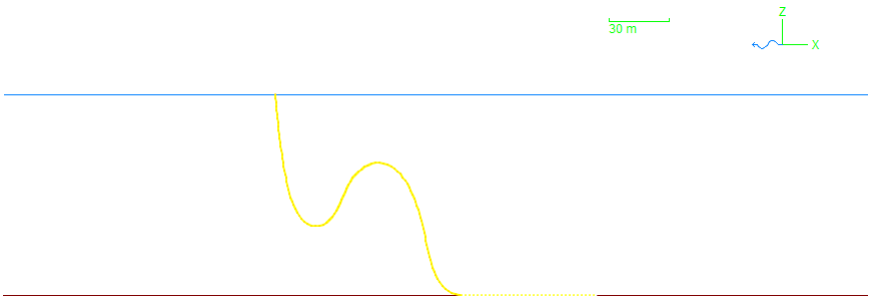


图6：带有浮力部分的立管，找到根部之前

现在，假设我们希望修改浮动长度，以达到最大猪圈Z坐标为-50米的目标。我们可以使用scipy.optimize中的fsolve方法来实现这一目标。

```
targetZ = -50.0
initialFloatLength = buoyedType.FloatLength
arclengthRange = OrcFxAPI.arSpecifiedSections (2, 2
) 。

def calcHogBendZ(floatLength):
    buoyedType.FloatLength = floatLength[0]
    buoyedType.InvokeWizard()
    model.CalculateStatics()
    Z = riser.RangeGraph('Z', arclengthRange=arclengthRange).Mean
    hogBendZ = max(Z) # Z值在hog bend的高点 print('Float length = {0},
    hog bend Z =
{1}'.format(floatLength[0], hogBendZ))
    返回 hogBendZ - targetZ
```

```
Float length = 2.0, hog bend Z = -
34.116378169350945 Float length = 2.0, hog bend Z =
-34.116378169350945 Float length = 2.0, hog bend Z
```

= -34.116378169350945
浮动长度=2.0000000298023224，猪圈弯曲z=-34.116376901460505

```
浮动长度=1.6266484834636032，猪弯Z=-49.88879328629302 浮动长度
=1.6240160904595078，猪弯Z=-50.003984076185006 浮动长度
=1.6241071364041273，猪弯Z=-49.99998031148309
浮动长度=1.624106688685604，猪弯Z=-49.9999999965303 浮动长度
=1.6241066886066888，猪弯Z=-49.9999999999915
```

将此映射到最初的抽象方程，$f(x) = 0$，未知变量x代表浮点长度。函数f被封装在`calcHogBendZ`中，它的作用如下。

- 设置线型向导`FloatLength`数据项为x。
- 通过调用`InvokeWizard`来调用行类型向导。
- 进行静态计算。
- 使用`arclengthRange`提取直线第2段（代表猪弯的浮力段）的Z坐标。
- 找到猪弯的最大Z坐标。
- 减去目标Z值。

这是一个相当复杂的函数，但是`fsolve`并不关心这个问题，而是非常愉快地找到了根。最后的模型如图7所示。

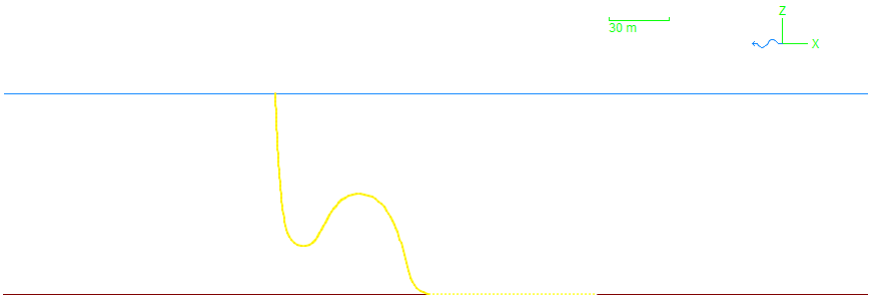


图7：带浮力部分的立管，找到根部后

这个例子展示了OrcaFlex与Python及其扩展包相结合的力量和灵活性。

8 进一步探索

本文件试图涵盖广泛的主题，但没有深入研究每一个细节。那么，如果你确实需要学习更多的细节呢？OrcaFlex文档的Python接口（www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/OrcFxAPIHelp/）有每个类和函数的全面细节。

对于更勇敢的人，你可以阅读`OrcFxAPI`模块的Python源代码。这可以在OrcaFlex安装目录下的`OrcFxAPI/Python`子目录中的OrcFxAPI.py文件中找到。通常情况下，参考文档应该提供足够的信息，但有时阅读源代码可以帮助获得理解。但请注意，你不应该修改这个文件。

有很多关于 Python 的一般资源。官方文档 (docs.python.org/) 非常好。我们还推荐Mark Lutz的《学习Python》一书（http://learning-python.com/books/about-lp.html），作为Python的优秀介绍。

Stack Overflow（http://stackoverflow.com/)是一个很好的资源，可以为非常具体的问题提供答案。通常情况下，在网上搜索包含你的问题的关键词，就会在Stack Overflow上找到任何数量的优秀答案。

对于最近版本的Python，pip包管理器通常是安装扩展包的最佳方式。例如，在命令提示符下执行以下命令，将安装numpy。

```
> pip install numpy 收集numpy
  下载numpy-1.10.4-cp35-none-win_amd64.whl (7.5MB) 100%
    |################################| 7.5MB 114kB/s
安装已收集的软件包: numpy 成功地安装了numpy-1.10.4
```

然而，并非所有的软件包都能通过pip正确安装。在准备本文时，我试图用 `pip install scipy` 来跟进前面的命令。由于我没有试图理解的原因，这一次失败了。在这种情况下该怎么做呢？Christophe Gohlke 在他的网站 www.lfd.uci.edu/~gohlke/pythonlibs/ 上维护了大量的 Python 扩展包。对于那些难以获得或安装的 Python 包，Christophe 的网站是非常宝贵的。

最后，如果一切都失败了，你可以随时通过电子邮件（orcina@orcina.com）联系我们Orcina。