Thematic Tutorials

Release 7.1

The Sage Development Team

译者：姚振 (HITwh)

March 20, 2016

注：译自 *thematic_tutorials.pdf* 的 P 314 - P 346

Chapter TEN：使用 Sage 进行数值计算

设计这篇文档的目的是给读者介绍在 Sage 上做数值计算时的工具。数值计算本质上意味着机器浮点精度上的计算。我们假设阅读第一部分的读者已经熟悉 Python 和 Sage 了，特别是对于如：最优化问题，数值化的线性代数，求解常微分方程或偏微分方程的数值解等。第二部分是关于使用编译代码的这又增加了对预备知识的了解，我们假定阅读的读者是熟悉 C 语言或者 Fortran 语言的。第三部分是关于 MPI 和并行编程的，仅需要 Python 的相关知识就可以了，如果熟悉 MPI 应该会有帮助。最后一部分是关于如何在 Sage 实现 3D 可视化的。

在当前版本的文档中我们假定读者熟悉数值分析的相关技术。本章节的目的不是教授数值分析的，二是解释如何使用 Python/Sage 实现你的想法。当然这篇文档并不是将所有都囊括在内的。而他的目的是给以读者指引，使读者能定位于数值计算相关的包和在何处可以获取等多的信息。

数值工具

Sage 有许多不同的组件可用于数值分析. 其中有三个包值得一提，他们分别是 NumPy，SciPy，and Cvxopt。 Numpy 是一个十分优秀的包，它为 Python 提供了简单快捷的数组操作。它包含许多基本的线性代数程序，矢量化数学程序，随机数生成器等。它支持的编程风格类似于 Matlab，并且对于许多 Matlab 上的方法，在 NumPy 上都有类似的实现。SciPy 是基于 NumPy 的，它为最优化问题，求根运算，统计学，线性代数，线性插值，快速傅里叶变换，数字信号处理等提供了许多各不相同的包。最后，Cvxopt 是一个解决最优化问题的包，它可以解决线性规划和二次规划问题，并且具有一个友好的线性代数界面。现在我们将介绍一下每个包。

批注 [by1]: that provides fast array facilities to python.

在开始之前，我们首先将注意力转向 NumPy 和 Matlab 的一个对比，它也包含 Matlab 命令在 NumPy 上的等价命令，链接：http://www.scipy.org/NumPy_for_Matlab_Users。 如果你不熟悉 Matlab，这很好，甚至更好，意味着你在考虑事情应如何完成时没有任何先入为主的观念。 还有，这个链接有一个关于 SciPy 和 NumPy 的不错的教程，比我们的更加综合。链接：
http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf

NumPy

NumPy 在默认情况下不会被 Sage 自动导入。 要使用 NumPy 我们首先要：

```
sage: import numpy
```

在 NumPy 中计算的基本对象是数组。创建一个数组是容易的。

```
sage: l=numpy.array([1,2,3])
sage: l
array([1, 2, 3])
```

NumPy 的数组可以存储 Python 的任意对象。 然而，从速度方面考虑，数值类型将自动在可能的情况下转换为本机硬件类型 （如 int、float、等）。 如果一个变量或数值的精度无法转换成本机硬件类型，这时将创建一个 Sage 的数组对象。你可以在这些数组上面进行任何运算，但是它们可能比原生类型慢一些。当 NumPy 的数组包含 Sage 或 Python 对象时，数据类型是被显式的表示为 object。当 NumPy 打印数组时，如果没有数据类型被显式的表示，那这种类型不是 float 就是 int。

批注 [by2]: When the numpy array contains Sage or python objects, then the data type is explicitly(明确地) printed as object.

```
sage: l=numpy.array([2**40, 3**40, 4**40])
sage: l
array([1099511627776, 12157665459056928801,
1208925819614629174706176], dtype=object)
sage: a=2.0000000000000000001
sage: a.prec() # 精度高于硬件浮点数
67
sage: numpy.array([a,2*a,3*a])
array([2.0000000000000000000, 4.0000000000000000000,
6.0000000000000000000], dtype=object)
```

一个数组的 dtype 属性表明这个数组的类型。为了更快的进行数值计算，希望它是某种形式的 float。如果数据类型是 float 类型，那么数组将被存储为机器 float 类型，这将占用更小的存储空间，获取更高的操作速度。

```
sage: l=numpy.array([1.0, 2.0, 3.0])
sage: l.dtype
dtype('float64')
```

你可以通过指定 dtype 参数创建指定的数据类型的数组。如果你要确保通过机器 float 类型计算，一个好方法是指定 dtype=float 创建数组。

```
sage: l=numpy.array([1,2,3], dtype=float)
sage: l.dtype
dtype('float64')
```

你可以像 list 类型一样访问数组内的元素，也就是切片操作。

```
sage: l=numpy.array(range(10),dtype=float)
sage: l[3]
3.0
sage: l[3:6]
array([ 3.,  4.,  5.])
```

你可以做基本的算术运算。

```
sage: l+l
```

```
array([  0., 2., 4., 6., 8.,10.,12.,14.,16.,18.])
sage: 2.5*l
array([  0. ,2.5,5. ,7.5,10. ,12.5,15. ,17.5,20. ,22.5])
```

注意：l * l将会使用 l 与 l 的各对应分量分别相乘。为了得到点积，需要使用numpy.dot()。

```
sage: l*l
array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.,  64.,
81.])
sage: numpy.dot(l,l)
285.0
```

我们也可以创建二维的数组：

```
sage: m = numpy.array([[1,2],[3,4]])
sage: m
array([[1, 2],
       [3, 4]])
sage: m[1,1]
4
```

这基本上等价于如下：

```
sage: m=numpy.matrix([[1,2],[3,4]])
sage: m
matrix([[1, 2],
        [3, 4]])
sage: m[0,1]
2
```

不同之处在于，使用numpy.dot()时 m 仅仅被视为数组。特别的m * m是各分量相乘；然而，使用numpy.matrix()时将做矩阵乘法。我们也可以做矩阵向量乘法和矩阵加法：

```
sage: n = numpy.matrix([[1,2],[3,4]],dtype=float)
sage: v = numpy.array([[1],[2]],dtype=float)
sage: n*v
matrix([[  5.],
        [ 11.]])
sage: n+n
matrix([[ 2.,  4.],
        [ 6.,  8.]])
```

如果 n 是通过 numpy.array()创建的，然后想做矩阵向量乘法，用 numpy.dot(n, v)。

所有的 NumPy 数组都有一个 shape 属性。对操作而言，这是一个很有用的属性：

```
sage: n = numpy.array(range(25),dtype=float)
sage: n
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,
9.,  10.,
```

```
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,
20.,  21.,
        22.,  23.,  24.])
sage: n.shape=(5,5)
sage: n
array([[  0.,   1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.],
       [ 15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.]])
```

它将一维数组转化为了$5*5$的数组。

NumPy 的数组也可以进行分片操作：

```
sage: n=numpy.array(range(25),dtype=float)
sage: n.shape=(5,5)
sage: n[2:4,1:3]
array([[ 11.,  12.],
       [ 16.,  17.]])
```

更重要的是，切片矩阵是引用原矩阵的：

```
sage: m=n[2:4,1:3]
sage: m[0,0]=100
sage: n
array([[   0.,    1.,    2.,    3.,    4.],
       [   5.,    6.,    7.,    8.,    9.],
       [  10.,  100.,   12.,   13.,   14.],
       [  15.,   16.,   17.,   18.,   19.],
       [  20.,   21.,   22.,   23.,   24.]])
```

你或许已经注意到原矩阵已经改变了，这或许是你希望也或者不是。如果你想修改切片矩阵，但是不想修改原矩阵，就需要作原矩阵的拷贝：

```
m=n[2:4,1:3].copy()
```

一些特别有用的命令：

```
sage: x=numpy.arange(0,2,.1,dtype=float)
sage: x
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,
0.9,  1. , 1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])
```

你可以看到 numpy.arange()创建了一个以 0.1 为步长从 0 到 2 的数组。有一个有用的命令 numpy.r_()，最好是用例子来解释：

```
sage: from numpy import r_
sage: j=numpy.complex(0,1)
sage: RealNumber=float
sage: Integer=int
sage: n=r_[0.0:5.0]
```

```
sage: n
array([ 0.,  1.,  2.,  3.,  4.])
sage: n=r_[0.0:5.0, [0.0]*5]
sage: n
array([ 0.,  1.,  2.,  3.,  4.,  0.,  0.,  0.,  0.,  0.])
```

numpy.r_() 提供了高效的构建 NumPy 数组的简约表达法。 注意：上面的 0.0:5.0 是 0.0, 1.0, 2.0, 3.0, 4.0 的简约表达式。 假如我们想把 0 到 5 分成 10 个数值，我们可以如下操作：

```
sage: r_[0.0:5.0:11*j]
array([ 0. ,0.5,1. ,1.5,2. ,2.5,3. ,3.5,4. ,4.5,5. ])
```

符号 0.0:5.0:11*j 被扩展为 0 到 5 之间 11 个（包含端点）等距的数值。注意：j 是 NumPy 虚构的数，它是创建数组的特殊语法。我们可以把所有这些技巧联合起来：

```
sage: n=r_[0.0:5.0:11*j,int(5)*[0.0],-5.0:0.0]
sage: n
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,
4.5,  5. ,  0. ,  0. ,  0. ,  0. ,  0. , -5. , -4. , -3. , -2. ,
-1. ])
```

另一个有用的命令是 numpy. meshgrid()，它用来生成网状栅格。举个例子，假设你想计算$f(x,y) = x^2 + y^2$在等距栅格$\Delta x = \Delta y = 0.25, 0 \leq x,y \leq 1$上的值。你可以按照如下操作：

```
sage: import numpy
sage: j=numpy.complex(0,1)
sage: def f(x,y):
....:     return x**2+y**2
sage: from numpy import meshgrid
sage: x=numpy.r_[0.0:1.0:5*j]
sage: y=numpy.r_[0.0:1.0:5*j]
sage: xx,yy= meshgrid(x,y)
sage: xx
array([[ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
       [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
       [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
       [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
       [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ]])
sage: yy
array([[ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ],
       [ 0.25,  0.25,  0.25,  0.25,  0.25],
       [ 0.5 ,  0.5 ,  0.5 ,  0.5 ,  0.5 ],
       [ 0.75,  0.75,  0.75,  0.75,  0.75],
       [ 1.  ,  1.  ,  1.  ,  1.  ,  1.  ]])
sage: f(xx,yy)
array([[ 0.    ,  0.0625,  0.25  ,  0.5625,  1.    ],
       [ 0.0625,  0.125 ,  0.3125,  0.625 ,  1.0625],
       [ 0.25  ,  0.3125,  0.5   ,  0.8125,  1.25  ],
```

```
       [ 0.5625,  0.625 ,  0.8125,  1.125 ,  1.5625],
       [ 1.    ,  1.0625,  1.25  ,  1.5625,  2.    ]])
```

你可以看到numpy.meshgrid() 产生了一对矩阵, 在这里表示为 xx 和 yy，这样 (xx[i,j],yy[i,j]) 有相匹配的 (x[i],y[j]) 。由于在栅格上求解 f 是，我们仅需要求解在 xx, yy 的每一对条目上。因为 NumPy 能自动的在数组上按照分量形式执行算术运算，所以在栅格上计算函数的值是容易的，仅需要很少的代码。

批注 [by3]: such that (xx[i,j],yy[i,j]) has coordinates (x[i],y[j])

numpy.linalg 是一个有用的模块。如果你想解一个方程 $\boldsymbol{Ax} = \boldsymbol{b}$，仅需要：

```
sage: import numpy
sage: from numpy import linalg
sage: A=numpy.random.randn(5,5)
sage: b=numpy.array(range(1,6))
sage: x=linalg.solve(A,b)
sage: numpy.dot(A,x)
array([ 1.,  2.,  3.,  4., 5.])
```

这里创建了已一随机的 5*5 的矩阵 A，然后当 b = [0.0,1.0,2.0,3.0,4.0]求解 $\boldsymbol{Ax} = \boldsymbol{b}$。在 numpy.linalg 模块中还含有一些其他的程序，他们大多数是不言而明的。例如 qr 和 lu 程序是做 QR 和 LU 分解的。命令 eigs 是计算矩阵的特征值的。你可以总是执行 < function name >? 来获取程序的文档，这对这些程序都是合适的。

希望这能给你一个对 NumPy 的初始印象。你应该继续探索这个包，因为这里面还含有很多的功能。

SciPy

类似于上一节，这里我推荐：http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf。这里有很多 SciPy 模块，尤其是 ***scipy.optimize***, ***scipy.stats***, ***scipy.linalg***, ***scipy.linsolve***, ***scipy.sparse***, ***scipy.integrate***, ***scipy.fftpack***, ***scipy.signal***, ***scipy.special***。这些包中的大部分都有比较好文档，通常你可以顾函数名而思义。 我建议去探索他们，例如，如果你：

```
sage: import scipy
sage: from scipy import optimize
```

然后：

```
sage: optimize.[tab]
```

将会显示一个可用函数的列表。你可以看到一连串的程序都可以寻找最小值。特别的：如果你执行：

```
sage: optimize.fmin_cg?
```

你发现这是一个使用共轭梯度算法来找到最小值的函数。

```
sage: scipy.special.[tab]
```

将会显示所有 SciPy 拥有的特殊功能。花一些时间到处找寻着看看，是一种让你熟悉 SciPy 的好方式。有一件事情是有一点烦人的，那就是若你执行：scipy.:math:[tab]，你将不会看到任何可导入的模块。例如：scipy.:math:[tab]将不会显示任何 Signal 模块的函数，但是：

```
sage: from scipy import signal
```

然后：

```
signal.[tab]
```

将会显示大量信号处理和滤波器设计的函数。 我上面列出的所有函数都可以导入，即使在默认情况下你并不能看到他们。

scipy.integrate

这个模块的程序都与常微分方程数值解和数值积分相关。让我们给出一个关于解常微分方程的实例：假设我们想求解下面常微分方程：

$$x''(t) + ux'(t)(x(t)^2 - 1) + x(t) = 0$$

用：

$$x' = y$$

$$y' = -x + \mu y(1 - x^2)$$

作为系统读取。我们想使用的是$scipy.integrate$内的$odein$。我们想求解这个常微分方程（ODE）, 计算在 0 到 100 的 1000 个点上(y,y′)的值， 使用下面的代码：

```
sage: import scipy
sage: from scipy import integrate
sage: def f_1(y,t):
....:     return[y[1],-y[0]-10*y[1]*(y[0]**2-1)]
sage: def j_1(y,t):
....:     return [ [0, 1.0],[-2.0*10*y[0]*y[1]-1.0,-
10*(y[0]*y[0]-1.0)] ]
sage: x= scipy.arange(0,100,.1)
sage: y=integrate.odeint(f_1,[1,0],x,Dfun=j_1)
```

当然，你若感觉需要，也可通过下面代码绘出结果，:

```
sage: pts = [(x[i],y[i][0]) for i in range(len(x))]
sage: point2d(pts).show()
```

Optimization

最优化模块内的例程都与求根，最小二乘估计，最小化理论有关。<可以写>

Cvxopt

Cvxopt 提供了许多程序，用来解决凸最优化问题，例如线性或二次规划。它也包含了一个不错的稀疏矩阵库，此函数库提供了连接到稀疏矩阵解算器 unfpack（Matlab 使用相同稀疏矩阵解算器）的界面，它还有一个不错的连接到 lapack 的接口。想要获取更多关于 Cvxopt 的细节，请参考文件：http://cvxopt.org/userguide/index.html。

稀疏矩阵被表示为三元组，分别为：非零值，行指数，列指数。到压缩稀疏列格式的转换是在内部完成的。例如，我们可以输入下列矩阵：

$$\begin{pmatrix} 2 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & -1 & -3 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 1 \end{pmatrix}$$

通过：

```
sage: import numpy
sage: from cvxopt.base import spmatrix
sage: from cvxopt.base import matrix as m
sage: from cvxopt import umfpack
sage: Integer=int
sage: V = [2,3, 3,-1,4, 4,-3,1,2, 2, 6,1]
sage: I = [0,1, 0, 2,4, 1, 2,3,4, 2, 1,4]
sage: J = [0,0, 1, 1,1, 2, 2,2,2, 3, 4,4]
sage: A = spmatrix(V,I,J)
```

解方程式 $AX = B$, $B = [1,1,1,1,1]$，我们可以执行：

```
sage: B = numpy.array([1.0]*5)
sage: B.shape=(5,1)
sage: print(B)
[[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 1.]]
sage: print(A)
[ 2.00e+00  3.00e+00     0         0         0    ]
[ 3.00e+00     0      4.00e+00     0      6.00e+00]
[    0     -1.00e+00 -3.00e+00  2.00e+00     0    ]
[    0         0      1.00e+00     0         0    ]
[    0      4.00e+00  2.00e+00     0      1.00e+00]
sage: C=m(B)
sage: umfpack.linsolve(A,C)
```

```
sage: print(C)
[ 5.79e-01]
[-5.26e-02]
[ 1.00e+00]
[ 1.97e+00]
[-7.89e-01]
```

注意：结果后来被存储到 B 中；m(B)是将我们的 NumPy 数组转换成一种 Cvxopt 可识别的格式。可以使用 Cvxopt 自己的矩阵命令直接创建一个 Cvxopt 矩阵，但是我个人认为 NumPy 数组是更好的。同时注意：我们显式的设置 NumPy 数组的形状，是为了更加清晰的表明它是一个列向量。

按顺序执行下列代码，我们可以计算近似最小度(AMD):

```
sage: RealNumber=float
sage: Integer=int
sage: from cvxopt.base import spmatrix
sage: from cvxopt import amd
sage: A=spmatrix([10,3,5,-2,5,2],[0,2,1,2,2,3],[0,0,1,1,2,3])
sage: P=amd.order(A)
sage: print(P)
[ 1]
[ 0]
[ 2]
[ 3]
```

下面是对一个简单的线性规划的例子，我们想要求解：

$$\begin{aligned} \text{minimze} \quad & -4x_1 - 5x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \leq 3 \\ & x_1 + 5x_2 \leq 3 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{aligned}$$

```
sage: RealNumber=float
sage: Integer=int
sage: from cvxopt.base import matrix as m
sage: from cvxopt import solvers
sage: c = m([-4., -5.])
sage: G = m([[2., 1., -1., 0.], [1., 2., 0., -1.]])
sage: h = m([3., 3., 0., 0.])
sage: sol = solvers.lp(c,G,h) #random
     pcost       dcost       gap    pres   dres   k/t
 0: -8.1000e+00 -1.8300e+01  4e+00  0e+00  8e-01  1e+00
 1: -8.8055e+00 -9.4357e+00  2e-01  1e-16  4e-02  3e-02
 2: -8.9981e+00 -9.0049e+00  2e-03  1e-16  5e-04  4e-04
 3: -9.0000e+00 -9.0000e+00  2e-05  3e-16  5e-06  4e-06
 4: -9.0000e+00 -9.0000e+00  2e-07  1e-16  5e-08  4e-08
```

```
sage: print sol['x']       # 下面之所以得到-00 和+00 是架构的原因
[ 1.00e...00]
[ 1.00e+00]
```

交互式的使用编译代码

这部分是关于在 Sage 中使用编译代码的。然而由于 Sage 绝大部分是建立在 Python 之上的，总体上来说编译代码对 Python 是有效的。使用 Sage 的与 f2py 更方便交互式地工作的界面时例外。你需要查看 f2py 网站来获取更多的使用命令行 f2py 工具的信息。ctypes 和 weave 例子可以运行在最新的 Python 版本上。（weave 并不是 Python 的一部分，必须单独安装）。如果你是用 Sage， Weave，Ctypes，和 f2py 都已经存在了。

批注 [by4]: The exception is that these notes assume you are using Sage's interface to f2py which makes it more convenient to work with f2py interactively.

首先为什么我们想使用编译代码呢？很显然，因为快，远远快于解释性语言 Python。Sage 有一个强大工具，它允许一个交互式的编译代码的调用，编译代码可以是由 C 或 Fortran 编写的。事实上，有 2-4 种方法，这取决于你要完成什么。一种方式是使用 Cython。Cython 是一种 C 和 Python 的混合语言，它基于 Pyrex。它拥有调用外部共享对象库的能力，并且对编写 Python 扩展模块非常有用。在 Sage 文档的其他地方也涵盖了 Cython/Pyrex 的一些细节。

假设你真的仅仅想编写 Python 代码，但是在特殊情况下，有一些十分耗时的代码块你可能希望使用 C/Fortran 编写或者简单的调用外部共享库去完成。在这种情况下，你有三种不同的选择，他们有不同的优缺点。

注意：在你尝试使用编译代码加速你遇到的瓶颈时，首先确保没有更简单的方法。尤其是，首先尝试矢量化，将你的算法运算使用在向量或 NumPy 数组上表达。这些算术运算都是直接在 C 中运行，所以将非常快。如果你的问题不适合以矢量形式表示，继续读下去。

在开始之前要我们注意，关于我们讨论的任何项目，这绝不是一个完整的介绍。能使你确定什么是可能的和使用不同的选项会有什么感觉是更有意义的。

f2py

F2py 是一个非常不错的包，它自动对 Fortran 代码打包使其可被 Python 调用。斐波那契的例子是来自 f2py 网站 http://cens.ioc.ee/projects/f2py2e/。

在 notebook 中，魔法般的 %fortran 将会在一次调用中自动编译任何 Fortran 代码，任何子程序都将会变得可调用（尽管名字都会变成小写）。作为一个例子，粘贴下面代码到一次调用中。正确的间距很重要，默认情况下，这些代码将被视为固定格式的 Fortran，如果有东西不在它该在的列，那编译器又该抱怨了。为了避免这种情况，可以编写 Fortran 90 代码，将首行更改为!f90。这之后将会是一个例子：

批注 [by5]: To avoid this, you can write fortran 90 code instead by making your first line !f90.

```
%fortran
C FILE: FIB1.F
```

```
      SUBROUTINE FIB(A,N)
C
C     CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB1.F
```

现在执行它。它将自动编译然后导入到 Sage 中（尽管他的名字会被转换成小写）。现在我们尝试调用它，我们需要以某种方式将其传递给一个数组 A，和数组的长度 N。他的工作方式是 NumPy 数组会自动转换成 Fortran 数组，Python 的标量转换成 Fortran 的标量。为了调用 fib，我们执行如下操作：

```
import numpy
m=numpy.array([0]*10,dtype=float)
print(m)
fib(m,10)
print(m)
```

注意：Fortren 是一个可以将任何字符串视为 Fortran 语言的函数。所以，如果你有一个 fortran 程序在文件 my_prog.f 中，那你可以执行如下：

```
import numpy
m=numpy.array([0]*10,dtype=float)
print(m)
fib(m,10)
print(m)
```

现在所有的 my_prog.f 中的程度都可被调用了。

在你的 Fortren 代码中可能也需要调用外部函数库。仅需要简单的告诉 f2py 将它们连接进来就好了。例如，你想使用 lapack（一个线性代数函数库）解一个线性方程。我们想要使用的函数是叫 dgesv，它的 signed 如下：

```
SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )


*  N       (input) INTEGER
*          The number of linear equations, i.e., the order of
```

```
*          the matrix A.  N >= 0.
*
*  NRHS    (input) INTEGER
*          The number of right hand sides, i.e., the number of
*          columns of the matrix B.  NRHS >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension
*          (LDA,N) On entry, the N-by-N coefficient matrix A.
*          On exit, the factors L and U from the factorization
*          A = P*L*U; the unit diagonal elements of L are not
*          stored.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >=
*          max(1,N).
*
*  IPIV    (output) INTEGER array, dimension (N)
*        The pivot indices that define the permutation matrix P;
*        row i of the matrix was interchanged with row IPIV(i).
*
*  B       (input/output) DOUBLE PRECISION array, dimension
*          (LDB,NRHS)
*          On entry, the N-by-NRHS matrix of right hand side
*          matrix B.
*         On exit, if INFO = 0, the N-by-NRHS solution matrix X.
*
*  LDB     (input) INTEGER
*   The leading dimension of the array B.  LDB >= max(1,N).
*
* INFO  (output) INTEGER
*   = 0:  successful exit
*   < 0:  if INFO = -i, the i-th argument had an illegal value
*   >0:  if INFO = i, U(i,i) is exactly zero.  The factorization
*           has been completed, but the factor U is exactly
*           singular, so the solution could not be computed.
```

我们可以如下做。注意，列表中的顺序实际上很重要，因为是按照他们的顺序传递给 gcc。Fortran.libraries 是一个已被链入的简单的名单库。你可以直接设置这个列表。所以 $fortran.libraries = ['lapack','blas']$等价于如下形式：

```
fortran.add_library('lapack')
fortran.add_library('blas')
```

现在

```
%fortran
!f90
Subroutine LinearEquations(A,b,n)
Integer n
```

```
Real*8 A(n,n), b(n)
Integer i, j, pivot(n), ok
call DGESV(n, 1, A, n, pivot, b, n, ok)
end
```

关于以上有数个地方需要指出。我们之前强调过，如果首行代码是!f90，它将被以 Fortren 90 代码对待，而不需要作为固定的格式。为了使用上面的代码，尝试：

```
a=numpy.random.randn(10,10)
b=numpy.array(range(10),dtype=float)
x=b.copy()
linearequations(a,x,10)
numpy.dot(a,x)
```

他将解出线性系统$\boldsymbol{Ax} = \boldsymbol{b}$的结果，并存储在$\boldsymbol{b}$中。如果你的函数库不在 Sage 的 local/lib 或 PATH 中，可以通过下面的代码添加到搜索路径：

```
fortran.add_library_path('path').
```

可以直接通过任务设置 fortran.library 路径。它应该是以一个路径的列表(字符串)传递给 gcc。为了给你一个 f2py 可以完成有更多事情使用看法，指出，使用*intent*()语句可以的控制 Python 语句产生结果的方式，尽管有些麻烦。 例如考虑下列根据原 Fibonacci 代码的修改：

批注 [by6]: note that using intent statements you can control the way the resulting Python function behaves a bit bitter

```
C FILE: FIB3.F
      SUBROUTINE FIB(A,N)
C
C     CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
Cf2py depend(n) a
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB3.F
```

注：说明一下*intent*()语句，它告诉 f2py ，*n* 是一个输入参数，*a* 是输出。它可以如下调用：

```
a=fib(10)
```

通常，你可能将所有输入声明使用 intent(in)传输给 Fortran 函数，所有输出显式的使用 intent(out)返回至一个元组。注：直接指明使用 intent(in)将输入传给函数意味着在函数调用之前仅仅关心它们的值。所以，在上面我们需要指定为输入的 n 告诉我们有多少斐波那契数需要计算，但是并不需要取回 n，因为它并不包含任何新的内容。相似的，A 是 intent(out)指定的输出，所以我们事先并不需要知道 A 具体的值，我们仅仅关心它之后包含的值。f2py 生成一个 Python 函数，所以仅仅需要通过声明 intent(in)，为其它参数提供空的空间，然后他就会返回到 intent(out)声明的输出。intent(in)默认情况下将所有参数包含在内。

批注 [by7]: F2py generates(形成) a Python function so you only pass those declared intent(in) and supplies empty workspaces for the remaining arguments and it only returns those that are intent(out).

现在考虑下面：

```
%fortran
        Subroutine Rescale(a,b,n)
        Implicit none
        Integer n,i,j
        Real*8 a(n,n), b
        do i = 1,n
           do j=1,n
             a(i,j)=b*a(i,j)
           end do
        end do
        end
```

你或许希望 Rescale(a,n)可重新调整 NumPy 矩阵 a 的尺寸。唉，它并不会这样做。所有你的输入再后来都不会改变。注意上面斐波那契的例子，一个一维数组被 Fortran 代码改变了，同样，当我们调用 lapack 一维向量 b 时被这个例子的结果所更新，然而，尽管 dgesv 表明已修改了输入矩阵，矩阵 A 却仍没有改变。这为什么不会发生在二维数组中呢？想要明白这个问题需要知道 C 和 Fortran 使如何存储数组的。Fortran 存储数组是列为主序，而 C 是以行为主序的。下个矩阵

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

被存储为：

(0 1 2 3 4 5)in C

(0 3 1 4 2 5)in Fortran

一位数组的在 C 和 Fortran 中的存储结构是相同的。因为 f2py 允许 Fortran 代码在合适的情况下操作一位数组，所以 Fortran 代码将会改变传递给它的一维 NumPy 数组。然而，二维数组在默认情况下就不相同了，f2py 拷贝 NumPy 数组（它被存储为 C 格式）为另一个 Fortran 格式的数组（例如通过转置）再传输给 Fortran 函数。我们将一种看到绕过这种复制的方法。首先我们指出 rescale 函数的一种写法：

```
%fortran

        Subroutine Rescale(a,b,n)
        Implicit none
```

```
        Integer n,i,j
        Real*8 a(n,n), b
Cf2py intent(in,out) a
        do i = 1,n
           do j=1,n
             a(i,j)=b*a(i,j)
           end do
        end do
        end
```

注意：调用它你需要使用：

```
b=rescale(a,2.0).
```

这里我没有传入 a 的维数 n。通常，f2py 会计算出它。正好提一下：f2py 可以自动为 Python 版本的函数生成一些文档，所以你可以检查函数需要的输入和最终的返回。使用下面命令：

```
rescale?
```

intent(in,out)指令告诉 f2py 在子程序的最后获取 *a* 的内容，然后将它以 NumPy 数组形式返回。这或许仍不是你想要的。最开始传入的 a 是不可修改的。如果你想修改最开始的 a，就要使用 intent(inout).。它基本上允许 Fortran 代码在原地处理数据：

```
%fortran

        Subroutine Rescale(a,b,n)
        Implicit none
        Integer n,i,j
        Real*8 a(n,n), b
Cf2py intent(inout) a
        do i = 1,n
           do j=1,n
             a(i,j)=b*a(i,j)
           end do
        end do
        end
```

如果你想使用 fortran 代码以原地模式处理 numpy 数组，你需要确保你的 NumPy 数组被存储为 Fortran 格式。你可以通过$order = 'FORTRAN'$ 在创建数组时确认。如下：

```
a=numpy.array([[1,2],[3,4]],dtype=float,order='FORTRAN')
rescale(a,2.0)
```

执行之后，我们所希望的新版本就产生了。这便是结合了上面两个的最终版：

```
%fortran

        Subroutine Rescale(a,b,n)
        Implicit none
        Integer n,i,j
```

```
        Real*8 a(n,n), b
Cf2py intent(in,out,overwrite) a
        do i = 1,n
           do j=1,n
             a(i,j)=b*a(i,j)
           end do
        end do
        end
```

intent (in,out,overwite) 说明如果 *a* 是按照 Fortran 格式存储的，我们按照原地模式处理，然而如果不是的话，就创建它的以 Fortran 格式存储的副本，并在之后返回它 。这是一种两全其美的方法。注意：你若想重复传输一个很大的数组给 Fortran 代码，避免使用 (inout)或(in,out,overwrite)复制是十分重要的。尽管你可以使用 Fortran 格式顺序的 NumPy 数组来避免复制，你也应该记住。

批注 [by8]: however if its not we copy it and return the contents afterwards.

批注 [by9]: Remember though that your numpy array must use Fortran ordering to avoid the copying.

想要获取更多的 F2py 的实例和高级使用技巧你需要参考 f2py 的官方网站：http://cens.ioc.ee/projects/f2py2e/。获取 f2py 命令行工具的文档，可以在 Sage 的 Shell 中使用：

```
!f2py
```

与 f2py 有关的更多有趣的例子

下面让我们看一下使用 f2py 的几个更有趣的例子。 我们将使用一个简单的迭代法来求解拉普拉斯方程。事实上，这个实现来自于 SciPy 的网站：http://www.scipy.org/PerformancePython?highlight=%28performance%29 。里面含有许多使用 Python 实现数值计算的内容。

下面的 Fortran 代码实现了求解拉普拉斯方程的松弛法的一次迭代：

批注 [by10]: a single iteration(迭代) of a relaxation(放松) method

```
%fortran
      subroutine timestep(u,n,m,dx,dy,error)
      double precision u(n,m)
      double precision dx,dy,dx2,dy2,dnr_inv,tmp,diff
      integer n,m,i,j
cf2py intent(in) :: dx,dy
cf2py intent(in,out) :: u
cf2py intent(out) :: error
cf2py intent(hide) :: n,m
      dx2 = dx*dx
      dy2 = dy*dy
      dnr_inv = 0.5d0 / (dx2+dy2)
      error = 0d0
      do 200,j=2,m-1
         do 100,i=2,n-1
            tmp = u(i,j)
            u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2+
     &           (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv
```

```
            diff = u(i,j) - tmp
            error = error + diff*diff
 100     continue
 200  continue
      error = sqrt(error)
      end
```

你若执行

```
timestep?
```

你发现需要传给 timestep 一个 numpy 数组和网格空间 dx,dy 然后它将返回更新了的 u，和一个误差估计值。把这个应用到实际问题的解决中，使用程序代码：

```
import numpy
j=numpy.complex(0,1)
num_points=50
u=numpy.zeros((num_points,num_points),dtype=float)
pi_c=float(pi)
x=numpy.r_[0.0:pi_c:num_points*j]
u[0,:]=numpy.sin(x)
u[num_points-1,:]=numpy.sin(x)
def solve_laplace(u,dx,dy):
   iter =0
   err = 2
   while(iter <10000 and err>1e-6):
      (u,err)=timestep(u,dx,dy)
      iter+=1
   return (u,err,iter)
```

现在调用程序：

```
(sol,err,iter)=solve_laplace(u,pi_c/
                             (num_points-1),pi_c/(num_points-1))
```

这个方程的解带有左右边界都是半正弦振动的边界条件。就以我们使用的 51X51 的网格为例，我发现它用来求解这 2750 次迭代需要花费 0.2s。如果你安装了 gnuplot()包（使用包管理工具 packages()搜索其名字，并安装它），你可以绘出：

```
import Gnuplot
g=Gnuplot.Gnuplot(persist=1)
g('set parametric')
g('set data style lines')
g('set hidden')
g('set contour base')
g('set zrange [-.2:1.2]')
data=Gnuplot.GridData(sol,x,x,binary=0)
g.splot(data)
```

为了查看我们已使用 f2py 生成的效果，让我们比较使用纯 Python 和使用 NumPy 数组的矢量化版本实现相同的算法。

批注 [by11]: To see what we have gained by using f2py let us compare the same algorithm in pure python and a vectorized version using numpy arrays.

```
def slowTimeStep(u,dx,dy):
    """直接使用 Python 循环需要一个时间步长"""
    nx, ny = u.shape
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)


    err = 0.0
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            tmp = u[i,j]
            u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                      (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
            diff = u[i,j] - tmp
            err += diff*diff

    return u,numpy.sqrt(err)

def numpyTimeStep(u,dx,dy):
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u_old=u.copy()
    # 实际的迭代
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
    v = (u - u_old).flat
    return u,numpy.sqrt(numpy.dot(v,v))
```

你可以尝试改变使用在我们的程序当中的 timestep 函数。Python 版本即便是在 50x50 网格上都很慢。他花费了 70s 来求解 3000 次迭代。使用 NumPy 程序进行了 5000 次迭代花费 2s 就达到了要求精度。作为对比，f2py 花费了 0.2s 进行了 3000 次迭代，就达到了相应的精度。需要指出 numpy 程序使用的是不同的算法，它是 Jacobi 而 f2py 实现的算法是 Gauss-Seidel。 这是 NumPy 花费更多次迭代的原因。即便如此，也可以看到 f2py 版本表现出了 numpy 版本五倍的性能。事实上，如果尝试在 500x500 网格上，我发现 NumPy 程序进行 500 次迭代耗时为 30s，而 f2py 版本仅仅耗时 2s。f2py 更是得到了 15 倍于 NumPy 程序的性能。每个实际的较小网格上的迭代是花费相对较低的，所以调用的 f2py 开销会显得明显。而在处理大型耗时的迭代时，f2py 的优势显而易见的。即便如此，在有的实例中它仍然会更快。注： 500x500 网格上做 500 次迭代使用 Python 将消耗 30min。

据我所知，在 Matlab 中实现该算法和我们刚刚在 NumPy 矢量化的情形类似。NumPy 是不亚于 Matlab 中的矢量组件的。所以除非还有我不知道的窍门，使用 f2py 你可以交互式的编写出比在 Matlab 中实现任何代码都快 15 倍的性能（如有错误，请您指正）。 可以使 f2py 版本比使用 intent(in,out,overwrite) 更快一些，那就是在定义 NumPy 数组时使用 order='FORTRAN'。 这消除了在内存中不必要的复制。

Weave

Weave 是为 C/C++而创建的工具，而 f2py 是为 Fortran 而创建的工具 (尽管我们知道使用 f2py 也可以封装 C 代码)。假如我们想要将一些数据存储到 NumPy 数组中，我们想编写一些 C/C++ 来快速的处理这些数据。对一个一般的例子，让我们写一个函数，算出 numpy 数组中内容的和：

```
sage: from scipy import weave
sage: from scipy.weave import converters
```

```
def my_sum(a):
    n=int(len(a))
    code="""
    int i;
    long int counter;
    counter =0;
    for(i=0;i<n;i++)
    {
        counter=counter+a(i);
    }
    return_val=counter;
    """


err=weave.inline(code,['a','n'],type_converters=converters.blitz
,compiler='gcc')
    return err
```

为了调用这个函数：

```
import numpy
a = numpy.array(range(60000))
time my_sum(a)
time sum(range(60000))
```

第一次 weave 代码执行这些代码时要编译，从次以后，程序就立即执行。你可以查找 Python 内置的求和函数和我们刚刚编写的求和函数在速度上的对比。我们解释一下刚才写的程序。正如你所看到的，要使用 weave 亦需要创建一个包含纯 C/C++ 代码的字符串。然后你可以在其上调用 weave.inline。把这段带有代码的字符串传递给 weave，同时将一个 Python 的 list 对象转换成 C 变量。在例子中我们可以在 weave 中引用 Python 对象 a 和 n。Numpy 数组通过$\boldsymbol{a(i)}$访问一维向量的元素，或者是使用$\boldsymbol{a(i,j)}$ 访问二维数组的元素。当然我们不会仅仅使用 python 对象，一般 weave 可用 python 所有的数据类型，例如 ints，floats 和 NumPy 数组。注意：numpy 数组不能变成 C 中的指针，这就是为什么他们是由 ()访问,而不是[]。如果你需要一个指针，那你应该将数据复制到一个指针。下面是一个有些难懂的例子：调用 lapack 求解线性系统 $\boldsymbol{ax = b}$：

```
def weave_solve(a,b):
    n = len(a[0])
    x = numpy.array([0]*n,dtype=float)

    support_code="""
    #include <stdio.h>
    extern "C" {
    void dgesv_(int *size, int *flag,double* data,int*size,
                int*perm,double*vec,int*size,int*ok);
    }
    """

    code="""
        int i,j;
        double* a_c;
        double* b_c;
        int size;
        int flag;
        int* p;
        int ok;
        size=n;
        flag=1;
        a_c= (double *)malloc(sizeof(double)*n*n);
        b_c= (double *)malloc(sizeof(double)*n);
        p = (int*)malloc(sizeof(int)*n);
        for(i=0;i<n;i++)
           {
           b_c[i]=b(i);
           for(j=0;j<n;j++)
             a_c[i*n+j]=a(i,j);
           }
        dgesv_(&size,&flag,a_c,&size,p,b_c,&size,&ok);
        for(i=0;i<n;i++)
           x(i)=b_c[i];
        free(a_c);
        free(b_c);
        free(p);
    """

    libs=['lapack','blas','g2c']
    dirs=['/media/sdb1/sage-2.6.linux32bit-i686-Linux']
    vars = ['a','b','x','n']
    weave.inline(code,vars,support_code=support_code,\
                  libraries=libs,library_dirs=dirs, \
                 type_converters=converters.blitz,compiler='gcc')
    return x
```

注意，我们必须使用 support_code 参数，它是我们附加在 C 代码的开头用来 include 头文件和声明函数的。注意：inline 也可以把所有我们这里用到的 distutils 编译器选项，连接到 lapack。

```
def weaveTimeStep(u,dx,dy):
    """Takes a time step using inlined C code -- this version
usesblitz arrays."""
    nx, ny = u.shape
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)

    code = """
           double tmp, err, diff,dnr_inv_;
           dnr_inv_=dnr_inv;
           err = 0.0;
           for (int i=1; i<nx-1; ++i) {
               for (int j=1; j<ny-1; ++j) {
                   tmp = u(i,j);
                   u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +\
                             (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv_;
                   diff = u(i,j) - tmp;
                   err += diff*diff;
               }
           }
           return_val = sqrt(err);
           """
    # 在 windows 安装 MSVC 后编译
    err = weave.inline(code, ['u', 'dx2', 'dy2', 'dnr_inv', \
                                    'nx','ny'],
                       type_converters = converters.blitz,
                       compiler = 'gcc')
    return u,err
```

使用我们之前的程序，你会发现这个版本花费相同的时间，f2py 版本做 2750 次迭代需要花费 0.2s。

获取关于 weave 的更多信息，请查看： http://www.scipy.org/Weave

Ctypes

Ctypes 是一个非常有趣的 python 包，它允许你导入共享对象函数库，并直接调用它们。尽管它被称为 ctypes，它也可以被用作从共享对象函数库中调用使用 Fortran 编写的函数。你需要知道唯一复杂点的东西就是 fortran 函数什么看起来和 C 相似。都是指针时是简单的，所以如果你的 Fortran 函数被作为 foo(A,N) 调用，A 是一个数组，而 N 是它的长度，然后在 C 中调用它，它接受一个指向 double 类型数组的指针和指向 int 类型的指针。另一件事情需要明确，C 和 Fortran 函数经常有强调附加。这就是说，一个 Fortran 函数 foo

批注 [by12]: The other thing to be aware of is that from C, fortran functions usually have an underscore appended.

可以在 C 中以 foo 出现（这是一般情形，也由编译器决定）。就说到这里，下面是一个 C 的示例。

作为一个例子，假设你编写以下简单的 C 程序：

```
#include <stdio.h>

int sum(double *x,int n)
{
  int i;
  double counter;
  counter = 0;
  for(i=0;i<n;i++)
    {
      counter=counter+x[i];

    }
  return counter;
}
```

你想要在 python 中调用，首先通过执行下面代码创建共享对象函数库。

```
gcc -c sum.c
gcc -shared -o sum.so sum.o
```

注意：在 OSX 上 –shared 需要被替换为 -dynamiclib ， sum.so 要被替换为 sum.dylib 。然后我们执行：

```
from ctypes import *
my_sum=CDLL('sum.so')
a=numpy.array(range(10),dtype=float)
my_sum.sum(a.ctypes.data_as(c_void_p),int(10))
```

注意：这里的 a.ctypes.data_as(c_void_p)返回一个 ctypes 对象，这个对象是一个指向底层数组 a 的指针。注意：尽管 sum 接受一个 double*类型，但只要我们的指针指向了正确的数据，不论是数据类型是什么，都会自动计算。

注意：实际上还有另一种方式可以传递所需的 double 型的数组。例如：

```
a=(c_double*10)()
for i in range(10):
   a[i]=i
my_sum.sum(a,int(10))
```

这个例子仅在 ctypes 中使用。 Ctypes 有对 C 数据类型的封装器。例如：

```
a=c_double(10.4)
```

将会创建一个可以传递给 C 函数的 ctypes double 对象。注意：有一个 byref 函数允许你通过引用来传递参数。在下一个例子 example. c 中 double*10 是一个 python 对象，它代表这个数组中含有 10 个 double 类型的数：

```
a=(c_double*10)()
```

令 a 等于一个有 10 个 double 类型数的数组。我发现这种方法通常不如使用 NumPy 数组类型有用，对更精确的数，使用 NumPy 数组可以更好的整合进 Python 和 Sage 中。

这里有一个使用 ctypes 直接调用 lapack 的例子。注意，程序仅在你的操作系统上已经安装了 lapack 共享对象函数库是才会运行。在 linux 这个文件将会是 liblapack.so ，你可能会使用 dgesv (OSX 使用 CLAPACK 因此缺乏下划线)。

批注 [by13]: OSX use CLAPACK hence the lack of the underscore

```
from ctypes import *
def ctypes_solve(m,b,n):
    a=CDLL('/usr/lib/liblapack.dylib')
    import numpy
    p=(c_int*n)()
    size=c_int(n)
    ones=c_int(1)
    ok=c_int(0)
    a.dgesv(byref(size),byref(ones),m.ctypes.data_as(c_void_p),\
                byref(size),p,b.ctypes.data_as(c_void_p),\
                  byref(size),byref(ok))
```

从完整性出发，让我们考虑一种使用 C 求解拉普拉斯方程的方法。假设你已在 C 上写了一个简单的求解程序，你想在 Python 中调用它，这样你便可以轻易地测试不同的边界条件了。你的 C 代码或许是这样的：

```
#include <math.h>
#include <stdio.h>

double timestep(double *u,int nx,int ny,double dx,double dy)
{
  double tmp, err, diff,dx2,dy2,dnr_inv;
  dx2=dx*dx;
  dy2=dy*dy;
  dnr_inv=0.5/(dx2+dy2);
  err = 0.0;
  int i,j;

for (i=1; i<nx-1; ++i) {
  for (j=1; j<ny-1; ++j) {
    tmp = u[i*nx+j];
    u[i*nx+j] = ((u[(i-1)*nx+j] + u[(i+1)*nx+j])*dy2 +
          (u[i*nx+j-1] + u[i*nx+j+1])*dx2)*dnr_inv;
    diff = u[i*nx+j] - tmp;
    err += diff*diff;
  }
}

 return sqrt(err);
}
```

```
double solve_in_C(double *u,int nx,int ny,double dx,double dy)
{
  double err;
  int iter;
  iter = 0;
  err = 1;
    while(iter <10000 && err > 1e-6)
      {
    err=timestep(u,nx,ny,dx,dy);
    iter++;
      }

  return err;
}
```

你可以通过下面的命令编译它：

```
gcc -c laplace.c
gcc -shared -o laplace.so laplace.o
```

现在，在 Sage（命令行或 notebook）中执行：

```
from ctypes import *
laplace=CDLL('/home/jkantor/laplace.so')
laplace.timestep.restype=c_double
laplace.solve_in_C.restype=c_double
import numpy
u=numpy.zeros((51,51),dtype=float)
pi_c=float(pi)
x=numpy.arange(0,pi_c+pi_c/50,pi_c/50,dtype=float)
u[0,:]=numpy.sin(x)
u[50,:]=numpy.sin(x)

def solve(u):
  iter =0
  err = 2
  n=c_int(int(51))
  pi_c=float(pi/50)
  dx=c_double(pi_c)
  while(iter <5000 and err>1e-6):
     err=laplace.timestep(u.ctypes.data_as(c_void_p),n,n,dx,dx)
     iter+=1
     if(iter %50==0):
        print((err,iter))
  return (u,err,iter)
```

注意这一行：laplace.timestep.restype=c_double。默认情况下 ctypes 假设返回值是 ints。若这不是你想要的，通过设置 restype 来确定返回值类型。若你执行上面的代码，然后

solve(u) 将会解出这个系统。使用 weave 和 fortran 来求解是性能相当的，都花费 0.2s。你也可以这样做：

```
n=c_int(int(51))
dx=c_double(float(pi/50))
laplace.solve_in_C(n.ctypes.data_as(c_void_p),n,n,dx,dx)
```

整个计算过程都在 C 中进行。这是十分快的。毫无疑问，我们可以有我们的 Fortran 程序或 weave 程序来在 C/Fortran 层完成整个计算过程，也会有相同的速度。

我之前说过，可以简单地使用 ctypes 调用一个用 Fortran 编写的共享对象库。关键是他必须是共享对象函数库，并且所有的 Fortran 参数都通过引用传递，可以是指针或使用 byref。当然，即使我们使用非常简单的数据类型，都有可能解决更加复杂的 C 结构。在这方面，获取更多关于 ctypes 可参见： http://python.net/crew/theller/ctypes/

更加复杂的 ctypes 实例

这里我们将可以看到更加复杂的 ctype 的示例。首先考虑下面 C 代码：

```
#include <stdio.h>
#include <stdlib.h>

struct double_row_element_t {
  double value;
  int  col_index;
  struct double_row_element_t * next_element;
};

typedef struct double_row_element_t double_row_element;

typedef struct  {
  int nrows;
  int ncols;
  int nnz;
  double_row_element** rows;
} double_sparse_matrix;




double_sparse_matrix * initialize_matrix(int nrows, int ncols)
{
  int i;
  double_sparse_matrix* new_matrix;
  new_matrix = (double_sparse_matrix *)\
malloc(sizeof(double_sparse_matrix));
  new_matrix->rows= (double_row_element **)\
```

```
malloc(sizeof(double_row_element *)*nrows);
  for(i=0;i<nrows;i++)
    {
      (new_matrix->rows)[i]=(double_row_element *)\
malloc(sizeof(double_row_element));
      (new_matrix->rows)[i]->value=0;
      (new_matrix->rows)[i]->col_index=0;
      (new_matrix->rows)[i]->next_element = 0;
    }
  new_matrix->nrows=nrows;
  new_matrix->ncols=ncols;
  new_matrix->nnz=0;
  return new_matrix;
}


int free_matrix(double_sparse_matrix * matrix)
{
  int i;
  double_row_element* next_element;
  double_row_element* current_element;
  for(i=0;i<matrix->nrows;i++)
    {
      current_element = (matrix->rows)[i];
      while(current_element->next_element!=0)
    {
      next_element=current_element->next_element;
      free(current_element);
      current_element=next_element;
    }
      free(current_element);
    }
  free(matrix->rows);
  free(matrix);
  return 1;
}

int set_value(double_sparse_matrix * matrix,int row, int col,
double value)
{

  int i;
  i=0;
  double_row_element* current_element;
  double_row_element* new_element;

  if(row> matrix->nrows||col > matrix->ncols||row <0||col <0)
    return 1;
```

```
  current_element = (matrix->rows)[row];
  while(1)
    {
      if(current_element->col_index==col)
    {
    current_element->value=value;
    return 0;
    }

      else
    if(current_element->next_element!=0)
    {
          if(current_element->next_element->col_index <=col)
        current_element = current_element->next_element;
      else
        if(current_element->next_element->col_index > col)
          {
        new_element = (double_row_element *) \
malloc(sizeof(double_row_element));
        new_element->value=value;
        new_element->col_index=col;
        new_element->next_element=current_element->next_element;
        current_element->next_element=new_element;
        return 0;
          }
    }
    else
      {
        new_element = (double_row_element *) \
malloc(sizeof(double_row_element));
        new_element->value=value;
        new_element->col_index=col;
        new_element->next_element=0;
        current_element->next_element=new_element;
        break;
      }

    }

  return 0;
}


double get_value(double_sparse_matrix* matrix,int row, int col)
{
  int i;
  double_row_element * current_element;
```

```
  if(row> matrix->nrows||col > matrix->ncols||row <0 || col <0)
    return 0.0;

  current_element = (matrix->rows)[row];
  while(1)
    {
      if(current_element->col_index==col)
    {
      return current_element->value;
    }
      else
    {
      if(current_element->col_index<col &&
current_element->next_element !=0)
        current_element=current_element->next_element;
      else
        if(current_element->col_index >col
                || current_element ->next_element==0)
           return 0;
    }
    }

}
```

把它放入 linked_list_sparse.c 文件中，然后编译：

```
gcc -c linked_list_sparse.c
gcc -shared -o linked_list_sparse.so linked_list_sparse.o
```

接下来考虑紧跟着的 python 辅助代码：

```
from ctypes import *

class double_row_element(Structure):
    pass

double_row_element._fields_=[("value",c_double),("col_index",c_int),
                             ("next_element",POINTER(double_row_element) )]


class double_sparse_matrix(Structure):
    _fields_=[("nrows",c_int),("ncols",c_int),("nnz",c_int),
                             ("rows",POINTER(POINTER(double_row_element)))]


double_sparse_pointer=POINTER(double_sparse_matrix)
sparse_library=CDLL("/home/jkantor/linked_list_sparse.so")
```

```
initialize_matrix=sparse_library.initialize_matrix
initialize_matrix.restype=double_sparse_pointer
set_value=sparse_library.set_value
get_value=sparse_library.get_value
get_value.restype=c_double
free_matrix=sparse_library.free_matrix
```

我们讨论一下上面的代码。原始的 C 代码被作为一个链表而存储进一个稀疏矩阵。Python 代码使用 ctptes 的结构类创建反映 C 代码结构的结构类。为了创建反映 C 结构的 Python 对象，简单的创建源于 Structure 的类。类的_fields_ 属性必须被设置为字段名称和值的元组所组成的列表。注意：当你需要在完全定义（作为链表）这个结构之前使用它，你可以先试用"Pass"，然后在之后指定填充内容，就像上面的例子那样。同样需要注意的是用于创建一个指向任意 ctypes 类型的指针 POINTER 操作符。我们可以按如下方式直接调用我们的库。

批注 [by14]: Also note the POINTER operator which creates a pointer out of any ctypes type

```
m=double_sparse_pointer()
m=initialize_matrix(c_int(10),c_int(10))
set_value(m,c_int(4),c_int(4),c_double(5.0))
a=get_value(m,c_int(4),c_int(4))
print("%f"%a)
free_matrix(m)
```

你可以仅仅通过 (*struct_object*).*field name*访问一个结构的具体内容。然而对指针而言，有一个内容属性。因此，在上面的*m.contents.nrows* 会让您访问 nrows 字段。事实上你可以手动地沿着链表设置，如下：

批注 [by15]: In fact you can manually(手动地) walk along the linked list as follows.

```
m=double_sparse_pointer()
m=initialize_matrix(c_int(10),c_int(10))
set_value(m,c_int(4),c_int(4),c_double(5.0))
a=m.contents.rows[4]
b=a.contents.next_element
b.contents.value
free_matrix(m)
```

比较 Cython 和 Pyrex

编写 Cython 和 Pyrex 的计算程序是完全可以的。从下面的网站你可以找到一个例子，链接： http://www.scipy.org/PerformancePython?highlight=%28performance%29 。在之前的解决方案中，一个潜在的关于 Cython 的缺点就是，需要用户熟练应用 NumPy 数组和 Sage 矩阵才能访问他们内部数据。与之相成对比，weave，Scipy 和 ctypes 的例子只需要知道 C 或 Fortran，从他们的角度看，NumPy 数据神奇地传递到 C 或 Fortran 而不需要没有进一步的思考。对于 Pyrex，为了作为一种交互编写代码的方式对比，访问 NumPy 数组和 Sage 矩阵的内部结构的工作需要隐藏。

并行计算

mpi4py

MPI 代表 message passing interface ，是并行程序设计的通用函数库。mpi4py 包是建立在 mpi 之上的，它允许任意的 Python 对象在不同的进程之间相互传递。Sage 在默认情况下不会安装这个包，要安装它可以执行：

```
sage: optional_packages()
```

寻找包名 openmpi-* 和 mpi4py-*然后执行：

```
sage: install_package('openmpi-*')
sage: install_package('mpi4py-*')
```

注意：openmpi 需要花费一些时间编译 (15-20 分钟左右)。 Openmpi 可以在集群上运行，但是这需要一些设置，才能在使进程在不同的机器上传递信息（尽管当你使用集群时，可能早已设置完毕）。最简单的例子是,如果你在一个共享内存或多核系统上 ，此时 openmpi 不需要你配置就可以工作。说实话，我也没有在集群当中使用过 mpi4py，尽管网络上有很多关于这个信息的话题。

现在，mpi 的工作方式是开启一组 mpi 进程，所有的进程运行都相同的代码。每个进程都拥有一个 rank，这是一个可用来识别它的数。下面的伪代码表示 MPI 编程的通常格式：

```
   ....

if my rank is n:
   do some somputation ...
   send some stuff to the process of rank j
   receive some data from the process of rank k

else if my rank is n+1:
   ....
```

每个进程寻找它应该做什么（通过 rank 来指定），进程之间可以发送和接收数据。举个例子。创建一个有下面代码的脚本 mpi_1.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("hello world")
print("my rank is: %d"%comm.rank)
```

若要执行它，你可以（在你的 Sage 文件夹下使用命令行）

```
./local/bin/mpirun -np 5 ./sage -python mpi_1.py
```

命令 *mpirun* $-np$ 5 在 mpi 中开启了这份程序的 5 个副本。在这种情况下，我们有 5 个纯 Python 风格的 Sage 副本运行脚本 mpi_1.py.。这个结果应该是五个 “hello worlds” 加上

五个进程的 rank。两个最重要的 mpi 操作就是接受和发送。考虑下面这个例子， 将下面代码放入脚本 mpi_2.py 中。

```
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank=comm.rank
size=comm.size
v=numpy.array([rank]*5,dtype=float)
comm.send(v,dest=(rank+1)%size)
data=comm.recv(source=(rank-1)%size)
print("my rank is %d"%rank)
print("I received this:")
print(data)
```

使用相同的命令，只是将上面的 mpi_1.py 替换为 mpi_2.py，将会产生五个输出，然后你会看到每个进程创建了一个数组，然后将它传输给下一个进程（最后一个传递给第一个）。注意：MPI.size 是 mpi 的进程数。 MPI.COMM_WORLD 是通信天地。

批注 [by16]: MPI.COMM_WORLD is the communication world.

关于 MPI 有一些精妙之处需要注意。较小的发送将被存至缓冲区。这意味着一个进程发送的较小的对象将被存储在 openmpi，然后这个程序继续执行，无论目标进程在何时执行接受操作，这个发送对象的都将被接受。当然对于一个大的对象，进程将会挂起，直到目标程序执行接受操作。事实上，上面的代码中若将[rank]*5 替换为[rank]*500，进程将会挂起。执行下面操作将会更好：

```
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank=comm.rank
size=comm.size
v=numpy.array([rank]*500,dtype=float)
if comm.rank==0:
   comm.send(v,dest=(rank+1)%size)
if comm.rank > 0:
    data=comm.recv(source=(rank-1)%size)
    comm.send(v,dest=(rank+1)%size)
if comm.rank==0:
    data=comm.recv(source=size-1)

print("my rank is %d"%rank)
print("I received this:")
print(data)
```

现在，第一个进程(rank 为 0)开始发送，然后进程 1 将会准备接受，然后它开始发送，在然后进程 2 开始接受，之后再发送，一直这样下去。这样不论我们传输的文件的大小，进程都不会互锁。

通常的习惯，有这样一个进程，一般 rank 为 0，作为引导。这个进程给其他进程传送数据和进程的结果，以及决定以后的计算应该如何继续进行。

考虑下面的代码：

```
from mpi4py import MPI
import numpy
sendbuf=[]
root=0
comm = MPI.COMM_WORLD
if comm.rank==0:
    m=numpy.random.randn(comm.size,comm.size)
    print(m)
    sendbuf=m

v=comm.scatter(sendbuf,root)

print("I got this array:")
print(v)
```

scatter 命令需要一个列表，然后将它平均的分到各个进程之中。这里，在根进程（root）之中创建了一个矩阵（被视为由行向量组成的列表），然后分散到每个进程（roots senbuf 被均分到所有的进程之中）。每个进程打印出它获得行。注意：scatter 命令被每个进程执行，但是只有当 root 执行它时，它起到发送和接收的作用（root 从自身获取一个行），而对其他进程而言只是接收。

这里还有一个 gather 命令，它可以将所有结果收集到列表当中。下一个例子中将使用 scatter 和 gather 。现在 root 分散一个矩阵当中的行，每个进程对它接收到的行的元素做平方。然后这些行向量通过 root 收集起来，将它们重新组合成一个新的矩阵。

```
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
sendbuf=[]
root=0
if comm.rank==0:
    m=numpy.array(range(comm.size*comm.size),dtype=float)
    m.shape=(comm.size,comm.size)
    print(m)
    sendbuf=m

v=comm.scatter(sendbuf,root)
print("I got this array:")
print(v)
v=v*v
recvbuf=comm.gather(v,root)
if comm.rank==0:
    print numpy.array(recvbuf)
```

还有一个广播命令能将一个对象发送到每个进程中。考虑下面这个小扩展。这和前面的相同，但是现在 root 进程在最后发送给每个进程一个要打印出来的字符串 “done”。

```
v=MPI.COMM_WORLD.scatter(sendbuf,root)
print("I got this array:")
print(v)
v=v*v
recvbuf=MPI.COMM_WORLD.gather(v,root)
if MPI.COMM_WORLD.rank==0:
    print numpy.array(recvbuf)

if MPI.COMM_WORLD.rank==0:
    sendbuf="done"
recvbuf=MPI.COMM_WORLD.bcast(sendbuf,root)
print(recvbuf)
```

MPI 编程是有难度的。这是“精神分裂的编程”，编写多线程执行的单个程序那就像“在大脑中涌现出许多声音”。

批注 [by17]: MPI programming is difficult. It is “schizophrenic programming” in that you are writing a single programming with multiple threads of execution “many voices in one head”.

并行的 Laplace 求解程序

下面的代码可并行的在网格上求解 Laplace’s 方程。它将一个正方形分成 n 平行条，其中 n 是进程数然后使用 jacobi 迭代器。代码工作的方式是 root 进程创建一个矩阵，然后分配它的片到其他的进程。在每次迭代中，每个进程将它上面的行传递给上一个进程，下面的行传递给下一个进程，因为他们需要知道迭代相邻点的值。然后他们重复迭代。每 500 次迭代将使用 gather 从每个进程中收集估计误差。你可以和我们在 f2py 中编写的求解程序来比较这些输出。

批注 [by18]: It divides a square into n parallel strips where n is the number of processes and uses jacobi-iteration.

```
from mpi4py import MPI
import numpy
size=MPI.size
rank=MPI.rank
num_points=500
sendbuf=[]
root=0
dx=1.0/(num_points-1)
from numpy import r_
j=numpy.complex(0,1)
rows_per_process=num_points/size
max_iter=5000
num_iter=0
total_err=1

def numpyTimeStep(u,dx,dy):
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u_old=u.copy()
    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
```

```
                    (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
    v = (u - u_old).flat
    return u,numpy.sqrt(numpy.dot(v,v))


if MPI.rank==0:
    print("num_points: %d"%num_points)
    print("dx: %f"%dx)
    print("row_per_procss: %d"%rows_per_process)
    m=numpy.zeros((num_points,num_points),dtype=float)
    pi_c=numpy.pi
    x=r_[0.0:pi_c:num_points*j]
    m[0,:]=numpy.sin(x)
    m[num_points-1,:]=numpy.sin(x)
    l=[ m[i*rows_per_process:(i+1)*rows_per_process,:] for i in
range(size)]
    sendbuf=l


my_grid=MPI.COMM_WORLD.Scatter(sendbuf,root)

while num_iter <  max_iter and total_err > 10e-6:


    if rank==0:
        MPI.COMM_WORLD.Send(my_grid[-1,:],1)

    if rank > 0 and rank< size-1:
        row_above=MPI.COMM_WORLD.Recv(rank-1)
        MPI.COMM_WORLD.Send(my_grid[-1,:],rank+1)

    if rank==size-1:
        row_above=MPI.COMM_WORLD.Recv(MPI.rank-1)
        MPI.COMM_WORLD.Send(my_grid[0,:],rank-1)

    if rank > 0 and rank< size-1:
        row_below=MPI.COMM_WORLD.Recv(MPI.rank+1)
        MPI.COMM_WORLD.Send(my_grid[0,:],MPI.rank-1)

    if rank==0:
        row_below=MPI.COMM_WORLD.Recv(1)



    if rank >0 and rank < size-1:
        row_below.shape=(1,num_points)
        row_above.shape=(1,num_points)
        u,err=numpyTimeStep(r_[row_above,\
```

```
                        my_grid,row_below],dx,dx)
        my_grid=u[1:-1,:]

    if rank==0:
        row_below.shape=(1,num_points)
        u,err=numpyTimeStep(r_[my_grid,row_below],dx,dx)
        my_grid=u[0:-1,:]

    if rank==size-1:
        row_above.shape=(1,num_points)
        u,err=numpyTimeStep(r_[row_above,my_grid],dx,dx)
        my_grid=u[1:,:]


    if num_iter%500==0:
        err_list=MPI.COMM_WORLD.Gather(err,root)
        if rank==0:
            total_err = 0
            for a in err_list:
                total_err=total_err+numpy.math.sqrt( a**2)
            total_err=numpy.math.sqrt(total_err)
            print("error: %f"%total_err)


    num_iter=num_iter+1



recvbuf=MPI.COMM_WORLD.Gather(my_grid,root)
if rank==0:
    sol=numpy.array(recvbuf)
    sol.shape=(num_points,num_points)
##写下代码使用这个解决方法解决你的问题
    print num_iter
    print sol
```

对于更小的网格，这可能会比简单的串行实现要慢，这是因为通信上的开销，对于更小的网格进程间的通信将比仅仅做迭代消耗更多的时间。然而，我发现在一个 1000x1000 的网格上使用 4 线程，并行版本花费了仅仅 6s，但是我们之前编写串行版本花费了 20s。

练习：重写上面的使用 f2py 或 weave 的程序，以便每个进程编译一个 Fortran 或 C 的 timestep 函数然后使用它，你得到的这个版本有多快？

批注 [by19]: Rewrite the above using f2py or weave, so that each process compiles a fortran or C timestep function and uses that, how fast can you get this?

可视化

计算机代数系统的一个最常用功能当然就是绘图了。目前 Sage 本身不支持可视化的。然而有许多非常强大的可视化程序可以在 Sage 中使用。本章的目的是讲解如何设置在它们。

我将使用的最重要的工具是 VTK, http://www.vtk.org。VTK 是一个神奇的可视化工具它可以实现的结果是令人难以置信的。他或许是现有的最好的可视化工具之一，他或许可以做你所希望一个可视化工具能完成的任何任务。当然，由于它可以完成如此之多，所以它在使用上可能有些棘手。幸运的是在 Python 上有几个工具可以使 VTK 更易上手。首先我们将关注 MavaVi, http://mayavi.sourceforge.net/, 和 easyviz.

安装可视化工具

在 Linux 上安装可视化工具

这部分假定你运行在 Linux 上。你可能需要管理员权限才能完成此部分。首先尝试：

```
sage: import Tkinter
```

若可以正常运行，那太棒了；如果不能，那意味着你的 Sage 在编译时并没有绑定 tcl/tk 。有两种原因：你若是用的是二进制，会出现这种情况；你若是从源码编译安装的但是系统之中不包含 tcl/tk 程序库，也会这样。可以为你的 Linux 发行版安装 tcl 和 tk 让程序库和开发包 (source 和 headers) 。然后在创建你 Sage 的 python

```
sage: install_package('python-2.5<version>.spkg')
```

在这里，你要替换使用 Python 的版本替换 < version > ，键入：

```
sage: !ls spkg/standard | grep python-2.5
```

这将提供给你上面使用的程序包的名字。此时，它给出多个名称，选择最高的一个版本号。现在再次尝试：

```
sage: import Tkinter
```

若它可以正常工作了，那我们就安装 vtk，但是你首先要执行*cmake*。通过在 Shell 中执行 cmake 测试你的系统中是否已经存在 (或在 Sage 中执行 !cmake)。如果已经存在了 cmake，那就准备就绪了，若不存在，可用发行版的软件安装工具安装，或执行：

```
sage: install_package('cmake-2.4.7')
```

现在我们想要编译用来完成所有艰难工作的 VTK 。要完成这些，要确保你系统中已经安装了 opengl 函数库。或许要安装类似于 libgl1-mesa-glx, libgl1-mesa-dev 的包。

```
sage: install_package('vtk-5.0.3.p1')
```

这将花费相当一段时间来编译，或许二十分钟到一个小时左右。一旦这完成了，我们就在安装 Python 包，下一部分将花费大约 10s。

```
sage: install_package('MayaVi-1.5')
sage: install_package('scitools++')
sage: install_package('PyVTK-0.4.74')
```

现在我们就完成了。

在 OS X 上安装可视化软件

首先我们要做的就是使用 OSX 的重建框架，这样它就可以创建一个图形化的窗口。要完成这些首先要在 terminal 中执行：

```
cd $SAGE_ROOT/local/lib
rm libpng*.dylib
```

这儿$SAGE_ROOT 就是 Sage 的安装目录。接下来在 Sage 中执行：

```
sage: install_package('python-2.5.1-framework')
```

下一步，我们构建 vtk，这将花费一段时间。

```
sage: install_package('vtk-5.0.3.p1')
```

最终：

```
sage: install_package('MayaVi-1.5')
sage: install_package('scitools++')
sage: install_package('PyVTK-0.4.74')
```

绘图

我们将使用两种方式绘制一个曲面。首先我们使用 easyviz。考虑下面代码：

```
import numpy
from scitools import easyviz
x = numpy.arange(-8,8,.2)
xx,yy = numpy.meshgrid(x,x)
r = numpy.sqrt(xx**2+yy**2) + 0.01
zz = numpy.sin(r)/r
easyviz.surfc(x,x,zz)
```

函数 surfc 需要一列 x 坐标，y 坐标和 NumP 数组 z。他绘制了一个在点$(\boldsymbol{x}[\boldsymbol{i}], \boldsymbol{y}[\boldsymbol{i}])$处高度为$\boldsymbol{z}[\boldsymbol{i}, \boldsymbol{j}]$的曲面，注意：使用 meshgrid 和矢量化 numpy 函数，在网格上求解

$$\frac{sin\sqrt{x^2+y^2}+1}{\sqrt{x^2+y^2}+1},$$

变的很容易。我们在前面在讨论 NumPy 时，已经讨论过 mashgrid。注意，现在你可以使用鼠标拖拽来从不同的角度观察图片。
我们可以通过添加一些底纹，颜色和一些标签是图片看起来更棒。就如下面这样：

```
import numpy
RealNumber=float
```

```
Integer =int
from scitools import easyviz
x = numpy.arange(-8,8,.2)
xx,yy = numpy.meshgrid(x,x)
r = numpy.sqrt(xx**2+yy**2) + 0.01
zz = numpy.sin(r)/r
l = easyviz.Light(lightpos=(-10,-10,5), lightcolor=(1,1,1))
easyviz.surfc(x,x,zz,shading='interp',colormap=easyviz.jet(),
          zmin=-0.5,zmax=1,clevels=10,
          title='r=sqrt(x**2+y**2)+eps\nsin(r)/r',
          light=l,
          legend='sin',
          )
```

现在让我们绘出一个矢量场。考虑下面的代码：

```
import numpy
from scitools import easyviz
RealNumber=float
Integer=int
j=numpy.complex(0,1)
w=numpy.zeros((5,5,5))
u=w+1.0
xx,yy,zz=numpy.mgrid[-1.0:1.0:5*j,-1:1:5*j,-1:1:5*j]
easyviz.quiver3(xx,yy,zz,w,w,u)
```

这需要绘出一个点无处不在的向量场。quiver3 的参数有 6 个 $n \times n \times n$ 的数组。前三个数组使向量的起始位置，三个组成一个点

$$(xx[i,j,k], yy[i,j,k], zz[i,j,k]) \; for \; 0 \leq i,j,k < n$$

第二组的三个数组是指向，例如，矢量是由 $(xx[i,j,k], yy[i,j,k], zz[i,j,k])$ 指向 $(w[i,j,k], w[i,j,k], u[i,j,k])$的。

现在我们给一个 MayaVi 的例子。首先让我们看一下如何像使用 easyviz 那样绘出一个函数：

```
import numpy
from mayavi.tools import imv
x=numpy.arange(-8,8,.2)
def f(x,y):
    r=numpy.sqrt(x**2+y**2)+.01
    return numpy.sin(r)/r
imv.surf(x,x,f)
```

这将会打开 mayavi，显示函数的图像。surf 的前两个参数是数组 x 和 y，使得函数在点 $(x[i], y[j])$处求值。最后一个参数是需要绘图的函数。他或许看起来与 easyviz 的示例有些不同可以试着让它看起来类似于 easyviz 例子。首先需要注意，在左侧有一个过滤列表和模块。在过滤菜单上双击 warpscalars 按钮，然后，改变比例因子，从 1 到 5。这需要像 easyviz 那样重新绘制这幅图。有相当多的其他选项，你可以尽情发挥。例如，接下来

单击 surfacemap 模块，然后你将看到可以使图像由不透明转变为透明。你也可以改变成线框或使它绘制轮廓。

备忘：给出更多的例子