Numerical Computing with Sage

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

This document is designed to introduce the reader to the tools in Sage that are useful for doing numerical computation. By numerical computation we essentially mean machine precision floating point computations. In particular, things such as optimization, numerical linear algebra, solving ODE's/PDE's numerically, etc. or the first part of this document the reader is only assumed to be familiar with Python/Sage. In the second section on using compiled code, the computational prerequisites increase and I assume the reader is comfortable with writing programs in C or Fortran. The third section is on mpi and parallel programming and only requires knowledge of Python, though familiarity with mpi would be helpful. Finally the last section is about 3d visualization in Sage.

In the current version of this document the reader is assumed to be familiar with the techniques of numerical analysis. The goal of this document is not to teach you numerical analysis, but to explain how to express your ideas in Sage and Python. Also this document is not meant to be comprehensive. Instead the goal is to be a road map and orient the reader to the packages relevant to numerical computation and to where they can find more information.

CHAPTER

ONE

NUMERICAL TOOLS

Sage has many different components that may be useful for numerical analysis. In particular three packages deserve mention, they are numpy, SciPy, and cvxopt. Numpy is an excellent package that provides fast array facilities to python. It includes some basic linear algebra routines, vectorized math routines, random number generators, etc. It supports a programming style similar to one would use in matlab and most matlab techniques have an analogue in numpy. SciPy builds on numpy and provides many different packages for optimization, root finding, statistics, linear algebra, interpolation, FFT and dsp tools, etc. Finally cvxopt is an optimization package which can solve linear and quadratic programming problems and also has a nice linear algebra interface. Now we will spend a bit more time on each of these packages.

Before we start let us point out http://www.scipy.org/NumPy_for_Matlab_Users, which has a comparison between matlab and numpy and gives numpy equivalents of matlab commands. If you're not familiar with matlab, thats fine, even better, it means you won't have any pre-conceived notions of how things should work. Also this http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf is a very nice tutorial on SciPy and numpy which is more comprehensive than ours.

1.1 NumPy

NumPy is not imported into sage initially. To use NumPy, you first need to import it.

```
sage: import numpy
```

The basic object of computation in NumPy is an array. It is simple to create an array.

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

NumPy arrays can store any type of python object. However, for speed, numeric types are automatically converted to native hardware types (i.e., `int`, `float`, etc.) when possible. If the value or precision of a number cannot be handled by a native hardware type, then an array of Sage objects will be created. You can do calculations on these arrays, but they may be slower than using native types. When the numpy array contains Sage or python objects, then the data type is explicitly printed as `object`. If no data type is explicitly shown when NumPy prints the array, the type is either a hardware float or int.

```
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() # higher precision than hardware floating point numbers
67
```

```
sage: numpy.array([a,2*a,3*a])
array([2.000000000000000000, 4.000000000000000000, 6.000000000000000000], dtype=object)
```

The `dtype` attribute of an array tells you the type of the array. For fast numerical computations, you generally want this to be some sort of float. If the data type is float, then the array is stored as an array of machine floats, which takes up much less space and which can be operated on much faster.

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

You can create an array of a specific type by specifying the `dtype` parameter. If you want to make sure that you are dealing with machine floats, it is good to specify `dtype=float` when creating an array.

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

You can access elements of a NumPy array just like any list, as well as take slices

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

You can do basic arithmetic operations

```
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])
```

Note that `l*l` will multiply the elements of `l` componentwise. To get a dot product, use `numpy.dot()`.

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

We can also create two dimensional arrays

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

This is basically equivalent to the following

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

The difference is that with `numpy.array()`, `m` is treated as just an array of data. In particular `m*m` will multiply componentwise, however with `numpy.matrix()`, `m*m` will do matrix multiplication. We can also do matrix vector multiplication, and matrix addition

```
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.]])
```

If `n` was created with `numpy.array()`, then to do matrix vector multiplication, you would use `numpy.dot(n,v)`.

All NumPy arrays have a shape attribute. This is a useful attribute to manipulate

```
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.]])
```

This changes the one-dimensional array into a 5×5 array.

NumPy arrays can be sliced as well

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

It is important to note that the sliced matrices are references to the original

```
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.]])
```

You will note that the original matrix changed. This may or may not be what you want. If you want to change the sliced matrix without changing the original you should make a copy

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

Some particularly useful commands are

```
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])
```

You can see that `numpy.arange()` creates an array of floats increasing by 0.1 from 0 to 2. There is a useful command `numpy.r_()` that is best explained by example

```
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_()` provides a shorthand for constructing NumPy arrays efficiently. Note in the above `0.0:5.0` was shorthand for `0.0, 1.0, 2.0, 3.0, 4.0`. Suppose we want to divide the interval from 0 to 5 into 10 intervals. We can do this as follows

```
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. ])
```

The notation `0.0:5.0:11*j` expands to a list of 11 equally space points between 0 and 5 including both endpoints. Note that `j` is the NumPy imaginary number, but it has this special syntax for creating arrays. We can combine all of these techniques

```
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. ])
```

Another useful command is `numpy.meshgrid()`, it produces meshed grids. As an example suppose you want to evaluate $f(x,y) = x^2 + y^2$ on a an equally spaced grid with $\Delta x = \Delta y = .25$ for $0 \leq x, y \leq 1$. You can do that as follows

```
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.    ]])
```

You can see that `numpy.meshgrid()` produces a pair of matrices, here denoted xx and yy, such that $(xx[i,j], yy[i,j])$ has coordinates $(x[i], y[j])$. This is useful because to evaluate f over a grid, we only need to evaluate it on each pair of entries in xx, yy. Since NumPy automatically performs arithmetic operations on arrays componentwise, it is very easy to evaluate functions over a grid with very little code.

A useful module is the `numpy.linalg` module. If you want to solve an equation $Ax = b$ do

```
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.])
```

This creates a random 5x5 matrix `A`, and solves $Ax = b$ where `b=[0.0,1.0,2.0,3.0,4.0]`. There are many other routines in the `numpy.linalg` module that are mostly self-explanatory. For example there are `qr` and `lu` routines for doing QR and LU decompositions. There is also a command `eigs` for computing eigenvalues of a matrix. You can always do `<function name>?` to get the documentation which is quite good for these routines.

Hopefully this gives you a sense of what NumPy is like. You should explore the package as there is quite a bit more functionality.

1.2 SciPy

Again I recommend this http://www.scipy.org/Wiki/Documentation?action=AttachFile&do=get&target=scipy_tutorial.pdf. There are many useful SciPy modules, in particular scipy.optimize, scipy.stats, scipy.linalg, scipy.linsolve, scipy.sparse, scipy.integrate, scipy.fftpack, scipy.signal, scipy.special. Most of these have relatively good documentation and often you can figure out what things do from the names of functions. I recommend exploring them. For example if you do

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

Then

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

will show a list of available functions. You should see a bunch of routines for finding minimum of functions. In particular if you do

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

you find it is a routine that uses the conjugate gradient algorithm to find the minima of a function.

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

will show all the special functions that SciPy has. Spending a little bit of time looking around is a good way to familiarize yourself with SciPy. One thing that is sort of annoying, is that often if you do scipy.:math:⟨ tab ⟩. You won't see a module that is importable. For example scipy.:math:⟨ tab ⟩ will not show a signal module but

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

and then

```
signal.[tab]
```

will show you a large number of functions for signal processing and filter design. All the modules I listed above can be imported even if you can't see them initially.

1.2.1 scipy.integrate

This module has routines related to numerically solving ODE's and numerical integration. Lets give an example of using an ODE solver. Suppose you want to solve the ode

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

which as a system reads

$$x' = y$$

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

The module we want to use is odeint in scipy.integrate. We can solve this ode, computing the value of (y, y'), at 1000 points between 0, and 100 using the following code.

```
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)
```

We could plot the solution if we wanted by doing

```
sage: p=[point((x[i],y[i][0])) for i in range(len(x))]
sage: plot(p).show()
```

1.2.2 Optimization

The Optimization module has routines related to finding roots, least squares fitting, and minimization. ⟨ To be Written ⟩

1.3 Cvxopt

Cvxopt provides many routines for solving convex optimization problems such as linear and quadratic programming packages. It also has a very nice sparse matrix library that provides an interface to umfpack (the same sparse ma-

trix solver that matlab uses), it also has a nice interface to lapack. For more details on cvxopt please refer to its documentation at http://cvxopt.org/userguide/index.html

Sparse matrices are represented in triplet notation that is as a list of nonzero values, row indices and column indices. This is internally converted to compressed sparse column format. So for example we would enter the matrix

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

by

```
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)
```

To solve an equation $AX = B$, with $B = [1, 1, 1, 1, 1]$, we could do the following.

```
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]
```

Note the solution is stored in B afterward. also note the m(B), this turns our numpy array into a format cvxopt understands. You can directly create a cvxopt matrix using cvxopt's own matrix command, but I personally find numpy arrays nicer. Also note we explicitly set the shape of the numpy array to make it clear it was a column vector.

We could compute the approximate minimum degree ordering by doing

```
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]
```

For a simple linear programming example, if we want to solve

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

```
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']      # ... below since can get -00 or +00 depending on architecture
[ 1.00e...00]
[ 1.00e+00]
```

CHAPTER

TWO

USING COMPILED CODE INTERACTIVELY

This section is about using compiled code in Sage. However, since Sage is built on top of Python most of this is valid for Python in general. 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. You should look at the f2py website for information on using the command line f2py tool. The ctypes and weave example will work in any recent Python install (weave is not part of Python so you will have to install it separately). If you are using Sage then weave, ctypes, and f2py are all there already.

Firstly why would we want to write compiled code? Obviously, because its fast, far faster than interpreted Python code. Sage has very powerful facilities that allow one to interactively call compiled code written in C or Fortran. In fact there 2-4 ways to do this depending on exactly what you want to accomplish. One way is to use Cython. Cython is a language that is a hybrid of C and Python based on Pyrex. It has the ability to call external shared object libraries and is very useful for writing Python extension modules. Cython/Pyrex is covered in detail elsewhere in the Sage documentation.

Suppose that you really want to just write Python code, but there is some particularly time intensive piece of your code that you would like to either write in C/Fortran or simply call an external shared library to accomplish. In this case you have three options with varying strengths and weaknesses.

Note that before you try to use compiled code to speed up your bottleneck make sure there isn't an easier way. In particular, first try to vectorize, that is express your algorithm as arithmetic on vectors or numpy arrays. These arithmetic operations are done directly in C so will be very fast. If your problem does not lend itself to being expressed in a vectorized form them read on.

Before we start let us note that this is in no way a complete introduction to any of the programs we discuss. This is more meant to orient you to what is possible and what the different options will feel like.

2.1 f2py

F2py is a very nice package that automatically wraps fortran code and makes it callable from Python. The Fibonacci examples are taken from the f2py webpage http://cens.ioc.ee/projects/f2py2e/.

From the notebook the magic %fortran will automatically compile any fortran code in a cell and all the subroutines will become callable functions (though the names will be converted to lowercase.) As an example paste the following into a cell. It is important that the spacing is correct as by default the code is treated as fixed format fortran and the compiler will complain if things are not in the correct column. To avoid this, you can write fortran 90 code instead by making your first line !f90. There will be an example of this later.

```
%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
```

Now evaluate it. It will be automatically compiled and imported into Sage (though the name of imported function will be lowercase). Now we want to try to call it, we need to somehow pass it an array A, and the length of the array N. The way it works is that numpy arrays will be automatically converted to fortran arrays, and Python scalars converted to fortran scalars. So to call fib we do the following.

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

Note that fortran is a function that can be called on any string. So if you have a fortran program in a file my prog.f. Then you could do the following

```
f=open('my_prog.f','r')
s=f.read()
fortran(s)
```

Now all the functions in my prog.f are callable.

It is possible to call external libraries in your fortran code. You simply need to tell f2py to link them in. For example suppose we wish to write a program to solve a linear equation using lapack (a linear algebra library). The function we want to use is called dgesv and it has the following signature.

```
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.
```

we could do the following. Note that the order that library are in the list actually matters as it is the order in which they are passed to gcc. Also fortran.libraries is simply a list of names of libraries that are linked in. You can just directly set this list. So that fortran.libraries=['lapack','blas']is equivalent to the following.

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

Now

```
%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
```

There are a couple things to note about this. As we remarked earlier, if the first line of the code is !f90, then it will be treated as fortran 90 code and does not need to be in fixed format. To use the above try

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

This will solve the linear system ax=b and store the result in b. If your library is not in Sage's local/lib or in your path you can add it to the search path using

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

You can also directly set fortran.library paths by assignment. It should be a list of paths (strings) to be passed to gcc. To give you an idea of some more things you can do with f2py, note that using intent statements you can control the

way the resulting Python function behaves a bit bitter. For example consider the following modification of our original fibonacci code.

```
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
```

Note the comments with the intent statements. This tells f2py that n is an input parameter and a is the output. This is called as

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

In general you will pass everything declared intent(in) to the fortran function and everything declared intent(out) will be returned in a tuple. Note that declaring something intent(in) means you only care about its value before the function is called not afterwards. So in the above n tells us how many fibonacci numbers to compute we need to specify this as an input, however we don't need to get n back as it doesn't contain anything new. Similarly A is intent(out) so we don't need A to have an specific value beforehand, we just care about the contents afterwards. F2py generates 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). All arguments are intent(in) by default.

Consider now the following

```
%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
```

You might be expecting Rescale(a,n) to rescale a numpy matrix a. Alas this doesn't work. Anything you pass in is unchanged afterwards. Note that in the fibonacci example above, the one dimensional array was changed by the fortran code, similarly the one dimensional vector b was replaced by its solution in the example where we called lapack while the matrix A was not changed even then dgesv says it modifies the input matrix. Why does this not happen with the two dimensional array. Understanding this requires that you are aware of the difference between how fortran and C store arrays. Fortran stores a matrices using column ordering while C stores them using row ordering. That is the

matrix

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

is stored as

$(0\,1\,2\,3\,4\,5)$ in C

$(0\,3\,1\,4\,2\,5)$ in Fortran

One dimensional arrays are stored the same in C and Fortran. Because of this f2py allows the fortran code to operate on one dimensional vectors in place, so your fortran code will change one dimensional numpy arrays passed to it. However, since two dimensional arrays are different by default f2py copies the numpy array (which is stored in C format) into a second array that is in the fortran format (i.e. takes the transpose) and that is what is passed to the fortran function. We will see a way to get around this copying later. First let us point one way of writing the rescale function.

```
%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
```

Note that to call this you would use

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

Note here I am not passing in n which is the dimension of a. Often f2py can figure this out. This is a good time to mention that f2py automatically generates some documentation for the Python version of the function so you can check what you need to pass to it and what it will return. To use this try

```
rescale?
```

The intent(in,out) directives tells f2py to take the contents of a at the end of the subroutine and return them in a numpy array. This still may not be what you want. The original a that you pass in is unmodified. If you want to modify the original a that you passed in use intent(inout). This essentially lets your fortran code work with the data inplace.

```
%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
```

If you wish to have fortran code work with numpy arrays in place, you should make sure that your numpy arrays are stored in fortran's format. You can ensure this by using the order='FORTRAN' keyword when creating the arrays, as follows.

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

After this executes, a will have the rescaled version of itself. There is one final version which combines the previous two.

```
%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
```

The (in,out,overwite) intent says that if a is in FORTRAN ordering we work in place, however if its not we copy it and return the contents afterwards. This is sort of the best of both worlds. Note that if you are repeatedly passing large numpy arrays to fortran code, it is very important to avoiding copying the array by using (inout) or (in,out,overwrite). Remember though that your numpy array must use Fortran ordering to avoid the copying.

For more examples and more advanced usage of F2py you should refer to the f2py webpage http://cens.ioc.ee/projects/f2py2e/. The command line f2py tool which is referred to in the f2py documentation can be called from the Sage shell using

```
!f2py
```

2.2 More Interesting Examples with f2py

Let us now look at some more interesting examples using f2py. We will implement a simple iterative method for solving laplace's equation in a square. Actually, this implementation is taken from http://www.scipy.org/PerformancePython?highlight=%28performance%29 page on the scipy website. It has lots of information on implementing numerical algorithms in python.

The following fortran code implements a single iteration of a relaxation method for solving Laplace's equation in a square.

```
%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
```

If you do

```
timestep?
```

You find that you need pass timestep a numpy array u, and the grid spacing dx,dy and it will return the updated u, together with an error estimate. To apply this to actually solve a problem use this driver code

```
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)
```

Now call the routine using

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

This solves the equation with boundary conditions that the right and left edge of the square are half an oscillation of the sine function. With a 51x51 grid that we are using I find that it takes around .2 s to solve this requiring 2750 iterations. If you have the gnuplot package installed (use optional packages() to find its name and install package to install it), then you can plot this using

```
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)
```

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):
    """Takes a time step using straight forward Python loops."""
    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()
    # 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))
```

You can try these out by changing the timestep function used in our driver routine. The python version is slow even on a 50x50 grid. It takes 70 seconds to solve the system in 3000 iterations. It takes the numpy routine 2 seconds to reach the error tolerance in around 5000 iterations. In contrast it takes the f2py routine around .2 seconds to reach the error tolerance using 3000 iterations. I should point out that the numpy routine is not quite the same algorithm since it is a jacobi iteration while the f2py one is gauss-seidel. This is why the numpy version requires more iterations. Even accounting for this you can see the f2py version appears to be around 5 times faster than the numpy version. Actually if you try this on a 500x500 grid I find that it takes the numpy routine 30 seconds to do 500 iterations while it only takes about 2 seconds for the f2py to do this. So the f2py version is really about 15 times faster. On smaller grids each actual iteration is relatively cheap and so the overhead of calling f2py is more evident, on larger examples where the iteration is expensive, the advantage of f2py is clear. Even on the small example it is still very fast. Note that a 500x500 grid in python would take around half an hour to do 500 iterations.

To my knowledge the fastest that you could implement this algorithm in matlab would be to vectorize it exactly like the numpy routine we have. Vector addition in matlab and numpy are comparable. So unless there is some trick I don't know about, using f2py you can interactively write code 15 times faster than anything you could write in matlab (Please correct me if I'm wrong). You can actually make the f2py version a little bit faster by using intent(in,out,overwrite) and creating the initial numpy array using order='FORTRAN'. This eliminates the unnecessary copying that is occurring in memory.

2.3 Weave

Weave is a tool that does for C/C++ what f2py does for fortran (though we should note it is also possible to wrap C code using f2py). Suppose we have some data stored in numpy arrays and we want to write some C/C++ code to do something with that data that needs to be fast. For a trivial example, let us write a function that sums the contents of a numpy array

```
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
```

To call this function do

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

The first time the weave code executes the code is compiled, from then on, the execution is immediate. You should find that python's built-in sum function is comparable in speed to what we just wrote. Let us explain some things about this example. As you can see, to use weave you create a string containing pure C/C++ code. Then you call weave.inline on it. You pass to weave the string with the code, as well as a list of python object that it is to automatically convert to C variables. So in our case we can refer to the python objects a and n inside of weave. Numpy arrays are accessed by $a(i)$ if they are one-dimensional or $a(i,j)$ if they are two dimensional. Of course we cannot use just any python object, currently weave knows about all python numerical data types such as ints and floats, as well as numpy arrays. Note that numpy arrays do not become pointers in the C code (which is why they are accessed by ( ) and not [ ]. If you need a pointer you should copy the data into a pointer. Next is a more complicated example that calls lapack to solve a linear system 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
```

Note that we have used the support_code argument which is additional C code you can use to include headers and declare functions. Note that inline also can take all distutils compiler options which we used here to link in lapack.

```
def weaveTimeStep(u,dx,dy):
    """Takes a time step using inlined C code -- this version uses
    blitz 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);
           """
    # compiler keyword only needed on windows with MSVC installed
    err = weave.inline(code, ['u', 'dx2', 'dy2', 'dnr_inv', 'nx','ny'],
                       type_converters = converters.blitz,
                       compiler = 'gcc')
    return u,err
```

Using our previous driver you should find that this version takes about the same amount of time as the f2py version

around .2 seconds to do 2750 iterations.

For more about weave see http://www.scipy.org/Weave

2.4 Ctypes

Ctypes is a very interesting python package which lets you import shared object libraries into python and call them directly. I should say that even though this is called ctypes, it can be used just as well to call functions from libraries written in fortran. The only complication is you need to know what a fortran function looks like to C. This is simple everything is a pointer, so if your fortran function would be called as foo(A,N) where A is an array and N is its length, then to call it from C it takes a pointer to an array of doubles and a pointer to an int. The other thing to be aware of is that from C, fortran functions usually have an underscore appended. That is, a fortran function foo would appear as foo from C (this is usually the case but is compiler dependent). Having said this, the following examples are in C.

As an example suppose you write the following simple C program

```
#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;
}
```

which you want to call from python. First make a shared object library by doing (at the command line)

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

Note that on OSX -shared should be replaced by -dynamiclib and sum.so should be called sum.dylib Then you can do

```
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))
```

Note here that a.ctypes.data as(c void p) returns a ctypes object that is void pointer to the underlying array of a. Note that even though sum takes a double*, as long as we have a pointer to the correct data it doesn't matter what its type is since it will be automatically cast.

Note that actually there are other ways to pass in the required array of doubles. For example

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

This example only uses ctypes. Ctypes has wrappers for C data types so for example

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

would create a ctypes double object which could be passed to a C function. Note that there is a byref function that lets you pass parameters by reference. This is used in the next example. c double*10, is a python object that represents an array of 10 doubles and

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

sets a equal to an array of 10 doubles. I find this method is usually less useful than using numpy arrays when the data is mathematical as numpy arrays are more well integrated into python and sage.

Here is an example of using ctypes to directly call lapack. Note that this will only work if you have a lapack shared object library on your system. Also on linux the file would be liblapack.so and you will probably use dgesv (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))
```

For completeness, let us consider a way to solve the laplace equation using C types. Suppose you have written a simple solver in C and you want to call it from python so you can easily test different boundary conditions. Your C program might look like this.

```
#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;
}
```

We can compile it by running at the command line

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

Now in sage (notebook or command line) execute

```
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)
```

Note the line laplace.timestep.restype=c double. By default ctypes assumes the return values are ints. If they are not you need to tell it by setting restype to the correct return type. If you execute the above code, then solve(u) will solve the system. It is comparable to the weave or fortran solutions taking around .2 seconds. Alternatively you could do

```
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)
```

which computes the solution entirely in C. This is very fast. Admittedly we could have had our fortran or weave routines do the entire solution at the C/Fortran level and we would have the same speed.

As I said earlier you can just as easily call a shared object library that is written in Fortran using ctypes. The key point is it must be a shared object library and all fortran arguments are passed by reference, that is as pointers or using byref. Also even though we used very simple data types, it is possible to deal with more complicated C structures. For this and more about ctypes see http://python.net/crew/theller/ctypes/

2.5 More complicated ctypes example

Here we will look at a more complicated example. First consider the following C code.

```
#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;
    }
    }

}
```

Put it in a file called linked_list_sparse.c and compile it using

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

Next consider the following python helper code.

```
from ctypes import *

class double_row_element(Structure):
    pass

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


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


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
```

Lets discuss the above code. The original C code stored a sparse matrix as a linked list. The python code uses the ctypes Structure class to create structures mirroring the structs in the C code. To create python object representing a C

struct, simply create class that derives from Structure. The _fields_ attribute of the class must be set to a list of tuples of field names and values. Note that in case you need to refer to a struct before it is completely defined (as in the linked list) you can first declare it with "Pass", and then specify the field contents as above. Also note the POINTER operator which creates a pointer out of any ctypes type. We are able to directly call our library 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=get_value(m,c_int(4),c_int(4))
print("%f"%a)
free_matrix(m)
```

Note that you can access the contents of a structure just by (struct_object).field name. However for pointers, there is a contents attribute. So, in the above, m.contents.nrows would let you access the nrows field. In fact you can manually 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)
```

2.6 Comparison to Cython/Pyrex

It is certainly possible to write a solver in Cython or Pyrex. From the http://www.scipy.org/PerformancePython?highlight=%28performance%29 website you can find an example. One potential downside to Cython over the previous solutions is it requires the user to understand how NumPy arrays or Sage matrices are implemented so as to be able to access their internal data. In contrast the weave, scipy, and ctypes examples only require the user to know C or Fortran and from their perspective the NumPy data magically gets passed to C or Fortran with no further thought from them. In order for pyrex to be competitive as a way to interactively write compiled code, the task of accessing the internal structure of NumPy arrays or Sage matrices needs to be hidden.

CHAPTER

THREE

PARALLEL COMPUTATION

3.1 mpi4py

MPI which stands for message passing interface is a common library for parallel programming. There is a package mpi4py that builds on the top of mpi, and lets arbitrary python objects be passed between different processes. These packages are not part of the default sage install. To install them do

```
sage: optional_packages()
```

Find the package name openmpi-* and mpi4py-*and do

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

Note that openmpi takes a while to compile (15-20 minutes or so). Openmpi can be run on a cluster, however this requires some set up so that processes on different machines can communicate (though if you are on a cluster this is probably already set up). The simplest case is if you are on a shared memory or multicore system where openmpi will just work with no configuration from you. To be honest, I have never tried to run mpi4py on a cluster, though there is much information about these topics online.

Now, the way that mpi works is you start a group of mpi processes, all of the processes run the same code. Each process has a rank, that is a number that identifies it. The following pseudocode indicates the general format of MPI programs.

```
   ....

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:
   ....
```

Each processes looks for what it's supposed to do (specified by its rank) and processes can send data and receive data. Lets give an example. Create a script with the following code in a file mpi_1.py

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

To run it you can do (from the command line in your sage directory)

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

The command mpirun -np 5 starts 5 copies of a program under mpi. In this case we have 5 copies of sage in pure python mode run the script mpi_1.py. The result should be 5 "hello worlds" plus 5 distinct ranks. The two most important mpi operations are sending and receiving. Consider the following example which you should put in a script mpi_2.py

```
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)
```

The same command as above with mpi_1.py replaced by mpi_2.py will produce 5 outputs and you will see each process creates an array and then passes it to the next guy (where the last guy passes to the first.) Note that MPI.size is the total number of mpi processes. MPI.COMM WORLD is the communication world.

There are some subtleties regarding MPI to be aware of. Small sends are buffered. This means if a process sends a small object it will be stored by openmpi and that process will continue its execution and the object it sent will be received whenever the destination executes a receive. However, if an object is large a process will hang until its destination executes a corresponding receive. In fact the above code will hang if [rank]*5 is replaced by [rank]*500. It would be better to do

```
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)
```

Now the first process initiates a send, and then process 1 will be ready to receive and then he will send and process 2 will be waiting to receive, etc. This will not lock regardless of how large of an array we pass.

A common idiom is to have one process, usually the one with rank 0 act as a leader. That processes sends data out to the other processes and processes the results and decides how further computation should proceed. Consider the following code

```
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)
```

The scatter command takes a list and evenly divides it amongst all the processes. Here the root process creates a matrix (which is viewed as a list of rows) and then scatters it to everybody (roots sendbuf is divided equally amongst the processes). Each process prints the row it got. Note that the scatter command is executed by everyone, but when root executes it, it acts as a send and a receive (root gets one row from itself), while for everyone else it is just a receive.

There is a complementary gather command that collects results from all the processes into a list. The next example uses scatter and gather together. Now the root process scatters the rows of a matrix, each process then squares the elements of the row it gets. Then the rows are all gathered up again by the root process who collects them into a new matrix.

```
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)
```

There is also a broadcast command that sends a single object to every process. Consider the following small extension. This is the same as before, but now at the end the root process sends everyone the string "done", which is printed out.

```
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 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".

3.2 Parallel Laplace Solver

The following code solves Laplace's equation in parallel on a grid. It divides a square into n parallel strips where n is the number of processes and uses jacobi-iteration. The way the code works is that the root process creates a matrix and distributes the pieces to the other processes. At each iteration each process passes its upper row to the process above and its lower row to the process below since they need to know the values at neighboring points to do the iteration. Then they iterate and repeat. Every 500 iterations the error estimates from the processes are collected using Gather. you can compare the output of this with the solver we wrote in the section on f2py.

```
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)
##Write your own code to do something with the solution
    print num_iter
    print sol
```

For small grid sizes this will be slower than a straightforward serial implementation, this is because there is overhead from the communication, and for small grids the interprocess communication takes more time than just doing the iteration. However, on a 1000x1000 grid I find that using 4 processors, the parallel version takes only 6 seconds while

the serial version we wrote earlier takes 20 seconds.

Excercise: 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?

CHAPTER

FOUR

VISUALIZATION

One of the most common uses of computer algebra systems is of course plotting. At the moment Sage does not have much in the way of intrinsic support for visualization. However, there are some very powerful open source programs for visualization that you can use from within Sage. The goal of this chapter is to explain how to set them up. The main tool we will be using is VTK, http://www.vtk.org. VTK is an amazing visualization tool, and the results that can be achieved are incredible. It is probably one of the best visualization tools in existence, and can probably do anything you might want a visualization tool to do. However, because it can do so much, it is a bit tricky to use potentially. Luckily there are a few tools for using VTK from within python that make it easier to get started. The ones we will focus on are MavaVi, http://mayavi.sourceforge.net/, and easyviz.

4.1 Installation of Visualization Tools

4.1.1 Installing Visualization Tools on Linux

This section assumes you are running linux. You may need administrator rights to complete this section. First try

```
sage: import Tkinter
```

If this works great if not it means your sage was not compiled with tcl/tk bindings. There are two possible reasons. If you used a binary then this will be the case. If you built from source but don't have the tcl/tk libraries on your system you will have the same problem. To fix this install the tcl and tk libraries and development (source and headers) packages for your linux distribution. Now you need to rebuild Sage's python

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

where you should replace < version > by the version of python. Type

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

This will give you the name of the python package to use above. In the case that it gives multiple names, choose the one with the highest version number. Now again try

```
sage: import Tkinter
```

If this works we can install vtk, but first you need cmake. Test if your system has it by typing cmake at the shell, (or !cmake in sage). If cmake is on your system then you are ready. If not either install cmake on your system using your distributions tools or do

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

Now we want to compile VTK which does all the hard work. To do this make sure you have the opengl libraries for you system installed. This will be something like libgl1-mesa-glx, libgl1-mesa-dev.

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

This will take quite a while to compile, probably 20 min to an hour or so. Once this is done we install the python wrappers, the next part takes about 10 seconds,

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

Now you're done.

4.1.2 Installing Visualization Software on OS X

The first thing we need to do is rebuild Python to use OSX's frameworks, so that it can create graphical windows. To do this first from the terminal do

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

where `$SAGE_ROOT` is the directory of your Sage install. Next from within Sage,

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

Next we will build vtk, this will take a while

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

Finally

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

4.2 Plotting

We will plot a surface two ways. First we will use easyviz. Consider the following code:

```
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)
```

The function surfc takes a list of x coordinates, and y coordinates and a numpy array z. Its plots a surface that has height z[i,j] at the point (x[i],y[i]). Note the use of meshgrid, and vectorized numpy functions that let us evaluate $\frac{\sin(\sqrt{x^2+y^2})+1}{\sqrt{x^2+y^2}+1}$ over the grid very easily. We discussed meshgrid at the beginning when we were talking about numpy. Note that you can drag the plot around with your mouse and look at it from different angles.

We can make this plot look a bit nicer by adding some shading and nicer coloring and some labels as follows.

```
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',
          )
```

Let us now try to plot some vector fields. Consider the following code

```
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)
```

This should plot a vector field that points up everywhere. The arguments to quiver3 are 6, $n \times n \times n$ arrays. The first three arrays are the location of the vectors, that is there will be a vector at $(xx[i,j,k], yy[i,j,k], zz[i,j,k])$ for $0 \leq i,j,k < n$. The second three arrays are the directions, i.e., the vector at $(xx[i,j,k], yy[i,j,k], zz[i,j,k])$ points in the direction $(w[i,j,k], w[i,j,k], u[i,j,k])$.

Now let us give some examples with MayaVi. First lets see how to plot a function like we did with easyviz.

```
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)
```

This will open mayavi, and display the plot of the function. The first two arguments to surf are arrays x and y, s.t. the function will be evaluated at $(x[i], y[j])$. The last argument is the function to graph. It probably looks a bit different than the easyviz example. Lets try to make it look similar to the easyviz example. First note that on the left there is a list of filters and modules. Double-click the warpscalars button in the filters menu, and change the scale factor from 1 to say 5. This should redraw the graph similar to how easyviz drew it. There are quite a few other options you can play around with. For example, next click on the module surfacemap, and you will see you can make the graph transparent by changing the opacity. You can also change it to a wireframe or make it plot contours.

TODO: More examples

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*