Solving the Master Equations in Python

Markus Hegland
markus.hegland@anu.edu.au

CMA/MSI/ANU and ARC Centre in Bioinformatics and NICTA
Overview

- Report on implementation done for some work for the ARC Centre in Bioinformatics and in NICTA
- Illustrates some concepts from the earlier introduction to Python
- Discuss implementations of some numerical techniques for new problems – modelling high-dimensional functions, solving linear ODEs and using Krylov space approximations
- Why Python? Rapid prototyping of ideas, understandable and efficient code, similar to Matlab in some respects, good interface to other code
- Contributions: a function class for probability distributions, a solver for linear differential equations, an approximation tool, simple master equations discretisation
A sparse grid module

- Defining the geometry – regions
  - sets in $N^d$ with an origin (lower left corner) and size (extensions in all directions), constructor
  - container
  - iterator

- Functions, domain and cell

- Sparse grid functions
Regions

A region is an interval in $\mathbb{N}^d$, i.e., a set

$$R = \{(x_1, \ldots, x_d) \mid o_i \leq x_i < o_i + s_i, \ x_i \in \mathbb{N}\}$$

We implement this as class Region:

- Constructor $R = \text{Region}(\ldots)$
- Incidence relation $x \in R$
- Iterator for $x \in R$: ...
Constructor – generate a region

usage: R = Region(size=(8,8))

class Region:
    def __init__(self, dimension=1, origin=(0,), size=(1,)):
        self.dimension = max(dimension, len(origin),
                             len(size))
        self.origin = origin
        self.size = size
        if len(self.origin) < self.dimension:
            self.origin = self.dimension*(0,)
        if len(self.size) < self.dimension:
            self.size = self.dimension*(1,)

3 attributes: dimension, origin and size
example of tuples: 4*(1,) = (1,1,1,1)
Incidence relation – check if a point is in the region

usage: if x in R: ...

def __contains__(self, item):
    import operator
    try:
        indicator=map(lambda it,orig,siz:0<=it-orig<siz, item, self.origin, self.size)
        return reduce(operator.and_, indicator, True)
    except:
        return False

- short function definition: f = lambda x,y : x*y
- apply functions to lists, reduction:
  map(sin, [1,2,3]) = [sin(1),sin(2),sin(3)]
  reduce(operator.and_, [a,b,c], True) = a & b & c
- try: if the first part returns an exception then goto except:
Iterator – iterate over a set of tuples

usage: for x in R: ...

```python
def __iter__(self):
    x = list(self.origin)
    while True:
        i = 0
        while x[i] >= self.origin[i]+self.size[i]-1:
            x[i] = self.origin[i]
            i += 1
        if i == self.dimension:
            raise StopIteration
        x[i] += 1
        yield tuple(x)
```

- StopIteration – exception raised at the end of the iteration
- yield – next step in iteration returns this
Functions

We implement piecewise constant functions and as child class a sparse grid piecewise constant function class
Construct pw constant function

eexample usage: \( f = \text{pwconstfunc}(\ldots) \)

class pwconstFunc:
    def __init__(self, dimension=1, domain=Region(),
                 cell=Region():
        self.dimension = max(dimension, domain.dimension,
                             cell.dimension)
        self.domain = domain
        self.cell = cell
        if self.cell.dimension < self.dimension:
            self.cell = Region(dimension=self.dimension)
        if self.domain.dimension < self.dimension:
            self.domain = self.cell  # constant function
        self.__initdata__()

The function is zero outside the domain and constant in every cell.
Initialise storage space

Data (function values) initialised separately to ease extensions.

```python
def __initdata__(self):
    from numpy import zeros, Float64
    shape = map(lambda siz, ce: int(siz/ce),
                self.domain.size, self.cell.size)
    self.array = zeros(shape, Float64)
```

Numpy implements a multidimensional array class (replaces Numeric and numarray and parts of scipy). “zeros” generates an array of zeros and “Float64” is the 8 Byte floating point type.
The function call

usage: \( y = f(x) \)

```python
def __call__(self, *xx):
    if isinstance(xx[0], tuple):
        x = xx[0]
    else:
        x = xx
    fvalue = 0.0
    if x in self.domain:
        indx = tuple(map(lambda orig, ce, xv: int((xv - orig) / ce),
                          self.domain.origin, self.cell.size, x))
        fvalue = self.array[indx]
    return fvalue
```

We allow both calls \( f(x_1, x_2) \) and \( f(x) \) where \( x = (x_1, x_2) \). “isinstance” to check the class or type of an object. “*xx” argument list.
class spgridFunc(pwconstFunc):
    def __init__(self, dimension=1, domain=Region(),
                 cell=Region(), maxlevel=0):
        self.maxlevel = maxlevel
        pwconstFunc.__init__(self, dimension, domain, cell)

    def __call__(self, *xx):
        fvalue = 0.0
        for f in self.componentfuncs:
            fvalue += f(*xx)
        return fvalue

One-liner for summation of values of component functions:
reduce(lambda s,f : s+f(*xx), self.componentfuncs, 0)
Inherits and initialisation from “pwconstFunc”
def __initdata__(self):
    from math import log; from operator import mul, add
    ds=self.domain.size; cs=self.cell.size;
    d=self.dimension
    # 1. Extend the domain size to 2**l1 x...x 2**ld
    levs = tuple([int(log(ds[i]/cs[i])/log(2)+0.5)
                  for i in range(d)])
    self.maxlevel = reduce(max, levs, self.maxlevel)
    ds = tuple([2**levs[i]*cs[i] for i in range(d)])
    # 2. Determine cells of components
    cl = reduce(lambda clx, i:
                 [cli+(l,) for cli in clx for l in range(levs[i]+1)
                  if reduce(add,cli,0)+l<=self.maxlevel],
                range(d),[()])
    ....
# 3. Construct the component functions

```python
self.componentfuncs = []
for l in cl:
    size = tuple(map(lambda x,y: x/2**y, ds, l))
    f = pwconstFunc(domain=self.domain, 
                    cell=(Region(size=size)))
    self.componentfuncs.append(f)
```

Some more “shorthand” – exercise in list comprehensions.
Roger Sidje’s Expokit

- State of the art code to determine $\exp(At)x$
- Reduction to Hessenberg matrix with Krylov space methods for large sparse matrices
- Pade approximation and doubling for small dense matrices
- Matlab and Fortran code
- We use expv.m, padm.m, dmexpv.f and dspadm.f
Translating expv.m to Python

Roger’s Matlab code

```matlab
V = zeros(n,m+1);
H = zeros(m+2,m+2);
V(:,1) = (1/beta)*w;
for j = 1:m
    p = A*V(:,j);
    for i = 1:j
        H(i,j) = V(:,i)'*p;
        p = p - H(i,j)*V(:,i);
    end;
    ....
V(:,j+1) = (1/s)*p;
```

Translation into Python

```python
V = []
H = zeros((m+2,m+2), Float64)
V.append((1/beta)*w)
for j in range(m):
    p = A(V[j])
    for i in range(j+1):
        H[i,j] = V[i].dot(p)
        p = p - H[i,j]*V[i]
    ....
V.append((1/s)*p)
```
Using f2py

from http://cens.ioc.ee/projects/f2py2e/:

- F2PY: Fortran to Python interface generator
- Calling Fortran and C functions from Python
- Calling Python functions Fortran or C (call-backs)
- Automatically handling the difference in the data storage order of multi-dimensional Fortran and Numerical Python (i.e. C) arrays.
- F2PY generated extension modules depend on NumPy package that provides fast multi-dimensional array language facility to Python.
f2py and expokit

- `f2py -c expokit.pyf expokit.f blas.o lapack.o` compiling the Fortran code and generating the shared object file which can be handled like a module from Python.

- How to call expokit from Python:

  ```python
  from numpy import zeros, Float64, array
  from expokit import dmexpv
  t=1; n=5; m = min(30, n-1)
  v = zeros(n, Float64); v[1] = 1.0
  wsp = zeros(n*(m+1)+n*(m+2)**2+4*(m+2)**2+7, Float64)
  iwsp = zeros(m+2)
  A = lambda v : array((-v[0],-v[1]+v[0],-v[2]+v[1]))
  w,tol,iflag = dmexpv(m,t,v,wsp,iwsp,A)
  ```
The signature file expokit.pyf

```python
python module expokit

interface

    subroutine dmexpv(n,m,t,v,w,tol,anorm,wsp,lwsp,
                       iwsp,liwsp,matvec,itrace,iflag)

    use __user__routines
    integer check(len(v)>=n),intent(hide),depend(v)::n=len(v)
    integer intent(in) :: m
    double precision intent(in) :: t
    double precision dimension(n),intent(in) :: v

    .......

    external matvec

    .......

    end subroutine dmexpv

end interface

end python module expokit
```
The callback signature

```python
python module __user__routines

interface

    subroutine matvec(n, v, w)
        integer optional, check(len(v)>=n), depend(v):: n = len(v)
        double precision dimension(n), intent(in) :: v
        double precision dimension(n), intent(out) :: w
    end subroutine matvec

end interface

end python module __user__routines

We needed to change the call to matvec in expokit.f to include \( n \) (not necessary in Python!):

```call matvec(n, wsp(j1v-n), wsp(j1v) )```
Another code from expokit: padm.m

Roger’s Matlab code

\[
\begin{align*}
I & = \text{eye}(n); \\
A_2 & = A*A; \\
Q & = c(p+1)*I; \\
P & = c(p)*I; \\
\text{odd} & = 1; \\
\text{for } k = p-1:-1:1, \\
& \quad \text{if } \text{odd} == 1, \\
& \quad \quad Q = Q*A_2 + c(k)*I; \\
& \quad \text{else} \\
& \quad \quad P = P*A_2 + c(k)*I; \\
& \quad \text{end;} \\
& \quad \text{odd} = 1-\text{odd}; \\
& \text{end;}
\end{align*}
\]

Translation into Python

\[
\begin{align*}
I & = \text{eye}(n) \\
A_2 & = A*A \\
Q & = c[p]*I \\
P & = c[p-1]*I \\
\text{odd} & = 1 \\
\text{for } k \text{ in range}(p-2,-1,-1): \\
& \quad \text{if } \text{odd} == 1: \\
& \quad \quad Q = Q*A_2 + c[k]*I \\
& \quad \text{else:} \\
& \quad \quad P = P*A_2 + c[k]*I \\
& \quad \text{odd} = 1-\text{odd}
\end{align*}
\]
A simple matrix class based on Numeric

class Matrix(Matrix.Matrix):
    import MLab, Numeric

    # identity matrix:
    eye = lambda n, ML=MLab : Matrix(ML.eye(n, ML.Float64))
    eye = staticmethod(eye)

def __mul__(A, B):    # matrix-vector product:
    if B.name == 'matrix.Vector':
        from Numeric import dot
        return Vector(dot(A.array, B.array[0,:]))
    else:
        return Matrix(numpymatrix.__mul__(A,B))
    ....

class Vector(Matrix):
    ....
Levels of Abstraction in Mathematical Software

(see Laurent’s talk)

- Application-specific interface: Programmer manipulates objects associated with the application
- High-level mathematics interface: Programmer manipulates mathematical objects, Weak forms, boundary conditions, meshes
- Algorithmic and discrete mathematics interface: PETSc emphasis
  - Programmer manipulates mathematical objects: Sparse matrices, nonlinear equations,
  - Programmer manipulates algorithmic objects: Solvers
- Low-level computational kernels: BLAS-type operations, FFT
PETSc

- Portable, Extensible Toolkit for Scientific Computation (PETSc)
- Assist with the development of complex parallel PDE codes
- Set the values: `MatSetValues(Mat,...)`
  - number of rows to insert/add
  - indices of rows and columns
  - number of columns to insert/add
  - values to add
  - mode: `[INSERT_VALUES,ADD_VALUES]`
  - `MatAssemblyBegin(Mat)`
  - `MatAssemblyEnd(Mat)`
from PETSc import *
class Matrix(Mat):
    def __init__(self, *args):
        Mat.__init__(self, destroy_me=True)
        if len(args) == 2:
            # generate zero matrix
            MatCreate(PETSC_COMM_WORLD, self.addr)
            m = args[0]
            n = args[1]
            self.setSizes(m, n, m, n)
            self.setType('seqdense')
            self.assemblyBegin(MAT_FINAL_ASSEMBLY)
            self.assemblyEnd(MAT_FINAL_ASSEMBLY)
        elif len(args) == 1:
            # conversion from sequence type
            ....
def eye(n):    # identity matrix
    A = Matrix(n,n)
    A.Shift(1.0)
    return A

eye = staticmethod(eye)
def promote(cls, mat):
    self = Matrix()
    self.destroy_me = False
    self.ref = mat
    self.set_value( mat.get_value() )
    return self

promote = classmethod(promote)
def __mul__(self, other):
    mat = self.matMult_BLAS(other)
    A = Matrix.promote(mat)
    return A
Pyrex – Features

- Pyrex is a Python-like language that is used to create C modules for Python.
- Applications:
  - Speed – C modules are faster
  - Wrap a C library to interface to Python
- C functions: `cdef f(x)`, C data types: `cdef int i,j`
- A fast for loop: `for i from 0 <= i < n:`
- Alternative: Write Python extensions – need to write a fair bit of “boilerplate code”, conversion between C and Python data types, tricky to debug
- : Pylinline, SWIG – no help to create new built-in Python type, but creates “boilerplate” and helps interface to basic Python types
How to write a C program in Python: Pyrex

Pyrex code:

```python
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        ...  
        n = n + 1
    return result
```

- Producing the C code:
  - `pyrexc primes.pyx`
- Compiling:
  - `gcc -c -I/usr/include/python2.4/ primes.c`
- Creating shared object:
  - `gcc -shared primes.o -o primes.so`
- Import in Python code:
  - `from primes import primes`
- Use in Python code:
  - `primes(40)`
C programming in Python: Pyxelator

- Classes for basic C-types like: CVoid, CInt, CDouble
- Construct and access arrays:
  ```python
  mem = CDouble.array(55)  # create a double array with 55 elements
  mem[0] = 1.2  # set the first element
  mem = CInt.array([3,4,5])  # create a new array from a list
  ```
- Pointers – how to cause segfaults:
  ```python
  mem = CDouble.pointer()  # create pointer object
  PetscMalloc(55, mem.addr)  # second arg = return arg
  x = mem.deref  # get the first element
  x.set_value(1.2)
  ```
How to get and install Lineal

- **Download**: (includes Pyrex, pyxelator, scons (make/build etc alternative), Tao (optimisation package) and PETSc)

  
  ```
  /local/repos/trunk lineal
  
  you need: numarray, blas, lapack, subversion, stuff for PETSc
  
documentation/installguide/
  ```

  ```
  export LINEAL_DIR='''/home/hegland/NICTA/lineal'''
  cd $LINEAL_DIR
  python make.py
  ```

  ```
  export PYTHONPATH=$LINEAL_DIR/lib:$PYTHONPATH
  cd $LINEAL_DIR/lib/PETSc
  python test_petsc.py ## not much output/takes a long
  ```