Improving Scientific Productivity using Python: An Example from an Ensemble Data Assimilation System in Meteorology

Dr. Louis J. Wicker
Meteorologist
NOAA National Severe Storms Laboratory
Adjunct Associate Professor
Univ of Oklahoma School of Meteorology
Louis.Wicker@noaa.gov
http://www.nssl.noaa.gov/users/lwicker/public_html

Long ago in a decade far far away from here...

- **Researchers / Students in 1985 had to know:**
  - How to program in Fortran with serial algorithms (maybe C if you were TRUELY geek...)
  - How to use a word processor
  - How to run a job on a big computer (remember Job Control Lang...eeh!)
- **Researchers / Students in 2005 need to know:**
  - 2-5 languages (F95, C++, CSH, Python, MatLab, etc...)
  - parallel programming (distributed parallelism, OMP does not count!)
  - 3 & 4D visualization
  - web programming
  - grid computing, data base management, etc, etc....
- No problem, right?

Compiled Languages

- As in ... F77, F95, F2000, C++, C, Ada, etc ...
- Generates the fastest executing code
- Are "traditional" development languages, they are the ones taught, the ones legacy codes are written in
- Development cycle:
  - write/compile/link/run ...
  - debug/compile/link/run ... repeat ...
- Access to Unix filesystem, files, URL's awkward..
- Some compiled languages (F77, C ?) do not promote the application of modern software practices like OOP, modular code, etc.

Interpreted (Scripting) Languages

- As in ...
  - Csh, Perl, Python, Ruby, Tcl, BASIC, etc.
  - Java is "in-between"
  - also IDL, Matlab, Mathematica, Maple, NCL ...
- Development cycle is
  - write/run ...
  - debug/run ...
  - debug/run ... etc.
- Built-in access to Unix, Web, etc.
- These languages tend to promote the development of good software through code reuse, built-in high-level constructs and OOP.

Advantages of Using Interpreted Languages

- Programs are generally written at a higher level
- Data structures are dynamic - do not have to be specified a a priori!
- OOP paradigm promotes code reuse and simple top level code
- Modules can include both functions and main drivers => easier development & testing
- Development cycle includes testing code snippets that you are trying to include in the modules in interpreter
  - eliminates more bugs up front
  - permits testing of new code ideas "inline"
- Result: smaller code, fewer bugs, faster development

Example: Read ascii data from file...

Input File

```
964.0000        305.29              15.713
  0.0000           305.29            15.713          -3.0730          8.4429
  127.17           304.52            15.137          -4.4020          12.094
  359.00           304.42            14.117          -5.1683          19.288
  593.61           304.35            13.016          -1.8411          19.884
  833.33           304.34            12.268           2.2744          19.839
  1080.0           304.55            10.787           5.1683          19.288
  1331.5           305.26            8.8552           6.9083          18.110
```

Fortran Code

```
integer, parameter :: nmax = 10000
integer n, ios
real p0, t0, q0
real, dimension(nmax) :: q, t, q, u, v
open(10,file='data.ascii',form='formatted')
read(10,*) p0, t0, q0
do n = 1,nmax
    read(10,*,iostat=ios)  z(n), t(n), q(n), u(n), v(n)
    if( ios == -1 ) exit
enddo
close(10)
```

Python Code

```
f = open("data.ascii", "r")
p0, t0, q0 = f.readline()
d = [float(x) for x in f.read().split()]
z, t, q, u, v = d[0::5],d[1::5],d[2::5],d[3::5],d[4::5]
f.close()
```

Why Python?

- Python uses natural syntax - most Fortran/C programmers would understand code structure upon reading it - to me it looks like a combo of Fortran + CSH.....
- Includes OOP, dynamic typing, regular expressions, etc.
- Strong community support of numerical operations (Numeric, Numpy, Numarray, SciPy)
- Interface software to combine Python with Fortran / C / C++ exists (F2PY & SWIG)
- netCDF & HDF5 interfaces exist (HDF5 => PyTABLES!)
- Visualization interfaces (VTK, Matlibplot, NCAR graphics)
- Large user community - commercial development, etc.

Some Python Things

(to know...)

- Our DA group at NSSL uses Python to
  - wrap NCAR command language scripts for command line and GUI access
  - create command line interfaces to our Fortran NAMELIST FILES: (pyRun)
  - create machine independent "Makefiles": pyMake
- Python has many add-on packages
  - PyNGL (Python interface to NCAR graphics) (also PyNCL)
  - Scientific Python (not SciPy) - netCDF interface
  - PyTables - interface to HDf5
  - Numerical Python (linear solvers, stats, matrix computation) - MatLab-like replacement
  - PyVtk - interface to VTK libraries for 3D viz
- About anything you could want, really...

PyMake combining Python and Make

```
#=================================================================
# main script

# Figure out which machine we are on

machine = platform.machine()  # i386, ppc, etc.
system  = platform.system()   # Darwin, Linux,
compiler = ""

print
print "Compiler python script for COMMAS"
print
print "Machine is ", machine
print "System is  ", system
print
#-----------------------------------------------------------------
#
# Mac Intel ifort Compiler definitions

if machine == 'i386' and system == 'Darwin' and compiler != "g95":

        F90         = "'ifort'"
        F77         = "'ifort'"
#-----------------------------------------------------------------
#
# Mac Intel G95 Compiler definitions

if machine == 'i386' and system == 'Darwin' and compiler == "g95":

        F90         = "'g95'"
        F77         = "'g95'"
#-----------------------------------------------------------------
#
# Mac PPC Compiler definitions

if machine == 'Power Macintosh' and system == 'Darwin':

        F90         = "'xlf90 -qsuffix=f=f90 -qsuffix=cpp=F90'"
        F77         = "'xlf -qfixed=132'"
#-----------------------------------------------------------------
#
# SGI Altix Compiler definitions

if machine == 'ia64' and system == 'Linux':
        F90         = "'ifort'"
        F77         = "'ifort'"
```

```
#-----------------------------------------------------------------------
# Makefile Macros....
        FCOMPILE = 'F90=' + F90 + ' F77=' + F77 + ' FPP=' + FPP + ' CC=' + CC + ' FIXED=' + FIXED + ' FREE=' + FREE   \
                 + ' OPT_F90=' + OPT_F90 + ' OPT_F77=' + OPT_F77 + ' SMP=' + SMP + ' PAR=' + ' CFLAGS=' + CFLAGS \
                 + ' LDR=' + LDR +  ' LOPT=' + LOPT + ' LIB=' + LIB                                              \
                 + ' NETCDF_LIB=' + NETCDF_LIB + ' NETCDF_INC=' + NETCDF_INC                                     \
                 + ' AR=' + AR + ' RANLIB=' + RANLIB

#-----------------------------------------------------------------------
# COMPILE DICTIONARY
#               name            location     target          compile options
#-----------------------------------------------------------------------
MAKE_DICT= {'fort_lib':     ['./Model', 'fortran',      FCOMPILE], \
            'commas':       ['./Model', 'commas' ,      FCOMPILE], \
            'commas_init': ['./Model', 'commas_init', FCOMPILE], \
            'com2v5d':      ['./Model', 'com2v5d',      FCOMPILE], \
            'filter':       ['./Filter','enkf',         FCOMPILE]}
#-----------------------------------------------------------------------
# Process command lines and compile code

if sys.argv[1] == 'all':
        targets = MAKE_DICT.keys()
        for key in targets:
                tmp = MAKE_DICT[key]
                os.system("rm " + tmp[1])
                ret = makeit(current_dir,tmp[0],tmp[1],tmp[2])

if MAKE_DICT.has_key(sys.argv[1]):
        tmp = MAKE_DICT[sys.argv[1]]
        os.system("rm " + tmp[1])
        ret = makeit(current_dir,tmp[0],tmp[1],tmp[2])

if sys.argv[1] == 'help':

        print "=============================================================================================="
        print "                        Python Compile Script for PyEnCOMMAS "
        print
        print "     pymake all  --> To create a running version of PyEnCOMMAS: "
        print "                     this creates the commas solver library and python interface routines "
        print "                     so one can run the scripts. "
        print
        print "     pymake com2v5d --> To create the COMMAS to Vis5D conversion program"
        print
        print "     pymake clean  --> cleans up the directory, removing object and module files  "
        print
        print "=============================================================================================="
```

NOAA/National Severe Storms Lab

PyRun: Controling fortran namelists

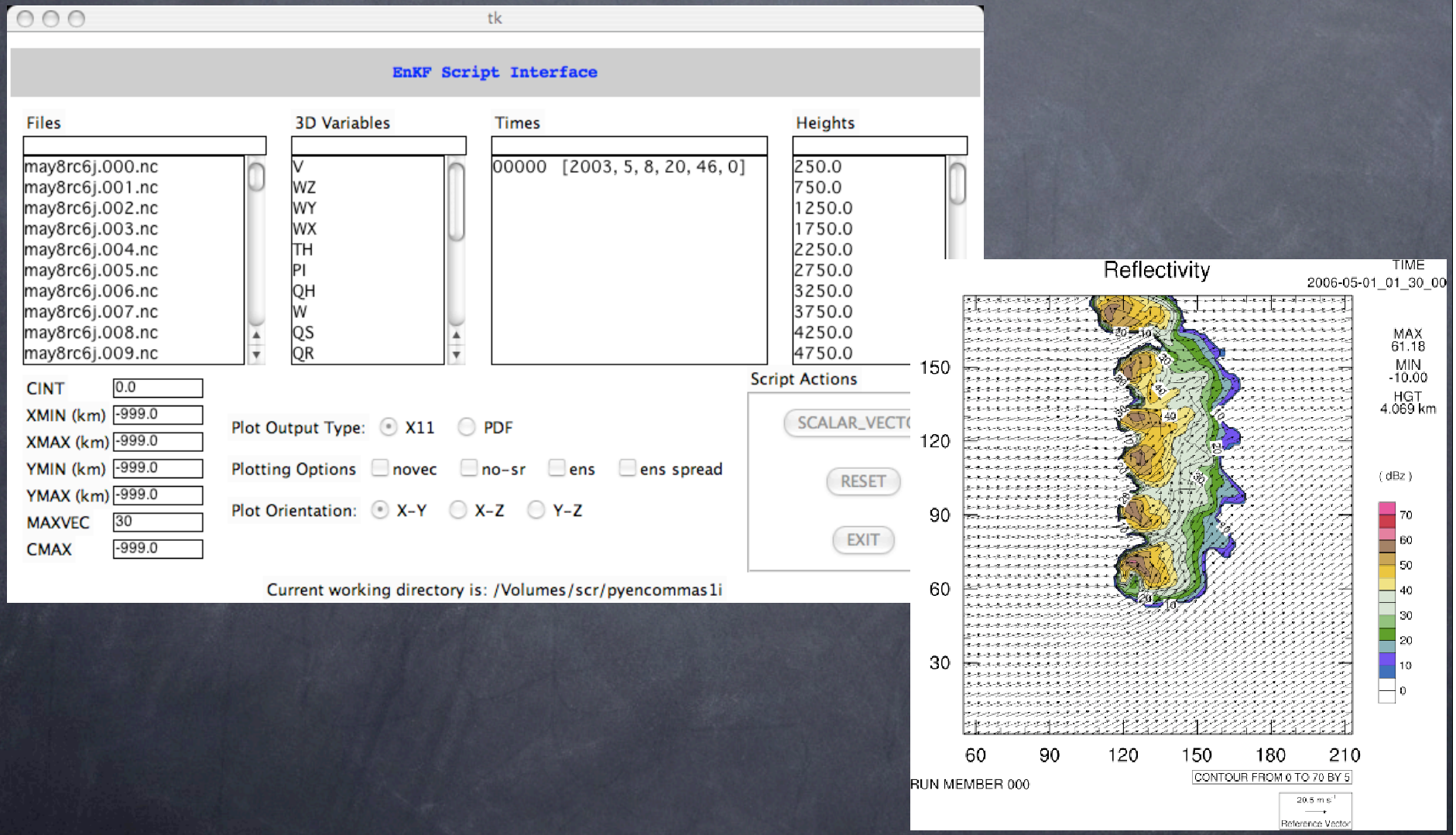
```
#!/usr/bin/env python
#
# This is a python script to set up and run COMMAS06
# type "pyrun -h" or pyrun --help for options
# The main python program is at the bottom of the file
# Python module imports

from optparse import OptionParser
import os, string
#
#
# These are DEFAULT RUN_PREFIX, RUN_MEMBER, RUN_START, and RUN_STOP definitions
#   They can be changed on the command line (type "pyrun -h" for more information)
#   NOTE:  These variables need to be defined as STRINGS (put things in quotes!)

RUN_PREFIX = "test2d"
RUN_MEMBER = "0"
RUN_START  = "-1"
RUN_STOP   = "7200"
RUN_DT     = "10"
RUN_MICRO  = "LFO"
RUN_SOUND  = './may20.sound'

# Create the input text stream used to create the namelist files

namelist_text = """
!===============================================================================
!==========                                                    ==========
!==========                                                    ==========
!==========                NAMELIST:  3D.RUN
!==========                                                    ==========
!==========                                                    ==========
!===============================================================================
&run
     member =   RUN_MEMBER,   ! member number name (number from 000 to 999)
     start =    RUN_START,    ! start    : -1 to start from time 0, otherwise must be a
                              !            history time in the netcdf file
    stop  =     RUN_STOP,     ! stop     : time (seconds) of end of simulation
    ugrid =       10.0,       ! ugrid    : grid motion in X direction
    vgrid =        0.0,       ! vgrid    : grid motion in Y direction
    tprint =       30,        ! tprint   : interval (seconds) for max/min print out
  thistory =       300,       ! thistory : interval (seconds) for history/restart dumps,
                              !            the model is restarted from these history dumps
  tvis5d =         60,        ! tvis5d   : interval (seconds) for vis5d output
/

                     .....
                     .....

                     300 more lines!!
```

```
#-------------------------------------------------------------------------
# Main program for testing...
#
if __name__ == "__main__":
    print "PYRUN:  Sets up model input deck and also can run COMMAS06.  Type 'pyrun -h' / 'pyrun --help' for more info"
    print
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-p",  "--prefix", dest="prefix",   type="string", help="Prefix of run, e.g., 'wk01'")
    parser.add_option("-n",  "--number", dest="number",   type="string", help="Member # of the run, e.g., 0, 1, 2, ..., 11, etc.")
    parser.add_option("-b",  "--begin",  dest="start",    type="string", help="Start time of the run in seconds from initial time, 0, 1200, etc.\
                                                                          A start time of -1 initializes the run")

# Run deck is prefix + "run" suffix

    RUN_FILE = prefix + ".run"

    namelist_text = namelist_text.replace("RUN_PREFIX", prefix)
    namelist_text = namelist_text.replace("RUN_MEMBER", number)
                    .
                    .
                    .
    namelist_text = namelist_text.replace("RUN_MICRO", micro)
    namelist_text = namelist_text.replace("RUN_SOUND", sound_file)

    namelist_file = open(RUN_FILE,"w")
    namelist_file.write(namelist_text)
    namelist_file.close()

    print "INPUT OPTIONS:"
    print "--------------"
    print "Prefix:               ", prefix
    print "Member number:        ", number
    print "Microphysics scheme: ", micro
    print "Start Time:           ", start
    print "Stop  Time:           ", stop
    print "Time Step:            ", dt
    print "Input Deck File:      ", RUN_FILE
    print "Output Run File:      ", output

    if options.run:
        print "COMMAS model will be automatically started"
    else:
        print "NO MODEL RUN REQUESTED"

    if options.delete:
        print "Input deck will be removed after model run is completed"

    if options.delete and options.run:
        print "Starting COMMAS run and will delete run file after completion"
        os.system("commas " + RUN_FILE + " >> " + output + "  2>&1 ; rm " + RUN_FILE + " &")

    elif options.run:
        print "Starting COMMAS run ....  "
        CMD = "commas " + RUN_FILE + " >> " + output + " 2>&1 &"
        print CMD
        os.system(CMD)
```

NOAA/National Severe Storms Lab

SV_GUI.py Interface:

Python + NCL



NOAA/National Severe Storms Lab

Numerical Weather Prediction in 2006

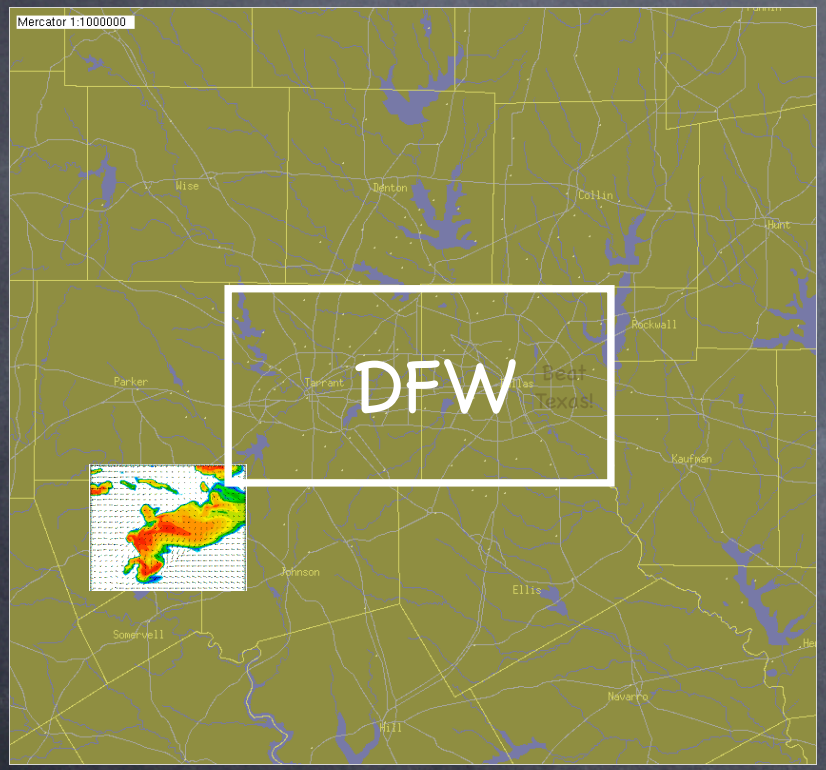
- Numerical weather prediction is the process where the atmosphere fluid equations (a set of PDE's) are discretized on the globe, observations are used to initialize the dependent variables, and the discrete equations are then integrated forward in time to create a weather forecast
- Problem is inherently probabilistic - especially at small scales
- Exponential growth in computational power now permit probabilistic approaches to NWP problem

Numerical Weather Prediction

- 30 years ago: forecast was a single model run whose grid could only resolve large scale storm features (e.g. the Low's)
- Next 20 years was spent increasing resolution and improving physical processes in a single forecast mode.
- Today instead of a single forecast, an ensemble of weather forecasts (10-100 simulations) are now used to produce a forecast that explicitly estimates forecast uncertainty.
- The ensemble information is also useful for incorporating observations: Data assimilation of observations via a technique known as the Ensemble Kalman Filter (EnKF).
- Now we are talking about predicting individual convective storms (like the OKC 3 May tornadic storm...)

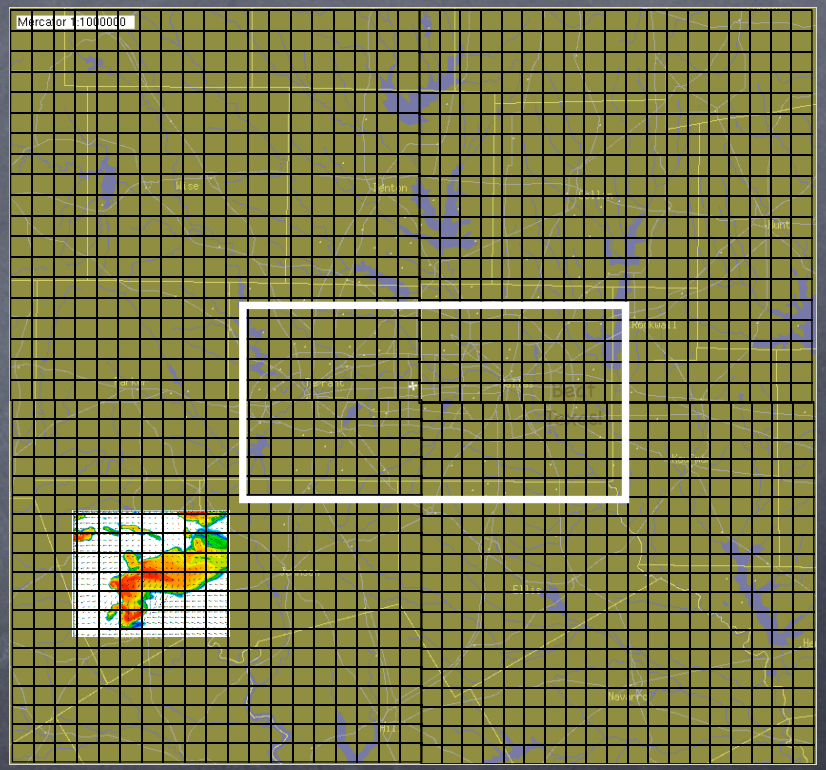
Storm-scale Numerical Weather Prediction?

1975



SINGLE (1) NWP Grid Point ($\Delta x \sim 200$ km)
7 vertical levels

2005



NWP Grid ($\Delta x \sim 4$ km)
70 vertical levels

A factor of ~25,000 in resolution (3D), and a ~10^6 increase in CPU cost
One hour of NWP model computer time today would require > 1 year to run on the 1975 computer!

NOAA/National Severe Storms Lab

Storm-scale NWP: What do you need?

- Weather prediction model (the forecast model) the predicts the weather on scales of ~ 1 km.
- Radar observations: Doppler velocity and reflectivity from the WSR-88D
- Data Assimilation: An algorithm whereby 4D observations are used to create the initial conditions for a geophysical forecast model.

  Ensemble Kalman filter: an DA algorithm that takes as input an ensemble of 3D forecasted weather fields (wind, pressure, rain, etc.) and the radar observations such that at the end, the model data match the radar data in a least squares sense.

What do you now have to deal with?

- Instead of 1 model forecast to initialize: now need 100's initializations
- Instead of running 1 forecast: now run 100's forecasts
- Instead of 1 model forecast output files: now have 100's
- Instead of restarting the model every hour: now need to do it every minute
- Instead of plotting 50 forecast fields: now have 1000's of fields to plot

How to manage THIS?

- Use Python to manage via an OOP + database approach
- Python classes are created to help set up the files, initialize, and start and stop the Fortran executables.
- Python dictionaries are used to manage the parameter lists that are needed for the observational data files and parameters, the model runs, and the EnKF algorithm.
- Pickling of Python objects is used to create persistence.
- Python "Glues" everything together and makes this manageable.

Python is the "Glue"

- Create Python classes to "hide" all the internal gobbly-gook...
- Three class objects
  - pyDART: observation class
  - pyEnKF: Kalman filter class
  - pyEnsemble: forecast model class
- Each python class has its own data and methods for executing operations needed
- Run forecast model, dump observations for filter, create input files and namelists for Fortran codes, etc.

run_ENKF.py Python Script Outline

reads user command input parameters: ensemble file name, etc.

reads parameters for model ensemble: file prefix name, # of members, date and time of integration, etc.

reads parameters for Kalman filter: what variables to be adjusted by the assimilation, observation bias and variance, etc.

read observation files: Determine what the integration blocks based on the availability of the radar observations e.g. Time = [22:08-22:10, 22:10-22:16, 22:16-22:17]

For each block in Time:

what time is it?
are there observations?
Yes? THEN
  Create observation header file for enkf
  Call Kalman filter
No? THEN
  for each member in ensemble, create NAMELIST file for params
  run model and integrate each member to next time in TIME

Examples of our code....

```
#--------------------------------------------------------------------------------
#
# Command line arguments

usage  = "usage: %prog [options] arg"
parser = OptionParser(usage)
parser.add_option("-f",  "--file",   dest="file",  type="string", help="Name of run and/or
ensemble object file (e.g., may20.exp")

(options, args) = parser.parse_args()

if options.file == None:
      print
      parser.print_help()
      print
      print "ERROR:  configuration file not defined...EXITING"
      print
      sys.exit(0)

#--------------------------------------------------------------------------------
#
# Simulation run parameters

experiment = ReadEnsemble(options.file)

run_dict  = param.read(experiment.config_file, 'run_ensemble')
init_dict = param.read(experiment.config_file, 'init_background_dict')
enkf_dict = param.read(experiment.config_file, 'enkf_dict')

trestart  = run_dict["trestart"]
thistory  = run_dict["thistory"]
tvis5d    = run_dict["tvis5d"]
tprint    = run_dict["tprint"]
ugrid     = run_dict["ugrid"]
vgrid     = run_dict["vgrid"]
```

Process command line arguments

Remind user how to run code

Read in python pickle object with ensemble info

Extract needed parameters out of run dictionary

Time Integration Loop

```
# START time loop
while time < stop:          # Find the next observation time that is >= the current time.

      if ObTimeSec[TimeIndex] > time:
            td        = ObTimeSec[TimeIndex] - time
            NextTime = int(round(time + dt*round( td / dt)))
            print 'RUN_DARTosse:  TimeIndex = ', TimeIndex
            print 'RUN_DARTosse:  ObTimeSec = ', ObTimeSec[TimeIndex]
            print 'RUN_DARTosse:  Time      = ', time
            print 'RUN_DARTosse:  NextTime  = ', NextTime

# Integrate ensemble members to next observation time.

            print 'RUN_DARTosse:  CALLING ThreadTimeStep at time  ',NextTime

            if run_model:

experiment.SetRunParams(time,NextTime,trestart,thistory,tvis5d,tprint,ugrid,vgrid)
                  experiment.ThreadTimeStep(nthreads=nthreads)

            print 'RUN_DARTosse:  COMPLETED ThreadTimeStep at time:    ',NextTime
      else:
            NextTime = time

# Assimilate observations

      for x in ObFiles:             # Search file list..
            if verbose:
                  print 'RUN_DARTosse: Name of observation file  ',x
            if x.find(str(ObTimeSec[TimeIndex])) != -1:
                  utc   = ObTime[TimeIndex]
                  strin = "%s  %s  %s  %s  %s  %s  %s '%s'" %
(ObFormat[TimeIndex],utc[0],utc[1],utc[2],utc[3],utc[4],utc[5],x)
                  if verbose:
                        print
                        print 'RUN_DARTosse: command written to enkf obfile list  ', strin
                  ofile = open(ObFileList, 'w+')
                  ofile.write(strin)
                  ofile.close()
                  cmd = 'enkf ' + str(NextTime) + ' ' + ObFileList + ' ' + ObTableFile +  ' ' +
TrueState[TimeIndex] + ' ' + str(nxyz3dtruth)
                  print
                  print 'RUN_DARTosse:  EnKF being called:  ',cmd
                  print
                  if run_enkf:
                        os.system(cmd)
                  print 'RUN_DARTosse:  COMPLETED ENKF for data file ',x,' at time:  ',NextTime
                  print
      print 'RUN_DARTosse:  COMPLETED ENKF for all data files at time ',NextTime

# Increment time and observation file time indices

      time = NextTime               # Set time to NextTime

      TimeIndex = TimeIndex + 1     # Increment TimeIndex (for ObFiles) by 1
      print "RUN_DARTosse:  Integration has been completed through ",time
#END TIME INTEGRATE LOOP
```

Model object method for setting model parameters

Model object method for running fcst models simultaneously (parallel)

All this string processing would really, really hurt in Fortran. Don't try this at home....

Comments

- Python manages all of the information to make these program interact. In this model, Fortran codes are still separate and act like Unix shell commands executed with in the Python program.
- Is all this doable in Fortran: Yes, very painfully
- How about CSH? Yes, perhaps as painfully
- Perl? Ruby? Sure - because at this point the Fortran algorithms and Python code are separated.
- Can we integrate things further (and do we want to?)

Should we go further....?

- F2PY can wed F77/F95 code to Python such that Fortran modules can be loaded into the interpreter.
- Advantages:
  - Removes the need for passing information through files which is messy
  - Can use python to store metadata about Fortran variables, again doable but awkward in F95
  - Python has excellent File I/O modules - reading and writing data to netCDF/HDF4/5 in Python is far simpler in Python than Fortran
  - OOP programming in Python is far easier than OOP programing in F95 (I have tried...)

Should we go further....?

- Disadvantages:
  - Integrating Python/Fortran generates code that is much more machine dependent code (F2PY works on 32/64 bit, but there are a few issues)
  - Data needs to be stored in row-major order in Python - doable. Differences between Numarray/Numpy/Numeric can introduce conversion overhead
  - Python 2.5 is 64 bit, but not all needed OSS code is 64 bit friendly. Our EnKF application needs large memory (> 4 GB). Have not checked on this since Python 2.3
- Our experience: If problem is I/O intensive and big memory, better off leveraging existing code and "gluing" the various Fortran applications together with Python.

Final Comments

- "PyEnCOMMAS" application developed and run on Mac (Intel & PPC), Intel Linux, and 64P SGI Altix.
- 6 member research "group": NSSL, NCAR, Univ of IL.
  - most knew only F90/CSH.
  - Learning Python was relatively easy
  - OOP concepts somewhat harder
- All believe that effort was worthwhile - management of EnKF application is much easier task
- Relatively few cross-platform issues (mostly plotting crap)
- Copy of this talk and other Python info available at:

  http://www.nssl.noaa.gov/users/lwicker/public_html/

Suggested Reading

- Python: A Visual Quickstart Guide by Chris Fehley - my "bible" for Python programing, mostly 300+ pages of examples of stuff you forget.
- Python Scripting for Computational Science, Hans P. Langtengen - a great book for seeing all the things one can do using Python in computation.
- Joel on Software by Joel Spolsky - a must for anyone who writes/manages large amounts of code - plus its very funny and gives an inside look at working at Microserf and other companies during the internet boom of the late-1990's