# 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

# 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!)
  - How to use a word processor
  - How to run a job on a big computer (JCL...ack..)
- **Researchers / Students in 2005 need to know:**
  - several languages (F95, C++, csh, Python, MatLab, etc...)
  - parallel programming
  - visualization
  - web programming
  - grid computing, data base management, etc, etc....

# Compiled Languages

- As in ... F77, F95, F2000, C++, C, Ada, etc ...
- Create fastest executing code
- Are "traditional" development tools, e.g. => taught
- Development cycle:
  - write/compile/link/run ...
  - debug/compile/link/run ... etc
- Access to Unix filesystem, files, URL's awkward..
- Some compiled languages (F77, C ?) do not promote the application of good software practices like OOP, modular code, etc.

# Interpreted (Scripting) Languages

- As in ...
  - Csh, Perl, Python, Ruby, Tcl, 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 and their built-in high-level constructs.

## Then vs Now?

- Software development is part of everyday scientific work (the computer is now the lab...)
- Increased computer capability (CPU, Memory, Disk) can run interpreted languages faster than compiled codes were run 10-15 years ago
- 1990's saw explosive development of scripting languages (Perl, Python, Ruby, Tcl++) and OSS software and operating systems (Linux)
- Scripts enable the masses to attempt GUI development
- OSS / WWW / Information / Examples  etc!

# Advantages of Using Interpreted Languages

- Programs are generally written at a higher level
- Modules can include both functions and main drivers => easier development & testing
- One can generate and execute code from the "inside"
- Development 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"
- File I/O, File I/O, File I/O!
- 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

```fortran
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

```python
f = open("data.ascii", "r")
p0, t0, q0 = f.readline()
d = 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?

- Language uses natural syntax - most Fortran/C programmers would understand code structure upon reading it - looks like 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 ( PyTABLES!)
- Visualization interfaces (VTK, Matlibplot, NCAR graphics)
- Large user community - commercial development, etc.

# Numerical Weather Prediction in 2006

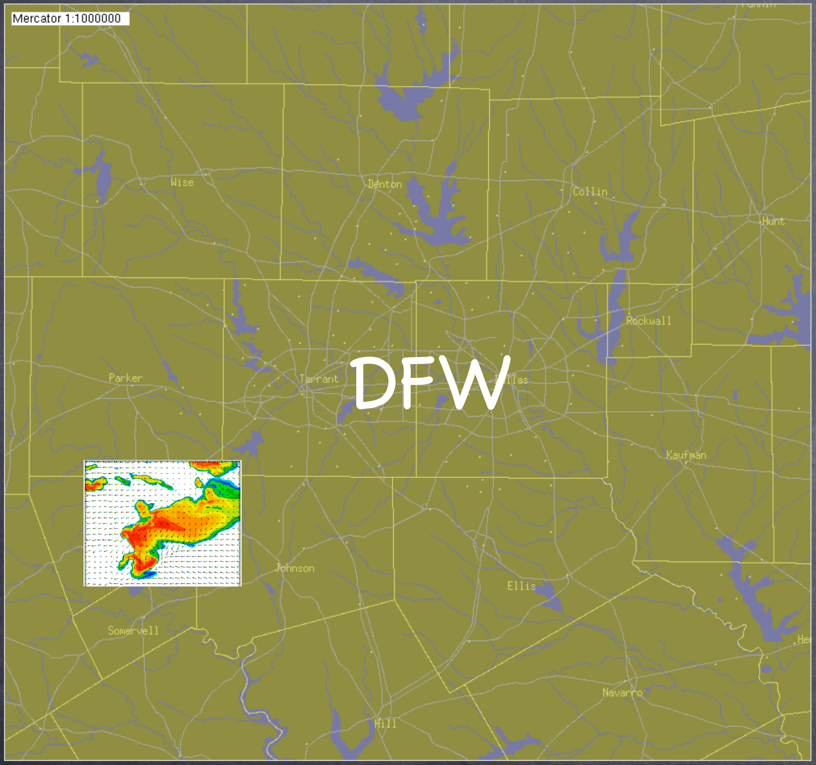
- Numerical weather prediction is the process where the atmosphere fluid equations (a set of PDE's) are discretized on the globe, observations are used to initialize the dependent variables, and the discrete equations are then integrated forward in time to create a weather forecast
- Problem is inherently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

# Numerical Weather Prediction in 2006

- Instead of a single forecast, an ensemble of weather forecasts (10-100 simulations) are now used to produce a forecast that explicity estimates forecast uncertainty.
- The ensemble is also useful for incorporating observations: A process known as data assimilation.
- 30 years ago prediction was barely able to resolve low and high pressure centers
- Now we are talking about resolving individual convective storms (like the OKC 3 May tornadic storm...)

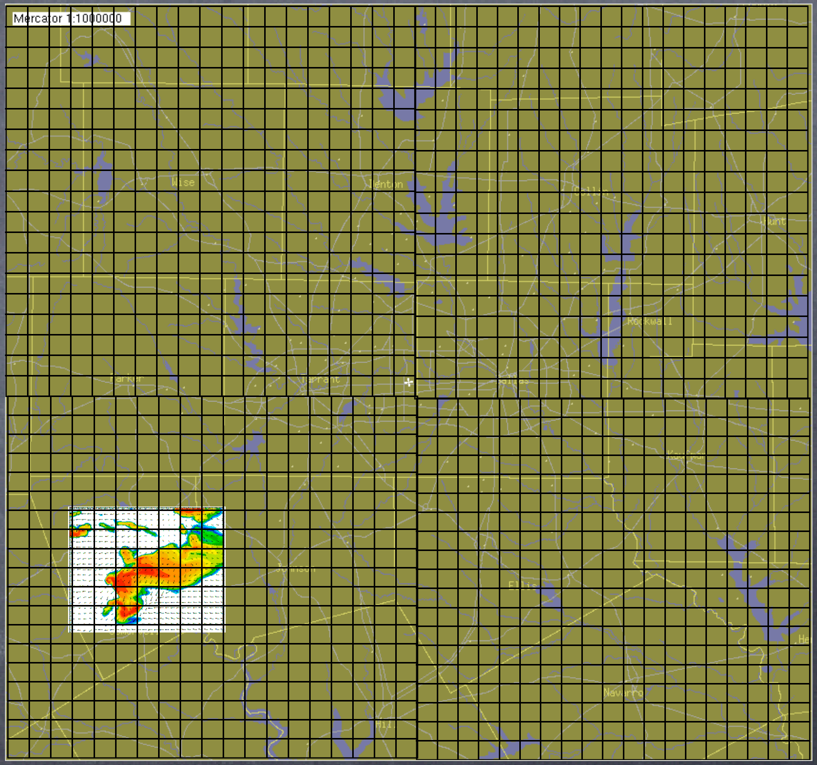
# Storm-scale Numerical Weather Prediction?

1975



**SINGLE** LFM Grid Point ($\Delta x \sim 190$ km)
7 vertical levels

2005



WRF Grid ($\Delta x \sim 4$ km)
50 vertical levels

**A ~ $10^6$x increase in CPU!**
**One hour of WRF computer time today would require > 4 years to run on the 1975 computer!**

NOAA/National Severe Storms Lab

# Storm-scale NWP needs storm-scale data: Radar data!

- Terminology
  - Weather prediction model (the forecast model) predicts the weather on scales of ~ 1 km.
  - Data assimilation: An algorithm whereby observations from the atmosphere are used to create the initial conditions for the forecast model.
  - Radar observations: Doppler velocity and reflectivity from the WSR-88D
  - Kalman filter: an algorithm that takes as input an ensemble of 3D forecasted weather fields (wind, pressure, rain, etc.) and from them create mathematical relationships between the model fields and the radar observations such that at the end, the model data match the radar data in some least squares sense.
    - Ensemble of model forecasts is used to approximate the evolution of the covariance matrix from "classic" Kalman filter.
    - Cannot evolve covariance matrix directly ( deg. of freedom~$10^8$)

# How to Deal with this?

**The Problem:** Manage the input radar data streams, initializating then starting/stopping for each radar sweep 50-100 model runs, controlling the Kalman filter operations, dealing with about 1000 parameters to track (and change) used by the models and the Kalman filter, and the statistical output from all of this..

- Essentially this is managed via an OOP + database approach (this problem can generate thousands of files...)
- Model and EnKF information (filenames, data files), run parameters (time steps, Kalman filter coefficients, error variances, etc.) are stored in a Python dictionary and stored to a file via pickling.
- "Glue" fortran codes together using Python classes

# 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 namelists for fortran, etc.

## run_ENKF.py Python Script Outline

**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 looks like 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
```

# Example 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 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

- At this point - Python is simply used as a string/shell/command processor. Fortran codes are the "executables" that Python controls.
- 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 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 - messy
  - Can use python to store metadata about Fortran variables - messy in F95
  - Python has excellent File I/O modules - reading and writing data to netCDF/HDFx in Python is far simpler in code than Fortran
  - OOP programming in Python is far easier than OOP programing in F95 (I have tried...)

## Should we go further....?

- Disadvantages:
  - 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, but creates conversion problems if Python is used for the I/O
  - Python 2.5 is now 64 bit, but not all needed OSS code is 64 bit friendly. Our EnKF application needs large memory ( > 4 GB)
- Bottom line: 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) and 64P SGI Altix.
- 6 people in NSSL research group
  - 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
- Few cross-platform issues (mostly plotting crap)
- copy of talk and other Python info available at:

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