NUMSCONS: GETTING CONTROL OF NUMPY BUILD SYSTEM BACK

A NEW BUILD SYSTEM FOR NUMPY, SCIPY AND COMPLEX C/ FORTRAN EXTENSIONS

What is the tutorial about ?

- Rationales and goals for a new build system, examples
- Limitation of distutils: why using scons ?
- Design of numscons
- How to use numscons:
  - what a C/Fortran extension developer should know
  - what a core numpy/scipy developer should know

What’s a build system ?

- How to get from sources to a built software
  - platform specific detection
  - compilation and link step
  - customization
- NOT about installation or deployment issues (eggs, inter-package dependencies, etc...)

Why bother ?

For our users

- User-friendliness:
    - build is often the first contact with the user
    - people want to play with build flags, compiler, etc...

For us, developers

- New and improved features:
  - better dependency handling
  - fine-grained control of build options
  - better configuration stage: easier library and platform dependencies handling
  - new features: ctypes extension, etc...
- Easy to understand: any numpy/scipy developer should be able to “touch” it.

numscons today

- version 0.9.1 (available in pypi, code on launchpad)
- Build numpy and scipy on
  - Mac OS X (gcc)
  - Linux (gcc/Intel/Sun)
  - Open Solaris (gcc/Sun)
  - Windows (mingw, Visual 2003/2005/2008)
- Support for MKL, Sunperf, ATLAS, FFTW2/3, Accelerate/Veclib

Examples

- Building a numpy C extension:

```
from numscons import GetNumpyEnvironment
env = GetNumpyEnvironment(ARGUMENTS)
env.NumpyPythonExtension("spam", source =
["spam.c"])
```

- Finding a dependency on libsndfile:

```
from numscons import GetNumpyEnvironment
env = GetNumpyEnvironment(ARGUMENTS)
config = env.NumpyConfigure()
config.NumpyCheckLibAndHeader('sndfile',
'sf_open', 'sndfile.h')
config.Finish()
```

Examples (2)

* Building quickly for debugging purpose:

```
CFLAGS="-DDEBUG -Wall -W -g" python setup.py build
```

* Building on with 4 cores:

```
python setup.py scons --jobs 4
```

* Building ala kbuild:

```
python setup.py scons --silent=1
```

```
PYEXTCC      build/scons/numpy/random/mtrand/mtrand.c
PYEXTCC      build/scons/numpy/random/mtrand/randomkit.c
PYEXTCC      build/scons/numpy/random/mtrand/initarray.c
PYEXTCC      build/scons/numpy/random/mtrand/distributions.c
PYEXTLINK    build/scons/numpy/random/mtrand/mtrand.os
```

Simple demos

- Basic build
- Parallel build
- Customized build
- Terse output
- Automatic dependencies

Why starting from scratch ?

Current build system

- numpy.distutils:
  - core part of numpy (scipy_core)
  - Handle fortran, blas/lapack detection, etc...
- big: numpy/distutils ~ 10000 loc
- depends on distutils implementation details: effective size of numpy.distutils = size(distutils) + size(numpy.distutils)
- fragile: difficult to modify something without breaking somewhere else.

Main design decisions of numscons

- Use scons for handling low level build issues (dependencies, flags, compiler configuration)
- Simple: ~ 3000 loc
- clear separation between core and customization
- Less magic than distutils, but easier to customize (for users and developers)
- Hardcode as little as possible, detect platform-specific features at runtime (fortran, etc...)

why scons ?

What is scons ?

- a make replacement in python

- From scons website:

SCONS IS AN OPEN SOURCE SOFTWARE CONSTRUCTION TOOL—THAT IS, A NEXT-GENERATION BUILD TOOL. THINK OF SCONS AS AN IMPROVED, CROSS-PLATFORM SUBSTITUTE FOR THE CLASSIC `MAKE` UTILITY WITH INTEGRATED FUNCTIONALITY SIMILAR TO `AUTOCONF/AUTOMAKE`.

scons scripts are in python

* Almost any python code is legal in scons scripts
* scons scripts are declarative
* access to python stdlib and numpy.distutils is available

scons has a configuration system

* scons has a basic configuration system ala autoconf
* Can check for type, their size, functions, headers, declaration
* Can be extended (but ugly: one of the worse part of scons IMHO)

Targets customization

- Each target can be built differently
- Compilation flags, extensions, etc... can be customized in a really fine-grained manner (per file if wanted)

Scons is extensible

- scons has many unpythonic aspects to it (in python 1.5.2., use of apply, etc...)
- But:
    - scons has a good manual
    - can be extended relatively easily: easy things are easy, complicated things can be hairy, but still possible
    - is relatively well tested
    - Good and responsive community
    - Are opened to discussion and improvements

Scons users

- Users of scons:
  - scons is the build system for doom3 on Linux
  - scons is used for major products of Vmware
  - ardour2 (Direct-to-disk audio software) uses scons, blender
  - Generally popular in the gaming open source scene (windows support)

Core scons concepts

Builders

- Builder: scons concept to build things
- Builder for object code, program, shared library, etc...

```
SharedLibrary("foo.c")   # Build a shared library
StaticLibrary("bar.c")   # Build a static library
Program("foobar.c")      # Build a program
```

- Custom builders possible

Builders customization

- Each builder can be given an arbitrary set of arguments

```
env = Environment()
# Add -DFOO on posix
env.Append(CPPDEFINES = ["FOO"])
# *Override* -DFOO to -DBAR
env.Object("foo", source = "foo.c", CPPDEFINES = ["BAR"])
env.Append(CPPDEFINES = ["BAR"])
env.Object("bar", source = "bar.c")
```

- Output:

```
gcc -o bar.o -c -DFOO -DBAR bar.c
gcc -o foo.o -c -DBAR foo.c
```

Dependency handling

- Targets builds from dependencies by walking through a DAG (like make)
- But dependencies are automatically inferred by scanning source code (implicit dependency)
- md5-based system to decide whether a target has to be rebuilt

Automatic dependency handling

```
# SIMPLE MAKEFILE
FOO.O: FOO.C
    $(CC) -C FOO.C -O FOO.O
```

```
#include "foo.h"
int foo()
{
   return 0;
}
```

- What if foo.h is changed ?
- scons scans automatically foo.c to find foo.c
- Scons uses scanners to scan source files
- You can add your own scanners (numscons: scanner for f2py <%include%>)

Scons signature system

- How to determine whether one needs to rebuild a target
- make uses time-stamps to determine whether a target is up to date
- scons uses md5: more reliable (NFS, time clock skew); md5 are put in a signature db file
- But scons also keeps the signature of the command lines, options, etc...: if the C compiler changes, scons will rebuild C code, if a library changes (ATLAS vs MKL), only link step will change, etc...
- Can be customized

Node concept

- At the DAG level, everything is a node
- Every builder returns a list of nodes:

```
foo = Object("foo.c")
bar = Object("bar.c")
# This is not portable (.obj on windows)
Program("foobar", source = ["foo.o", "bar.o"])
# But this is
Program("foobar", source = [foo, bar])
```

- Internally, in scons, everything is a node, but you can generally ignore the distinction between e.g. a file and its node
- (only needed for advanced use of scons/numscons)

Environments

- Global object to keep configurations

```
env = Environment()

env2 = env.Clone()
env.Append(CFLAGS = "-O2")

env.Program("foo.c")
env2.Program("bar.c")
```

- Each environment has builders attached to it
- Builders wo environments use a default environment

scons configure system

- If you depend on libfoo, how to detect it on the system ?

```
env = Environment()

config = env.Configure()
config.CheckLib("sndfile", "sf_open", "#include
<sndfile.h>")
config.Finish()
```

- Can be extended, but non trivial tests are really difficult

Scons tools

- Scons concept to handle compilers, linkers, etc...
- A tool is a python module with two public methods called by scons
- A tool set up environment values of an environment
- A new compiler can be supported by a scons tool
- Worst part of scons design (configure/tools problems are somewhat linked): tools are not reentrant, fragile, and not reusable.

More about scons

- man scons is complete and readable
- scons manual available on http://www.scons.org
- wiki with many examples + Mailing list
- Non trivial projects using numscons will require scons knowledge

scons for numpy ?

- Distutils revamp features list: (By David M Cooke)

  - better dependency handling
  - make it easier to use a specific compiler or compiler options.
  - allow `.c` files to specify what options they should/shouldn't be compiled with (such as using `-O1` when optimization screws up, or not using `-Wall` for `.c` made from Pyrex files)
  - simplify `system_info` so that adding checks for libraries, etc., is easier
  - a more "pluggable" architecture: adding source file generators (such as Pyrex or SWIG) should be easy.
  - better `setuptools` support
  - more as I think of them…

- scons solve almost all the above “for free”
- Extending scons to build python extensions and fortran
- Instead of “fixing” distutils, I improve scons....

Scons for numpy ?

* scons solve almost all the distutils shortcomings "for free"
* But scons has limited/no support for
  * python extensions
  * fortran
* Instead of "fixing" distutils, I improve scons (significant patches included upstream)

numscons

- A new distutils command which drives a scons process
- numscons: a set of extensions around scons to build numpy and scipy

numscons: architectural choices

Goals

- **Simplicity** (for numscons users and numscons developers)
- Use autoconf philosophy for platform specifics: **do not depend on versions**, but **test capabilities**
- Less magic than distutils, but **easier to customize** (mere-mortals should be able to add new compiler, customize flags)

Architecture

SETUP.PY

```
def configuration(parent_package='',top_path=None):
    from numpy.distutils.misc_util import Configuration
    config = Configuration('foo',parent_package,top_path)
    config.add_sconscript('SConstruct')
    return config
```

DISTUTILS PROCESS

CALL SCONS COMMAND WITH ARGUMENT

SCONSTRUCT FILE

```
from numscons import GetNumpyEnvironment
env = GetNumpyEnvironment(ARGUMENTS)

# Now one can do whatever we could with scons, and more...
env.NumpyPythonExtension("spam", source = ["spam.c"])
```

SCONS PROCESS

Architecture

- Add a scons command to distutils:
  - scons scripts are added through setup.py files
  - options passed to scons on the command line
- scons scripts get their environment through a numscons function GetNumpyEnvironment
- After this call, like being in scons + numscons add-in
- Not easy to give information from scons back to distutils

subpackages

* numpy and scipy: collection of subpackages
* Difficult problem from a build POV:
  * build and configuration can be run anywhere in the tree
* Two possibilities:
  * recursive scons: how to do configuration (recursive configuration ?), build directory problem
  * calling scons for every subpackage: simpler; current numscons design

subpackages (2)

- Calling scons for every subpackage:
  - scons process called many times (scipy ~ 20 subpackages)
  - scons + numscons + numpy import everytime
  - Consequence on some design decisions: numscons optimizes its own import time heavily
- Decision made at the beginning: I still think it was the right one given the constraints (no modification of the source tree)

Build directory

- distutils put everything in the build directory by default
- numscons put everything in build/scons, and “install” binaries where distutils expects them
    - Uses the VariantDir mechanism of scons
    - Removing build directory: start from scratch (like distutils)
    - In place build works: internally, very easy to change in numscons
    - One could imagine different build directories
- Hopefully, nobody needs to care

Build directory (2)

- VariantDir: difficult to understand
  - Used for build directories (debug vs release built)
  - What it really does: duplicate sources into the variant dir
- From a user POV: mostly transparent, all path are “translated” by scons
- The actual mechanism is fairly complicated, but totally transparent to users, and developers who use numscons.

Numscons organization

* Three fundamental subpackages in numscons namespace
  * numscons.core: set scons from distutils arguments, customize compilers (1000 loc)
  * numscons.checkers: handle blas/lapack/fft (900 loc) and fortran configuration (400 loc)
  * numscons.tools: extra tools (f2py, vs2005/vs2008). Hopefully will mostly go upstream

numscons.core

* GetInitEnvironement:

    1. Initialize a NumpyEnvironment from distutils

    2. Initialize compilers from distutils-passed commands to scons tools name

    3. Customize compilers (given user configuration)

    4. Add custom builders (Python extension, etc...)

* Misc utilities (compiler detection, configuration, etc...)

numscons.checkers

- Blas/lapack checkers: support for sunperf, atlas, mkl, veclib and accelerate
- Two layers: perflib (mkl, sunperf, atlas) and “meta lib” which uses perflib as an implementation
- Use code snippet for testings instead of testing for file existence (more robust w.r.t broken configurations)
- customization from env (MKL=None) and site.cfg handled automatically

numscons.checkers.fortran

* Handle fortran support: do it like autoconf
  * Checkers for C/Fortran support, fortran mangling, etc...
  * Detected at runtime through code snippets: robust to “weird” configurations (icc + sun fortran, gcc + intel fortran, etc...)
  * In theory, should be robust to fortran runtime mismatch (g77-built atlas with gfortran-built scipy)

What’s left to be done

- More work on windows (2.6/3.0 and SxS nightmare)
- Use consistent code style + documentation
- A lot of code in numscons could end up upstream (~ 1/3: visual studio 2003/2005/2008, dlltool/dllwrap)
- For 2.0: getting rid of distutils ?

How to use numscons

As a user

- Basic usage: `python setup.py scons`
- Can be customized from user environment:

```
CFLAGS=”-DDEBUG -g” CC=colorgcc python setup.py scons
```

- site.cfg customization should work

As a developer

Boilerplate

- Three files: setup.py, SConscript and SConstruct
- Setup.py:

```
def configuration(parent_package='',top_path=None):
     from numpy.distutils.misc_util import Configuration
     config = Configuration('pyext',parent_package,top_path)
     config.add_sconscript('SConstruct', source_files =
['hellomodule.c'])
     return config
```

Boilerplate (2)

- SConstruct (always the same)

```
from numscons import GetInitEnvironment
GetInitEnvironment(ARGUMENTS).DistutilsSConscript('SConscript')
```

- SConscript (do the real work)

```
from numscons import GetNumpyEnvironment
env = GetNumpyEnvironment(ARGUMENTS)

env.DistutilsPythonExtension('spam', source = ['hellomodule.c'])
```

Basic task: C extension

- Simple python extension:

```
env.DistutilsPythonExtension("hello", source =
["hellomodule.c"])
```

- Simple numpy extension:

```
env.NumpyPythonExtension("hello", source =
["hellomodule.c"])
```

- Simple numpy extension:

```
env.NumpyCtypes("hello", source =
["hellomodule.c"])
```

Basic configuration

- Checking for header, declaration:

```
config = env.NumpyConfigure()
config.CheckDeclaration("SYS_WAIT")
config.CheckHeader("stdint.h")
config.CheckType("int32_t")
config.Finish()
```

- Everything is logged in package-specific file (config.log)
- Can generate a config.h (config_h argument of NumpyConfigure)

Basic task: dependency

- Your extension depends on library foo, with header foo and function do_foo:

```
config = env.NumpyConfigure()
config.NumpyCheckLibAndHeader("foo", "do_foo", "foo.h", "foo_opt")
config.Finish()
```

LIBRARY

HEADER TO CHECK

SYMBOL TO CHECK

SECTION NAME

- Note: not implemented for ctypes

More advanced tasks

Fortran blas

```
from numscons.checkers.perflib import CheckF77BLAS
config = env.NumpyConfigure()
config.CheckF77BLAS()
config.Finish()

# Now, env has the necessary flags, libs to compiler blas
```

Generating code

- Autoconf-like .in processor:

```
#define FOO1 @SYMBOL1@        #define FOO1 foo
#define FOO2 @SYMBOL2@   →    #define FOO2 bar
```

- Sconscript:

```
# dictionary of symbols : value
env['SUBST_DICT'] = {"@FOO1@": "foo", "@FOO2@": "bar"}
# Generate foo.h from foo.h.in, with expanded
# macro from env["SUBST_DICT"]
env.SubstInFile("foo.h", "foo.h.in")
```

- Note: if SUBST_DICT changes, automatic rebuild

Fortran mangling

- C++ source file:

```
extern "C" void @HELLO@();
int main() {
        @HELLO@();
        return 0;
}
```

- scons script:

```
config = env.NumpyConfigure()
# Detect f77 compiler mangling; set a mangler in env["F77_NAME_MANGLER"] if
# successful
config.CheckF77Mangling()
config.Finish()

# Generate a .cxx file from template with true mangled fortran symbol
env['SUBST_DICT'] = {'@HELLO@' : env['F77_NAME_MANGLER']('hello')}
env.SubstInFile('main.cxx.in')
```

Fortran runtime support

- Linking Fortran with C/C++

```
config = env.NumpyConfigure(custom_tests = {'CheckF77Clib' : CheckF77Clib})
# Automatically detect link flags to link C and C++ with fortran
if not config.CheckF77Clib():
    raise Exception("Could not check F77 runtime, needed for interpolate")
config.Finish()
# At this point, the link flags are automatically added
```

- Output

```
Checking gfortran C compatibility runtime ...-L/usr/local/
gfortran/lib/gcc/i386-apple-darwin8.10.1/4.4.0 -L/usr/local/
gfortran/lib/gcc/i386-apple-darwin8.10.1/4.4.0/../../.. -
lgfortranbegin -lgfortran
```

Detecting optimized libraries

- Testing for perflibs explicitely

```
from numscons.checkers.perflib import
CheckATLAS, CheckAccelerate, CheckMKL,
CheckSunperf
config = env.NumpyConfigure()
config.CheckATLAS(autoadd = 0)
config.CheckMKL(autoadd = 0)
config.CheckAccelerate(autoadd = 0)
config.CheckSunperf(autoadd = 0)
config.Finish()
```

- autoadd option: do not update env

Conclusion

Conclusion

- Numscons is usable today as an alternative build system for most numpy/scipy users/developers needs
- Simple things are easy; complex, customized builds are doable, with scons knowledge
- Should be more extensible and flexible than distutils
- First alpha (public API freeze) planned soon

Questions ?