

---

July 18, 2002

## **Pyfort Reference Manual**

Version 8

Paul F. Dubois (dubois@users.sourceforge.net)

*Advanced Software Technologies Group  
Center for Applied Scientific Computing  
Lawrence Livermore National Laboratory  
Livermore, CA*

---

## Legal Notice

---

Copyright (c) 1999, 2000, 2002. The Regents of the University of California. All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 between the U.S. Department of Energy and The Regents of the University of California for the operation of UC LLNL.

### DISCLAIMER

This software was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

# Table of Contents

1.0	Introducing Pyfort	1
1.1	Pyfort extends Python with Fortran routines	1
1.2	What's New?	1
1.3	Credits	1
1.4	About community	1
2.0	Obtaining and installing Pyfort	1
2.1	How to obtain Pyfort	1
2.2	Other packages you will need	1
2.3	Configuration	1
2.4	Installing Pyfort	2
2.5	Testing	2
3.0	Overview	2
3.1	Module and Project Files	2
3.2	About compiler ids	3
4.0	Command line	3
4.1	Options	3
4.2	Obsolete Options	3
5.0	Creating a Fortran Extension	4
5.1	A Simple Fortran Extension	4
5.2	Preparing the input file	4
5.3	Describing a routine's interface	4
5.4	The Pyfort Project Editor	5
5.5	Compilation and Linking Options	7
5.6	Calling the routine from Python	7
5.7	Array output	7
5.8	The Python signature is determined from the Fortran signature.	7
5.9	Using valued scalars	8
5.10	Packaging with Python Code	8
6.0	Using the wrapper options	9
7.0	Extending Numerical Python with C	9
7.1	Using the C compiler option	9
7.2	C Example	10
8.0	Inter-language communication issues	10
8.1	Representation issues	10
8.2	One-dimensional array issues	10
8.3	Multiple-dimension array storage	11
8.4	Checking rank and extent	11
9.0	Limitations	11
9.1	Explicitly-interfaced routines are not yet supported	11
9.2	Other Limitations and Known Problems	12

10.0	Input grammar	12
10.1	Declaring array sizes	13
11.0	Fortran compilers	13
11.1	Pyfort compiler ids	13
11.2	Compilers supported	13
11.3	Adding a new compiler	14
12.0	Pyfort Project Files	14
13.0	Notes for Specific Platforms	15
13.1	Visual Fortran on Windows	15

**Index 17**

## 1.0 Introducing Pyfort

### 1.1 Pyfort extends Python with Fortran routines

Pyfort is a tool for connecting Fortran routines (and “Fortran-like” C) to Python ([www.python.org](http://www.python.org)) and its Numerical Python array extension ([numpy.sf.net](http://numpy.sf.net)). Pyfort translates an module file that describes the routines you wish to access from Python into a C language source file defining a Python module. Pyfort will also build and install this extension into Python.

Fortran was changed significantly by the introduction of the Fortran 90 standard. We will use the phrase “modern Fortran” to indicate versions of Fortran from Fortran 90 onwards.

Pyfort’s input uses a syntax that is a subset of the modern Fortran syntax for declaring routines and their arguments.

The current release does not yet support modern Fortran’s “explicit-interface” routines. However, the tool was designed with this in mind for a future release.

Pyfort can in most cases also build and install the extension you create.

### 1.2 What’s New?

To find out the latest developments, see the `changes.txt` file in the distribution.

### 1.3 Credits

Pyfort was created using the Earley-algorithm parsing tool “Spark” by J. Aycock, University of Victoria<sup>1</sup>. Paul Dubois implemented Pyfort by adding a new front end to part of a tool written by Brian Yang. He has substantially modified Brian’s wrapper generator to fit a more general application audience. Thanks to Michiel de Hoon, University of Tokyo, for the C-compiler interface.

---

1. John Aycock, “Compiling Little Languages in Python”, Proceedings of the Seventh International Python Conference, Fortec Seminars, Boston, MA, 1998.

## 1.4 About community

The author does not have access to a wide variety of Fortran compilers on different platforms. We’re counting on a community effort to add to Pyfort. Please see Section 11.0 on page 13 for how to contribute.

## 2.0 Obtaining and installing Pyfort

### 2.1 How to obtain Pyfort

The Pyfort project page at SourceForge contains documentation and releases. It is:

<http://sourceforge.net/projects/pyfortran>

Download the latest release from the release area of the page. Unpack the archive and enter the top-level directory that it creates. Examine the file `README` for the latest release information.

### 2.2 Other packages you will need

1. Distutils. [Distutils is a standard part of Python from version 1.6](#). If you have an older Python, first obtain and install Distutils from: <http://www.python.org/sigs/distutils-sig>
2. Numerical Python. Obtain and install Numerical Python from: <http://sourceforge.net/projects/numpy>  
Please see “Other Limitations and Known Problems” on page 12 for a work-around for failures to import Numeric when importing Pyfort-generated modules under certain circumstances.

### 2.3 Configuration

Before installing Pyfort you may wish to edit the file `configuration.py`. Most people will *not* need to do this. Here are the things you can do:

1. Set the default Fortran compiler  
Pyfort generates code that must fit the requirements of the Fortran compiler used to generate the Fortran libraries, but Pyfort itself does not use the Fortran compiler. Pyfort uses a name, called a “compiler id”, that identifies a set of code generation behaviors (See “About compiler ids” on page 3 for more details). An attempt has been made to choose the standard platform-specific

compiler. See “Compilers supported” on page 13 for a list of eligible compiler ids.

Users of Alpha-based computers will need to use `g77alpha`, not `g77`, to match the GNU `g77` Fortran compiler.

Users may override the default on the command line. This would only be necessary if the Fortran library was compiled using a different Fortran compiler than corresponds to the default compiler id.

## 2. Install Pyfort outside of Python.

Set a directory name for project installations by setting the variable `prefix="/somewhere"`, where *somewhere* is the desired location.. If blank, the default, Pyfort installs its files into the Python that was used to install Pyfort, in the `site-packages` subdirectory. (E.g. `$PYTHON/lib/python2.2/site-packages`.) You must also add `--prefix=somewhere` to the invocation of `setup.py` when you install.

## 3. Change the suffix used to generate the project container directory by setting `project_suffix`.

The default is `"_dir"`. You would probably only change this if you have a conflict with some existing package.

## 2.4 Installing Pyfort

Using the python into which you wish to install Pyfort, execute:

```
python setup.py install [--prefix=/somewhere]
```

All the files are python source files; no compilation is required. If you are going to use the `--prefix` option to install Pyfort outside of Python you must also set this in the `prefix` variable in `configuration.py` as explained above.

One script file “`pyfort`” for Unix is installed in the `bin` subdirectory. Pyfort install the extensions you create in the `site-packages` area (e.g. `PYTHON/lib/python2.2/site-packages`).

### Using the GUI Editor requires Tkinter

If you wish to use the GUI Editor for Pyfort project files as described later in this manual, your Python must have been built with Tkinter support. This is usually true for most Python installations.

## 2.5 Testing

### Fortran test

To make and execute the test routine, in subdirectory `test`:

#### 1. `pyfort -i testpyf`

This will build and install the extension. Write permission in the `site-packages` directory is required for the installation. If you do not have this, omit the `-i` and then locate the module `testpyf.so` in the “`build`” directory subtree and move it to the `test` directory, or set `PYTHONPATH` to point to it.

#### 2. `python test.py`

will exercise the routines in `testme.f` and print out the results.

#### 3. You may uninstall the test with:

```
pyfort -u testpyf
```

### Fortran-like C test

Similarly, in subdirectory `testc`, execute:

```
Install: pyfort -i testc
```

```
Test: python test.py
```

```
Uninstall: pyfort -u testc
```

## 3.0 Overview

### 3.1 Module and Project Files

Pyfort takes a “module file” describing the Fortran routines to be called from Python, and creates two output files:

- the C source for a Python extension that is the “glue” between Python and the Fortran library; and
- a text file documenting the way in which the routines will be called from Python.

Depending on the way in which you describe the Fortran routines, the calling sequence from Python can either be the same as from Fortran or substantially simplified to eliminate redundant array-size information, work arrays, and the need to create output arrays yourself.

One or more Pyfort modules can be grouped into a “project” so that they can be built together and uninstalled together. Projects are described in Pyfort Project Files (`.pfp`) which can be edited with a text

editor or with a Tkinter-based GUI editor included with Pyfort. The first time you build a given project the editor will open to help you write it.

### 3.2 About compiler ids

This documentation refers to a “compiler id”. A compiler id is the name of an attribute, such as *g77*, of a compiler object defined in Pyfort’s *fortran\_compiler* module. A compiler id is not usually the same as the name you use to actually run the compiler, although it can be. Instead, a compiler id names a certain compiler object in Pyfort that is used to generate the code of the extension module in the way required by the corresponding Fortran compiler.

The problem is that the correct code to be generated depends on the manufacturer of the compiler *and* the architecture of the target machine. For example, *f77* is the name both of a Fortran compiler for a Solaris machine from Sun, and of a Fortran compiler for an HP workstation. Unfortunately, the code we must generate differs in these cases. Therefore, we must have separate identifying names such as “solaris” or “hp” for these compilers. Similarly, *g77* on a Linux machine and *g77* on a DEC Alpha require different code generation, so we have separate compiler ids *g77* and *g77alpha*.

The compiler id Pyfort uses should correspond with the Fortran compiler we used to make the Fortran libraries. We discuss the available compiler ids in “Fortran compilers” on page 13. In most cases a *-c* option is unnecessary; the person installing Pyfort makes the correct compiler id the default by editing *configuration.py*.

If you are on Windows, see “Visual Fortran on Windows” on page 15.

## 4.0 Command line

Once the module file is prepared, *pyfort* can be executed.

```
pyfort [options] project_name
```

Executing *pyfort* with no arguments will print usage information, including information on the name of the current default compiler id. If the file *project\_name.pfp* does not exist, the editor will open to create it. You must exit the editor with the “Finish”

button, and answer the queries affirmatively, to continue the build.

### 4.1 Options

- b (default) build the project but do not install; use debug options where possible.
- c *compiler\_id*: Choose a Fortran compiler id (e.g., *g77*). See “Fortran compilers” on page 13 for a list of acceptable compiler ids. The default id is chosen appropriately for the platform in the file *configuration.py*; *configuration.py* must be modified before installing Pyfort, or the installation must be re-run after modifying it. This default is shared by all users of that Pyfort installation.
- e Invoke the GUI editor for the project file. If the editor edits through the “Finish” button, you will be asked if you wish to proceed with the build.
- g Build with the debug option.
- i Build and install.
- o *output\_directory*: Place the generated C module file in this directory. Default is “*build/temp.platform*”, where *platform* is the value of Python’s *sys.platform* variable. This option does not affect the placement of the documentation files generated for each Pyfort module file in the project, which are always placed in the current directory and have the name of the Pyfort module file followed by “.txt”.
- u Uninstall the project.
- V print the Pyfort version number and exit.
- X print the executable name of the Fortran compiler and exit. This can be useful in order to find out the default Fortran compiler or to use in creating Makefiles to compile the Fortran. The executable name is not a full path name; you must ensure that the Fortran compiler you want is the first file in your path with that name.

### 4.2 Obsolete Options

The following options are provided for backward compatibility to versions prior to 8.0. A project file will be written for you which you may wish to use in the future. To use these options, use the name of a single Pyfort input module file *something.pyf* in place of the project name.

- f The pyf file named on the command line is in free format (that is, there is no column 1 comment convention).
- L *directory*: Add *directory* to the list of link directories.
- l *library\_name*: Add *library\_name* to the list of link libraries.
- m *module\_name*: Name the extension module *module\_name*.
- p *package\_name*: The setup routine’s “packages” argument will be the list [*package\_name*] if this option is used. Usually *package\_name* is the name of a subdirectory that contains supporting Python code including an `__init__.py` file.
- d *directory*: The setup routine’s “library\_dirs” argument will be {*package\_name*: *directory*}. Usually *directory* is the name of a subdirectory with Python code for a package if this name is different from the package name. If the -p option is not specified, *package\_name* is ‘’.

Packages are explained in the Python documentation. The setup routine and its arguments “packages” and “library\_dirs” are explained in the Distutils documentation.

## 5.0 Creating a Fortran Extension

### 5.1 A Simple Fortran Extension

In this example, we have a simple Fortran function that we wish to make available in Python. First we need to describe that routine to Pyfort, and then we need to compile it, generate the Python / Fortran “glue” required, and compile that linked against the Fortran routines and the Fortran run-time libraries. Pyfort will manage this process for us after we prepare the input file.

If you want to build the Fortran sources yourself, rather than letting Pyfort do it, compile them into a library and use the advanced options explained in “Compilation and Linking Options” on page 7.

## 5.2 Preparing the input file

### Pyfort module files

A module file consists of one or more Fortran routine descriptions. The result of the process will be to make these routines callable from Python. Certain Fortran-like routines written in C can also be used instead of Fortran.

The module file is so-named because the result is to generate a Python module from it. Combined Python / Fortran packages can be built too. See “Packaging with Python Code” on page 8.

Pyfort module files should be named with a “.pyf” extension. The name of the module file, less the extension, is used as the name of the module to be created, unless a different name is specified in the project file.

In the following description, *italics* denote names which the user will supply as desired, while square brackets [...] indicate optional input. The lexical conventions are fully described in “Lexical conventions” on page 12.

A Fortran module file is case-insensitive. Routines will be called from Python using lower case.

### 5.3 Describing a routine’s interface

To describe a function or subroutine, you simply enter the part of the function or subroutine that describes the input and return values, using a syntax that is similar to modern Fortran.

You can use the “intent” attribute to declare each argument as an input, output, or both input and output. The default is that an argument is an input argument. Pyfort produces a Python module containing a method whose name is the same as your routine’s name, and which takes the input arguments as its argument list and produces the output arguments, including a function value if any, as its result.

A special intent *temporary*, not corresponding to any Fortran intent, can be declared for an array which is used only by the called routine as a workspace but whose contents are not wanted as output.

Array arguments, both input and output, should have their lengths described in terms of expressions involv-



ing other (integer) arguments, integer constants, and the usual arithmetic operators.

We will describe the formal specifications later, but the following examples should suffice for most purposes.

Suppose we have a Fortran function `sum` that adds up the elements of an argument `x` an array of type `real` and length `n`

**FIGURE 1.** Fortran in file `arrayut.f`

```
function sum (n, x)
  integer n
  real x (n)
  real sum
  integer i
  sum = 0.0
  do 100 i = 1, n
    sum = sum + x(i)
100  continue
  return
end
```

We are going to build a module “arrayut” that will contain this function as “arrayut.sum”. We name the Fortran file `arrayut.f`. We create a module file named `arrayut.pyf` (the name `arrayut` will become the name of our extension module). The file `arrayut.pyf` contains:

**FIGURE 2.** Module file `arrayut.pyf`

```
function sum (n, x)
  ! sum (n, x) = sum of the real array x (n)
  integer n
  real x (n)
  real sum
end function sum
```

The line right after the function head containing the exclamation point is a comment. Any comment lines right after the routine declaration but before the first argument declaration will be used as a documentation string for the Python function, so including it is a good idea. Comments can also be given as a line beginning with a `c` or a `C` in column 1.

Every argument to the routine (and the procedure name itself, if it is a function), must be typed in the declarations. Spaces must be used to separate keywords and identifiers, but unlike in Fortran may not occur within identifiers, numbers, etc.

Now are going to build a module “arrayut” that will contain this function as “arrayut.sum”. While Pyfort is going to handle the process for us, it is useful to understand the steps involved.

To create the Python extension we must compile the Fortran routines, run Pyfort, compile the resulting C modules, and link the Fortran and C into a shared module that Python can load. Here are the steps:

1. Pyfort creates the `arrayutmodule.c` C extension file from the Pyfort module file `arrayut.pyf`. A text file `arrayut.txt` documenting the Python calling sequences of the Fortran routines is also created.
2. If requested, Pyfort compiles the Fortran routine(s) into a library.
3. Pyfort compiles and links the C extension into a shared object file, linking it against `libarrayut.a` and the Fortran runtime libraries.
4. Pyfort installs the module into Python.

Execute:

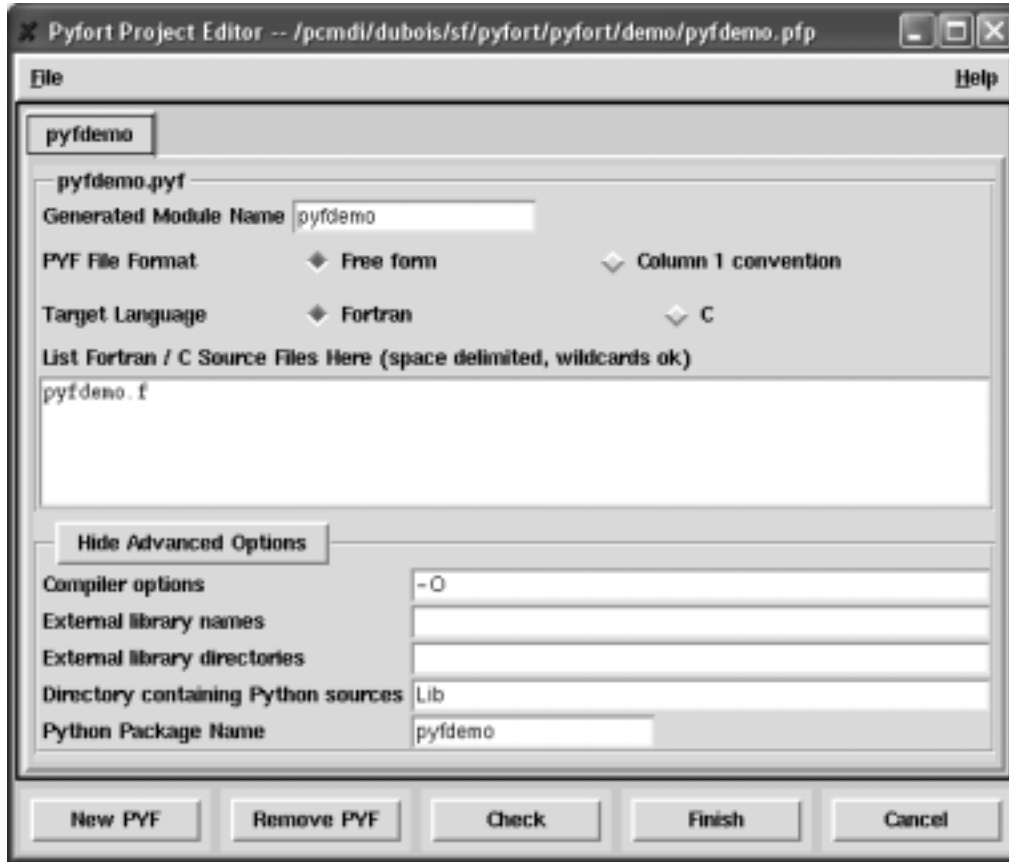
```
pyfort arrayut
```

## 5.4 The Pyfort Project Editor

Since this is the first time we are building the project, the project editor will appear. The appearance of the

project editor when invoked in the Pyfort demo directory is shown in Figure 3 on page 6.

**FIGURE 3.** Pyfort Project Editor



Click “New PYF” and select `mystuff.pyf`. Click the “Free form” radio button if you are not using column 1 comments. In the large box we need to give a space-delimited list of our Fortran sources. Making sure the “Target Language” selection is Fortran, type “arrayut.f” into the source box.

You can use as many lines as you wish, and you can use wildcards as used by the standard Python library module, `glob`. Each white-space delimited name you enter will be passed to the `glob.fnmatch` routine to match filenames. The wildcards are similar to most Unix shell conventions. File names should be relative to the directory that holds the project file.

When it builds your project, Pyfort creates the desired Python module `arrayutmodule.c` and the documentation file `arrayut.txt`. The names of these files and of the Python module is determined by the name of the module file. Only one Python module can be described in a single module file. You can choose a

different generated module name by changing the name in the “Generated Module Name” box..

You can click “Check” to perform some basic sanity checks on the project.

Now click “Finish”. Your package will be compiled and, if you used the `-i` option, installed.

A project file `arrayut.pfp` will have been created, and the next time you execute `pyfort arrayut` the editor will not appear unless you use the `-e` option. You’ll also see `arrayut.txt`, the documentation for your routines. Except for this file, all the products of your build are in the subdirectory “build”, and you can clean up simply by removing it.

You can uninstall the module with “`pyfort -u arrayut`”.

## 5.5 Compilation and Linking Options

If you want to do the Fortran compilation more carefully, such as specifying optimization options, or additional external libraries for linking against, click the button to turn on the Advanced Options.

You would use the library options in two cases:

1. Your Fortran sources depend on functions in one or more other libraries.
2. You want to compile the sources yourself. In that case, you compile them into a library, leave the “Sources” box blank, and treat them as an external library.

Libraries are specified in a special way to increase portability. Library names are given without the “lib” prefix and the suffix such as “.a” that they might have on Unix. Thus, a library `libs.a` would be specified with the name “rs”.

You fill in any library names, space delimited, in the Library Names box, and in the Library Directories box give a space-delimited list of directories to search for the external libraries. Other compile options for the Fortran routine go in the remaining box.

The remaining advanced options have to do with packaging Python code along with your compiled code, and are discussed below in Section 5.10 on page 8.

## 5.6 Calling the routine from Python

From Python, you will access your sum routine by doing:

```
import arrayut
```

and your routine `sum` will be known as `arrayut.sum`

Here is “sum” in action:

```
import Numeric, arrayut
x = Numeric.arange(10) / 2.0
print arrayut.sum(len(x), x)
```

And, as with any Python object, the user can see the documentation for the function `sum` by printing its so-called “doc string”:

```
print arrayut.sum.__doc__
```

This would print the comment line(s) from our input, in this case:

```
sum(n, x) = sum of the real array x(n)
```

It is possible to instruct Pyfort to calculate the argument `n` from the length of `x`, instead of requiring it as an input. See “Using valued scalars” on page 8.

Note that Pyfort will handle any needed conversion of input values:

```
y = x.astype(Float32)
print arrayut.sum(len(y), y) # works also.
```

On a computer where a Fortran variable declared “real” is a 32-bit quantity but a Python “float” is 64-bit, the array `x` was copied into a 32-bit real array that was then passed to the Fortran routine `sum`. On the same computer, the call with `y` as an argument simply passed `y` as-is, since it was already the right size.

## 5.7 Array output

Assume `normer` (command, `x`, `y`, `n`) is a Fortran routine whose first argument is a string that should either be “norm” or “none”, a real input array `x` (`n`), and a real output array `y` (`n`), where `n` is the integer length of these two arrays. The specification for this function can be added to `arrayut.pyf` by adding the following just below the end of the `sum` function:

```
subroutine normer (command, x, y, n)
  character*(*) command
  integer n
  real x(n)
  real, intent(out):: y(n)
end
```

The resulting Python method might be called like this:

```
y = arrayut.normer(“norm”, x, len(x))
```

## 5.8 The Python signature is determined from the Fortran signature.

As the preceding example illustrates, any Fortran argument declared as having intent “out” is not present in the Python calling signature; rather, it becomes an output. If the routine is a function, the function value is also an output. If there is only one output, it becomes the return value of the python function. Otherwise, if there is more than one output, a tuple of the values is returned, with the Fortran function value if any first, followed in order by the other outputs.

### Intent “inout”

An argument of intent “inout” is present in the Python function’s calling signature and is *not* one of the out-

puts. Note that the concept of “inout” does not map well to Python and generally should be avoided. Pyfort will reject such an argument unless: the actual argument is an array with the exact typecode expected by the Fortran compiler. If you have a Fortran routine with a scalar inout argument, you should declare it as an array of length 1 in the Pyfort input and pass an array of length one as an actual argument.

If the Fortran routine modifies this array the results will be present in the Python variable passed to the routine, which therefore should be a variable not an expression. Such an error cannot be detected.

### Intent “temporary”

Because Fortran originally had no dynamic memory management, many older routines require you to pass in space for temporary arrays. While a Fortran user must declare and pass such arrays, a Python user does not have to do so. We simply tell Pyfort that an argument is a temporary by giving it an intent of “temporary”.

An argument of intent “temporary” must be an array. It is present in neither the input or output of the Python function. Instead, it is created on the fly before the call to the Fortran routine and disposed of immediately afterwards. This saves the Python program from needing to create a temporary to pass in.

## 5.9 Using valued scalars

Considering the function *sum* again (Figure 1 on page 5), it is inconvenient to have to call it from Python using both the array and its length as arguments. Instead, we can declare the argument *n* to be calculated using the length of the array *x*. To do this, we change the declarations to:

```
function sum (n, x)
  ! sum (n, x) = sum of the real array x (n)
  integer n = size (x)
  real x (n)
  real sum
end function sum
```

The function *size (x)* is the modern Fortran intrinsic function for the length of an array. It takes an optional second argument giving the number of the dimension in the case that *x* is multi-dimensional. Alternate forms of the above declaration for *n* are:

```
integer n = size (x, 1)
integer:: n = size (x, 1)
integer:: n = size (x)
```

Any of these has the same effect. The argument *n* is no longer part of the calling sequence from Python. So now we call *sum* as:

```
sum (x)
```

The initial value specified for *n* can be any expression involving integer expressions and the size operator. The arrays given as first arguments to size operators must be input arguments of intent ‘in’ or ‘inout’. The checking of array sizes is done after all such “valued” scalars have been calculated.

Currently the expression used to calculate a valued scalar can involve another valued scalar only if that scalar occurred earlier in the argument list.

Valued scalars can be used as the sizes of output or temporary arrays. Note that only input arrays can be used as targets of the size operator.

Suppose a Fortran routine copies a two dimensional array *a* to an output array *acopy*: We declare this in Pyfort as:

```
subroutine copy2 (a, n, m, acopy)
  real a(n, m)
  real, intent (out) acopy (n,m)
  integer n=size (a, 1), m = size (a, 2)
end
```

Then this routine is called from Python as *acopy = copy2 (a)*, and never fails due to incorrect sizes.

Sometimes mathematical routines need work arrays and their length depends on the size of other inputs. Here for example is a declaration for a routine that takes two arrays and needs a work array *t* whose length is equal to the product of the lengths of the other two arrays plus five:

```
function alpha (n, x, m, y, ldt, t)
  real alpha, x (n), y (m)
  real, intent (temporary):: t (ldt)
  integer n = size (x), m = size (y), ldt = n * m + 5
end
```

This function is called from Python as *alpha (x, y)*, returning a real scalar value.

## 5.10 Packaging with Python Code

To access the Advanced Options in the GUI, click the button to make them visible. There are two options that assist you in making a fancier extension, either by including a simple set of additional Python modules

or by making a complete Python package. There are two motivations for doing so.

1. You can customize the interface so that the call from Python to Fortran is safer, more attractive, has default and keyword arguments, and so on.
2. By making an official “package” of the combined Python and Fortran modules, you can prevent name pollution and control which names are visible to the end user.

Python packages are more fully explained in the Python documentation or any good book on Python. The Python Directory box is the place to enter the name of a directory containing additional Python (\*.py) modules to be installed along with your package.

If you specify a Package Name, your Python Directory must be given and that directory must contain a file named “\_\_init\_\_.py”. This name has two underscores on each side. This is a special file name that Python uses to consider a directory to represent a “package”. If you use a package, all the names of the modules in the package will be relative to the package name. For example, if we created an \_\_init\_\_.py file containing:

```
from arrayut import sum
and specified a package name “au”, then we would access sum as au.sum rather than arrayut.sum. However, we could also completely hide the Fortran routine sum and make a Python routine that called it by writing instead:
```

```
import arrayut
def sum (x):
    if len(x) == 0: return 0
    return arrayut.sum(len(x), x)
```

The Project Editor makes it easy to experiment. However, be sure to uninstall the project before trying a new variant to be sure of your results.

## 6.0 Using the wrapper options

Each wrapper function created by Pyfort can take an extra, optional argument which controls the behavior of the wrapper. The default value of this extra argument can be set with the module method `set_pyfort_option(OPTION_NAME)`. Each module created by Pyfort has its own, individual control. The current value can be queried with `get_pyfort_option()`.

The available options are integer attributes of the module. The available values for this option are:

- NONE: no effect. Output arrays are transposed on return to Python (but see note below).
- TRANSPOSE (default): any multidimensional arrays will have their data areas transposed upon input. This corrects for the fact that Python stores arrays in row-major order while Fortran uses column-major storage. Output arrays are transposed on return to Python (but see note below).
- MIRROR: the Fortran array declares the array arguments in the reverse of the order in which the Python input has them. Thus the data area of the Python array is as expected by the Fortran routine already, but the shape is backwards. Output arrays are created with their shape reversed, but not transposed upon return.

Note: It should be noted that for options NONE or TRANSPOSE, a multidimensional output array created by Pyfort is a “non-contiguous” array. This is a form of “lazy” evaluation of a transpose operator. If such an output array is then passed to another Pyfort-wrapped routine using the TRANSPOSE option, then in fact no data movement takes place.

## 7.0 Extending Numerical Python with C

### 7.1 Using the C compiler option

Pyfort can make it easy to create extensions to Numeric with C by writing C routines that look a lot like Fortran. Use the Project Editor to choose “Target Language” as C and list your C source files in the Source box. The environment variable “CC” can be used to control the C compiler to be used, which should match the one used for compiling Python itself.

In the Pyfort module file you describe the routines as if they were Fortran. For each argument that is an array, either input or output, the C routine will expect an appropriately-typed pointer. For example, an input array that is described as:

```
doubleprecision x(n)
```

will correspond to an argument to the C routine of type pointer-to-double (double\*).

C modules are built with the transpose option set to zero.

Subdirectory testc contains a complete example showing the Makefile, Pyfort module file, project file, C routines, and Python test routine.

## 7.2 C Example

The C routine:

```
double ss (int n, double* x) {
    int i;
    double d;
    d = 0.0
    for (i=0; i < n; ++i) { d += x[i]*x[i];}
    return d;
}
```

can be described with this Pyfort module file ssmodule.pyf:

```
function ss (n, x)
integer:: n = size(x)
doubleprecision ss, x(n)
end
```

and then called from Python:

```
import Numeric
x = Numeric.array([1., 2., 3.])
print ssmodule.ss(x)
```

In the project file, just check the “C” option for ssmodule.pyf. A single module may contain C routines or Fortran routines, but not both. If this is a hindrance, use a single Python package to hold the names from the different compiled modules.

## 8.0 Inter-language communication issues

### 8.1 Representation issues

When interfacing any two languages there is a problem caused by differing representations. Standards committees being what they are, the representation for a particular type is usually compiler-dependent. Fortran has a notorious problem with type “logical”, for example; attempts to mix routines compiled with different compilers may encounter a problem because

the two compilers do not agree on what constitutes “.true.”.

Python’s native floating-point type is promised to be the same as a C “double”. On many machines this corresponds to a Fortran “doubleprecision”. The Numeric package includes the ability to explicitly create arrays of 32-bit or 64-bit floating-point numbers. This is done by adding a second argument to the array constructor, e.g.,

```
x = array(arange(1000), Float32)
```

This is in general not necessary with Pyfort, as Pyfort will do the conversions required. However, if you wish to minimize the space requirements of a particular algorithm you may wish to specify the correct type at the Python level.

### 8.2 One-dimensional array issues

Given that we are going to pass a Python array or list or tuple to a Fortran routine, we have different semantics for Fortran and Python: Python arrays have a length; Fortran array arguments (except for explicit-interface arrays) are passed only by address, and it is assumed that one of the arguments or a common block variable is available to provide the length. Currently, Pyfort requires that the length of the array be derivable from one of the integer input arguments using ordinary arithmetic operations. So, for example, we might have:

```
function h (n, m, x, y, z)
integer n, m
real x (n), y (n + m - 1), z ((n+1) * m), h
end
```

On input, the actual sizes of the arrays x, y, and z will be compared to the values of n and m and an exception will be thrown if all is not as expected. See “Checking rank and extent” on page 11.

Fortran arrays are usually indexed from 1 but this is changeable on a per-array basis. Python arrays are always indexed from zero. It is possible to design a new “Fortran array” object for Python but the indexing of such arrays would have a unique interpretation and that would mean users would have to be very conscious of which kind of array they were dealing with. For example, in Python, x[-1] denotes the last element of the array; if we had a Python object with arbitrary lower index then we would have to change that interpretation. We have instead simply chosen to consider the Python indices as zero-based counters into the array extent.

Checking of array sizes is discussed below (“Checking rank and extent” on page 11).

### 8.3 Multiple-dimension array storage

C and Python use row-major order but Fortran uses column-major order. That is, in Python and C, assuming  $x$  is two-dimensional,  $x[0, 0]$  is the first element in memory and  $x[0, 1]$  is the second element in memory. In Fortran,  $x(1,1)$  is the first element and  $x(2,1)$  is the second.

The storage-order problem presents an insoluble dilemma for a tool such as Pyfort. An argument passed to Fortran by Pyfort may have come from a native C or Python source, in which case the data, but not the shape, needs to be transposed in order to have Fortran perceive it correctly. Or, it may have come from an extension object and be in the right order already.

For multiple dimensional arrays, Pyfort leaves it to you to put the array in the correct storage order for Fortran. To do otherwise might be a convenience for some cases but reduce the range of applicability of Pyfort. The following Python routine can be used for passing a Numeric array to Fortran:

```
import Numeric
def row_major(x):
    "Same shape but row-major order for data."
    return Numeric.reshape(Numeric.transpose(x),
                           x.shape)
```

When Fortran returns an array, Pyfort creates the returned object as an array object whose data area is still in column-major order but which Python considers non-contiguous. This “lazy transpose” means the returned array will behave correctly in Python and leaves open the possibility that the actual movement of the data into row-major order may never have to be carried out.

Another aspect of this question is whether we should design for minimum data movement or for maximum safety and convenience. The current design takes the latter approach, but we continue to investigate how both goals could be achieved.

### 8.4 Checking rank and extent

What kind of checking should of array sizes should be done? Here again there is no obvious answer. We could check total size, total size and rank, or rank plus

specific dimension extents. Pyfort checks rank plus specific dimension extents to be consistent with the spirit of modern Fortran.

However, to increase flexibility, if the final dimension of an input array is declared to be 1 in the Pyfort input, then no checking of the input’s size in that dimension is done. The user may use the Fortran convention of declaring this dimension with an asterisk.

Note that this would not make sense for output or temporary arrays.

## 9.0 Limitations

### 9.1 Explicitly-interfaced routines are not yet supported

This release of PyFort is only capable of calling routines with “implicit” interfaces. “Explicit interface” is a concept introduced in Fortran 90.

A subprogram has an explicit interface if it is contained in a module, another subprogram, or is declared in an interface block in some module.

Only routines with explicit interfaces can use some of modern Fortran’s powerful features. Fortran 77-style programs do not have routines with explicit interfaces, even if compiled with a modern Fortran compiler, and therefore do not face any limitation with respect to Pyfort.

For the moment, it is possible to interface to an explicitly-interfaced routine by writing a simple Fortran wrapper that itself does not have an explicit interface, and then connect Python to that wrapper.

Here is the reason for this limitation. The Fortran standard does not prescribe the descriptors that will be used to pass an array to an explicitly-interfaced routine. (Since the argument must include more information than just the address, some sort of structure is necessary, but the standard leaves exactly *what* structure as an “implementation detail”). Thus, a tool like Pyfort will have to deal with this on a compiler-specific basis.

The second feature of modern Fortran that it would be nice to support is the ability to define structured types. There is again, unfortunately, no standard for how the pieces of such a type will be arranged in memory

unless a SEQUENCE specifier is given, but such a specifier may not be desirable for every user.

## 9.2 Other Limitations and Known Problems

The current release has the following limitations.

1. The following Fortran 77 types are handled correctly: integer, real, real\*8, real\*16, complex, complex\*8, complex\*16 may work but we haven't tested it yet.
2. Character arguments should be declared character\*(\*). In fact, a character argument in Fortran should always be typed this way anyway as giving it a specific size doesn't actually do anything.
3. Character return values are not yet supported on all platforms. The test routine has a test for this that will tell you if the character test fails.
4. Complex scalar return values are not supported on all platforms, particularly the Solaris compiler. The test routine will detect this.
5. Array dimensions should be given as a total size since the form (lower: upper) is not yet supported. Python doesn't understand non-zero based arrays anyway.
6. An expression used to calculate a valued scalar can involve another valued scalar only if that scalar occurred earlier in the argument list.
7. Numeric Python arrays are used in Pyfort-generated extensions (although in calling Fortran routines you can in fact pass a variety of Python objects such as lists, in addition to Numeric arrays). We have run into one odd error that we do not particularly understand, but we know the solution. Ensure that any module that imports a Pyfort-generated module first imports Numeric. This should not be necessary since the Pyfort-generated module imports Numeric itself, but under certain circumstances it seems to be needed to prevent failure of the importation of the Pyfort-generated module.
8. The expression used to calculate a valued scalar can involve another valued scalar only if that scalar occurred earlier in the argument list.
9. Not all Fortran compilers are yet known to Pyfort. Please add the necessary information to fortran\_compiler.py and submit a patch on our SourceForge site.

## 10.0 Input grammar

The full input grammar is close to that for specifying interface blocks and modules in modern Fortran. In the following description, *italics* denote names which the user will supply as desired, while square brackets [...] indicate optional input. Alternative choices are listed so: {this|that}.

### Lexical conventions

Comments begin with an exclamation point and including everything to the end of that line. A "c" or "C" in column 1 is also recognized as beginning a comment, unless the -f option is used.

The present implementation uses only those comment lines immediately following the start of each routine.

Input is free format but for compatibility with modern Fortran, an ampersand (&) at the end of a line can be used to indicate a continuation line. Column 6 conventions are *not* recognized. No identifier should contain internal whitespace or be the same as a Fortran keyword, such as module, subroutine, function, end, contains, interface, integer, real, doubleprecision, complex, logical, dimension, intent. A non-alphanumeric character or line break must separate an identifier.

The types doubleprecision and doublecomplex may include spaces between the "double" and the part that follows.

Input is case-insensitive. Functions in the generated module will have lower-case names when called from Python.

### Grammar

A Pyfort module file consists of zero or more <routine> specifications, each of the form:

```
{function|subroutine} name ( [arg1, arg2,...]
  <declarations>
end [{function|subroutine}] [name]
```

The Fortran option to use the return type of a function as a prefix to the function statement is not yet supported; a declaration for the function type will be needed in the declarations.

Compound constructs, such as <routine>, that use an "end" statement to mark their closing also accept an optional "end-tag" consisting of a repeat of the con-



struct type and name. If given, the construct type and name must match correctly.

In the <routine> specification, *arg1*, *arg2*,... are simple names for each argument to the Fortran routine *name*.

The <declarations> consist of zero or more individual <declaration> statements. Each <declaration> can be in one of two forms, the traditional or attribute form. The traditional form is:

```
<typespec> <itemlist>
```

while the attribute form (which must be used for output arguments) is:

```
<typespec> [, <attributelist>]:: <itemlist>
```

A <typespec> is the name of a type followed by an optional kind parameter in one of the forms *\*kind*, (*kind*), or (*kind = kind*).

Each item in the comma-delimited <itemlist> consists of a name and optional dimension information (*size-expression* [, *size-expression*,...]), each *size-expression* being an integer expression involving integer constants, the names of integer arguments to this routine, and the standard arithmetic operators. The form of each *size-expression* can also be in the form lower-bound: upperbound, where each part is such an expression.

An <attributelist> is a comma-delimited list of attributes, each of which is one of:

intent (in) -- denotes an input argument (the default)

intent (out) -- denotes an output argument

intent (inout) -- denotes an input argument that is modified

intent (temporary) -- denotes an output argument that is not wanted, e.g., the Fortran argument is a user-supplied workspace.

Thus the declaration for an output array *x* of integer type and length *n* is:

```
integer, intent (out):: x (n)
```

Each identifier in one of the declarations should be:

- the name of either the routine itself, if and only if the routine is a function; or
- the name of one of the arguments.

Each argument must be declared exactly once, and if the routine is a function, the function name must be declared without an intent.

## 10.1 Declaring array sizes

Each array argument must be declared using a size that is an expression that includes only:

- Literal integers
- Arithmetic operators and parentheses
- The names of other input arguments of type integer

A final dimension may be specified as 1 or an asterisk if desired; this means the length of the input in that dimension will not be checked (“Checking rank and extent” on page 11).

## 11.0 Fortran compilers

### 11.1 Pyfort compiler ids

Pyfort must write the C extension file taking into account these factors:

- The correspondence between C types and Fortran types;
- The name the Fortran compiler uses for external; names; and,
- The convention used by the Fortran compiler for passing arguments.

In order for Pyfort to work with your Fortran compiler you need only choose the compiler id of a supported compiler that treats these items the same way as yours does.

Pyfort has a number of compiler objects with pre-defined names. You must choose one of these compiler ids. If there is no suitable compiler, you can define a new one as described in “Adding a new compiler” on page 14.

### 11.2 Compilers supported

The ids listed below can be used as an argument to Pyfort using the *-c* option. The default value is set in the file *configuration.py* before installing Pyfort.

- *g77* is the GNU Fortran compiler. It adds an underscore to the Fortran name to get the

link name, or two underscores if the Fortran name contained an underscore.

- `g77alpha` is the GNU Fortran compiler using a type mapping suitable for the DEC Alpha chip.
- `solaris` is the Sun Solaris compiler. It adds an underscore to the Fortran name to get the link name.
- `pgf77` is the Portland Group Fortran 77 compiler on Linux. It adds an underscore to the Fortran name.
- `pgf90` is the Portland Group Fortran 90 compiler on Linux. Use this choice if your Fortran was compiled with `pgf90` in order to link in the correct libraries.
- `sgi` is for a Silicon Graphics workstation.
- `ffc` is for the Fujitsu Fortran Compiler. These settings have been tested on Linux. The contributor believes this may be the same as the Lahey compiler.
- `absoft77` and `absoft90` are for the Absoft compilers.
- `f77_OSF1` is for the Dec Alpha.
- `vf` is Dec Visual Fortran on Windows. See “Visual Fortran on Windows” on page 15.
- `cc` or `gcc` is the C compiler (!). The `testc` sub-directory shows an example of using this option. This option allows you to extend Numerical Python in C in a natural way.

### 11.3 Adding a new compiler

To add a new compiler id you edit the file `fortran_compiler.py`. This file contains definitions for the existing compiler ids listed above.

Attributes of the compiler object need to be set that define the directories and libraries to link with to obtain the Fortran compiler’s runtime support. Usually these libraries are defined in the Fortran compiler’s documentation.

For many compiler and platform combinations, the only thing that needs to be done is to create a new compiler object using class `F77Compiler`, set these fields, and add a case to `get_fortran_compiler()` to test for the new id.

If one of the standard compilers is adequate except that the mapping between Fortran and Python types must be changed, you can instantiate the class with a

different typemap. An example of this is the definition of `g77alpha` in file `fortran_compiler.py`.

You can also add a new class that inherits from `FortranCompiler` and redefine its methods as necessary. The class `G77Compiler` is an example.

There may be one or several runtime libraries that need to be loaded, and some that may be only needed if the Fortran contains certain constructs such as I/O. Generally, deciding this list in the absence of good documentation is a trial and error process.

Link failures may not become evident until the module is imported and the dynamic load fails with a message about a missing external. If the name seems closely related to the name of one of your own procedures defined in your Pyfort module file, the problem is very likely the mapping of external names. Use the utility `nm` to determine how the externals have been named in your Fortran library and adjust the mapping of names accordingly. `G77Compiler` is an example of a class that changes the name mapping.

If a missing external seems like some obscure name you never heard of, it is almost always a missing runtime library.

A non-standard convention for argument passing will likely involve defining a new child of `FortranCompiler`. Each of the methods in the compiler class does the generation of a distinct portion of the generated code for a module, and we have tried to add a comment on each method about just what it is doing. Adjust the compiler by overriding methods with new versions to change the code generated. We apologize for the lack of documentation in this area.

## 12.0 Pyfort Project Files

Starting with Version 8, Pyfort uses files called `PyFort Project` files, or “.pfp” files. If the `project_name` you give does not correspond to an existing project file `project_name.pfp`, or if you use the `-e` option, a GUI editor will appear that will help you create or edit the project file. Once you have created the file, subsequent invocations will build and install your Pyfort extension without invoking the editor again, unless the `-e` option is used. As the editor exits, you will be asked to decide whether or not to proceed with the installation of your extension.

The basename of the project file is used, with “\_project” added, as the name of a subdirectory which will hold all the modules in the project. This directory is added to the Python path using a project\_name.pth file in Python’s site-packages directory.

### Format of project files

The pfp files have a very simple textual format, and you might find it convenient to view or edit the file with a text editor.

A project file has two basic parts: descriptions of Pyfort module files (.pyf files) and descriptions of compiled libraries.

Each Pyfort module file is described in the Pyfort project file with a statement of this form:

```
pyf (filename,  
     sources=[],  
     generate_as='',  
     libraries='',  
     library_directories='',  
     compiler_options='',  
     package_name='',  
     python_directory = '',  
     module_name='',  
     freeform = 0,  
     use_c_compiler=0  
     )
```

For example, if the Pyfort module file is example.pyf, and the corresponding Fortran is in a file “rs.f”, and there is no accompanying Python code to go with it, the line in the project file might be:

```
pyf ('example.pyf', sources=['rs.f'])
```

If we wanted to generate the module as myexample instead of example, we would have:

```
pyf ('example.pyf', sources=['rs.f'],  
     generate_as = 'myexample')
```

## 13.0 Notes for Specific Platforms

This section contains notes for particular systems.

### 13.1 Visual Fortran on Windows

The following information was contributed by Reinhold Niesner.

Subdirectory ‘windows’ contains a sample Pyfort application on Windows. This is basically the same test routine as in subdirectory ‘test’.

pyfort.bat will have been installed in your Python directory in subdirectory ‘Scripts’. Be sure that is in your path.

#### 1. Start DIGITAL Visual Fortran

Create a new dll project (File -> New -> Fortran Dynamic Link Library); for 'Project Name' type 'testme' and choose as location the directory where 'testme.f' resides; choose 'empty dll project'

#### 2. Add your fortran files (Project -> Add To Project -> Files), here 'testme.f'

#### 3. Make a definition file with all symbols to be exported (name it 'testme.def'); if you don't know how to do this see below.

#### 4. Add the definition file to the project (Project -> Add To Project -> Files -> testme.def)

#### 5. Run Pyfort

```
pyfort -b -ltestme -L<where-testme.lib-is> testpyf.pyf
```

After this you should have a file called 'testpyf.pyd'. Copy this file, 'testme.dll' and 'testit.py' to the directory and you should now be able to run testit.py

### Creating an export symbols file

If you don't know how to export symbols do the following to create one:

#### 1. Compile without a definition file

#### 2. In a DOS window, change to the directory where the created dll resides (usually 'Debug' or 'Release')

#### 3. Copy the script dllsymbols.py to this directory

#### 4. Execute: python dllsymbols.py testme.obj

This uses the visual studio tool dumpbin.exe, make sure it's in your PATH or edit the respective line in the script.



## Index

### A

- Alpha-based computers 2, 3
- array
  - output 7
  - size 8, 10
- arrays
  - size and rank checking 11
- asterisk, for declaring array length 11
- automatic sizing 8
- Aycock, John 1

### C

- C compiler 9
- character arguments 12
- character-valued functions 12
- checking input sizes 11
- clean 6
- code generation 3
- column-major 11
- command line 2, 3
- comments 7
- compiler ids 3
- compiler, adding a new 14
- complex return value bug 12
- configuration 1, 2
- contiguous 9
- conversion 7
  - precision 10
  - type 10

### D

- declaring array sizes 8
- default
  - code generation location 3
- dimension, indeterminate 11
- dimensions, multiple 11
- documentation 1
  - doc string 7
- documentation, inadequate 14

### E

- execution 3
- explicit-interfaces 11
- explicit-interface 1
- extension
  - overview 4
- extent 11
- external names 14

### F

- floating-point 10
- Fortran 90 1
- Fortran 95 1
- Fortran compiler 3
- Fortran compiler, setting default 1
- fortran\_compiler 3
- Fortran-like C 9

### G

- g77 3
- g77alpha 3, 14
- generating code 3
- get\_fortran\_compiler 14
- get\_pyfort\_option 9
- grammar 12
- GUI Editor 5

### I

- implicit vs, explicit interfaces 11
- indeterminate lengths 11
- initializing
  - array sizes 8
- input file 4
- input, preparing 4
- input, preparing Pyfort 4
- Installing 2
- intent 4
  - out 7
  - temporary 4, 8

### L

- length 10
- libraries
  - linking 4
- library
  - Fortran, compiling 4
- library\_dirs
  - setup option 4
- limitations 11
- linking
  - troubleshooting 14
- loading, dynamic 14
- logical type 10

### M

- MIRROR (transpose option) 9
- missing external 14
- modern Fortran
  - definition 1
- module
  - generated C file 3
  - module file 4
  - module statement 4

### N

- non-contiguous 9, 11
- NONE (transpose option) 9
- Numeric 1, 10, 12
- Numerical Python 1

### O

- option
  - c 3
  - d 4
  - l and -L 4
  - o 3
  - p 4
- options 2
  - wrapper 9
- options, command-line 3

## **P**

- packages 4
  - setup option 4
- permissions 2
- precision, floating point 10
- prefix 2
- Project Editor 5
- project\_suffix 2
- Python 1

## **R**

- rank 11
- rank, checking 11
- README 1
- real, Fortran type 7
- releases, code 1
- representations 10
- routines
  - describing 4
- row-major 11
- runtime libraries 14

## **S**

- scalars, valued 8
- set\_pyfort\_option 9
- setup 2
- signatures, determining routine 7
- size 8, 10
- SourceForge 1
- space, saving 10, 11
- Spark 1
- standards committees, irreponsibility of 10
- storage order 11
- strings 12

## **T**

- temporary 4
- testing 2
- transpose 9, 11
- TRANSPOSE (transpose option) 9
- type conversion 7

## **U**

- u 3
- uninstall 3, 6
- Usage 2

## **V**

- V 3
- valued scalar 8, 12
- version 3

## **W**

- wrapper options 9

## **X**

- X 3

## **Y**

- Yang, Brian 1