The Go Language for Science

David J. Raymond
Physics Department and Climate and Water Center
New Mexico Tech
Socorro, NM 87801
USA
david.raymond@nmt.edu

February 12, 2025

Contents

1 Introduction **2**
1.1 Information on Go . . . 3
1.2 Using the *go* command . . . 4
1.3 Using the *gccgo* compiler . . . 5
1.4 *Go* or *gccgo*? . . . 6
1.5 Getting started . . . 7
1.6 Emacs Go mode . . . 9
1.7 Speed of execution . . . 9

2 Language elements **9**
2.1 Elementary data types . . . 10
2.2 Variables, expressions, and assignment statements . . . 10
2.3 Constants . . . 12
2.4 Functions . . . 12
2.5 Branching . . . 13
2.6 Arrays, slices, and pointers . . . 14
2.7 Looping . . . 19
2.8 Structures . . . 20

2.9 Maps . . . 22
2.10 Methods . . . 23
2.11 Scoping rules . . . 24
2.12 Concurrency and parallel processing in Go . . . 25
2.13 Calling C language functions from gccgo . . . 27

3 Common mistakes in writing Go programs 31

4 Standard packages 32
4.1 Errors . . . 32
4.2 Conversions . . . 33
4.2.1 Casts . . . 33
4.2.2 Interpretations . . . 34
4.2.3 External representations . . . 35
4.2.4 Formatted conversion . . . 36
4.3 Operating system services . . . 37
4.3.1 Command line arguments . . . 37
4.3.2 Exiting a program . . . 37
4.3.3 Executing another program . . . 38
4.4 Input and output . . . 38
4.4.1 File handling . . . 38
4.4.2 Formatted I/O . . . 41
4.5 Strings . . . 45
4.6 Math . . . 45
4.6.1 Standard math . . . 45
4.6.2 Complex math . . . 46
4.6.3 Random numbers . . . 46

5 Other packages 46
5.1 Candis . . . 46
5.2 Gomatrix . . . 51
5.3 Fast Fourier transform . . . 61
5.4 Sio, a simple input/output package . . . 65
5.5 Gompi, a gccgo wrapper for C language MPI calls . . . 67

1 Introduction

Go is a relatively new language of considerable interest to scientists. It is a

compiled language in the tradition of C, but designed to improve vastly the language's simplicity, safety, and expressiveness. It is a product of Google, in particular, of Robert Griesemer, Rob Pike, and Ken Thompson. Thompson is a pioneer in computer science from Bell Labs, having invented the original version of Unix among many other computer tools in daily use. Go is an open source project and is already used by a large number of individuals and corporations. Numerous libraries, including those in linear algebra and fast fourier transforms are available for Go, and there is an interface by which the vast collection libraries developed in C can be tapped.

Among the safety features of Go are automatic garbage collection (allocated memory is automatically freed when not needed), safe pointers (you can't trash arbitrary memory), and automatic bounds checking for arrays. Type conversions on variables are explicit (you can't multiply an *int* variable times a *float* variable; you have make an explicit conversion), though literal integers (like 7) are interpreted as floats in an expression with floating point variables. There are no included header files; compatibility checking between routines in different files is done automatically. Though all variables must be declared, the process for doing so is streamlined, with explicit type specification only needed when the compiler can't guess the appropriate type. The type system is simple, unlike many modern languages. Unlike C, there is a *complex* type.

Though Go is very new by programming language standards, two separate compilers already exist for it, *go* and *gccgo*. The first brings with it a novel and pleasant development environment whereas the second is a more traditional implementation based on the Gnu gcc backend. The go compiler is very fast, allowing rapid turnaround for code development. The gcc version fits better into the traditional programming environment and is reputed to produce somewhat faster code. Both versions are available as official packages in Arch Linux, as well as in many other Linux distributions.

A previous knowledge of C and/or Python would be helpful in understanding these notes, but is not necessary.

1.1 Information on Go

- The website for Go is here.
- This web page has an excellent tutorial.
- The documentation for the *go* command is here.

- Advice on the effective use of Go is here.
- The language reference is here.
- Documentation of Go packages is here.

1.2 Using the *go* command

Once installed, minimal setup is required to use the *go* version of the compiler and development environment: simply make a directory somewhere and create (as in your .bashrc file) the environment variable *GOPATH* to point to it. In this directory create the subdirectories *pkg* and *src*. Optionally, if you want the compiled binaries to live in this directory, create the subdirectory *bin*. You may also want to put the *bin* directory in your execution path by adding it to your *PATH* environment variable.

One can run simple, single package programs without the above setup:

```
go run myprogram.go
```

This is quick and it is useful for exploring language features. However, for compiling more complex programs consisting of multiple packages, a different technique can be used. First of all, install the source code for all packages including the calling program in *src* under their own directory; i.e., *aaa.go* would go in *src/aaa* etc. Then, in the main directory, run

```
go build myprogram
```

where *myprogram.go* is the top-level calling package (with package name *main*). The compiler will look for all the needed packages in *src*, compile them and link them with the main package. The executable is left in the main directory. If you want to install them automatically in the *bin* directory, replace *build* with *install* in the above command.

If Go programs are built in different directories, then *GOPATH* should be separated by a series of paths separated by colons. Alternatively, *GOPATH* can be set on the fly, for instance, in a Makefile:

```
myprogram : src/myprogram/myprogram.go src/subpackage/subpackage.go
export GOPATH=‘pwd‘; go build myprogram
```

typing *make* will then set *GOPATH* correctly and build the program, leaving the executable in the main directory.

By default, *go* links all called libraries and sub-packages statically, which results in very large executables, mainly because the system stuff is big. In order to make system libraries link in a shared fashion (which is the normal way of doing things in Linux), run

```
go install -buildmode=shared -linkshared std
```

This generally has to be done as root, but it only needs to be done once as long as the compiler isn't updated. (It is perhaps a bug that this isn't done on the installation of the *go* compiler!) Then compile your program as follows:

```
go build -linkshared myprogram
```

If it is desired to make sub-packages link dynamically, then on each one run

```
go install -buildmode=shared -linkshared subpackage
```

etc.

1.3 Using the *gccgo* compiler

Code compiled with the gccgo compiler fits in better with the standard way of doing things in linux. For a program consisting of a single package (named *example* here), just run

```
gccgo -o example example.go
```

The executable is called *example*.

If a program consists of two or more packages, compile all packages to *.o* files first, and then create the executable:

```
gccgo -c sub1.go
gccgo -c sub2.go
gccgo -c example.go
gccgo -o example example.o sub1.o sub2.o
```

Note that the lowest level packages (i.e., those not importing other user packages) must be compiled to *.o* files first, followed the packages at the next level up, etc., until the top level is reached. All of this can be automated in Makefiles by making higher level *.o* packages depend on the lower level *.o* packages. The normal gcc optimization flag *-O* works. The *-g* flag works also, and prints the offending line when a runtime error occurs. There is a manual page on *gccgo*, but it is not very informative. See the above link instead.

1.4 *Go* or *gccgo*?

Which *go* compiler should you use? Both work, but other considerations enter.

Gccgo has the advantage that it fits in nicely with existing *gcc*-family compilers. With its superior optimization, *gccgo* tends to produce faster code than *go*. If arrays are statically declared, *gccgo* is as fast as *C* in the execution of a sample heat equation solver. However, this is a significant constraint on programming style in *go*, so this may not be a consideration. With dynamically allocated arrays and slices, the heat equation solver takes about 2 times as long as equivalent *C* code. This is probably due to the overhead of array bounds checking and other safety measures.

The main operational difference between the two compilers is that *go* operates in its own peculiar environment. If the root directory of this environment is called *mygo*, then *src*, *pkg*, and *bin* subdirectories contain respectively source code for packages, compiled packages, and executable programs. An environment variable *GOPATH* must be set to the root directory for the *go* compiler to function. Within these constraints, *go* has an impressive infrastructure, but one must learn to live within the environment. Binary files can be moved anywhere once compiled. However, in order to have *go* code in multiple directory trees, one must reset the *GOPATH* environment variable for each one, or alternatively set *GOPATH* to a colon-separated list of root directories. An alternative is to create a Makefile that redefines this variable on the fly:

```
myprogram : src/myprogram/myprogram.go
export GOPATH=‘pwd‘; go build myprogram
```

This way there will be no unexpected name clashes between different directory hierachies. (Note that *go build* leaves the compiled program in the root

directory while *go install* compiles the program and moves the binary to *bin*.) Making packages available to others on the same computer can be done by defining, for instance, another root directory in */usr/local/lib/localgo* and installing these common packages there. In this case one would set one's *GOPATH* variable as

```
export GOPATH=‘pwd‘:/usr/local/localgo
```

This way, the compiler will check both directories for source code and compiled packages.

One current disadvantage of the *go* compiler is that all libraries are statically linked to the program, which results in very large executable files.

1.5 Getting started

Here is a simple Go program that illustrates some basic characteristics of the language. Cut and paste it into your editor and save it as the file *simple.go*. You can then execute it with the command “go run simple.go”.

```
// a very simple go program
package main

import (
"fmt"
"math"
"math/cmplx"
)

func main() {
fmt.Println("Hi there!")
fmt.Println("3 + 5 =", 3 + 5)
fmt.Println("math.Pow(5,3) =", math.Pow(5,3))
fmt.Println("math.Sin(2.5) =", math.Sin(2.5))
fmt.Println("1<<3 =", 1<<3)
fmt.Println("17>>3 =", 17>>3)
fmt.Println("3 > 5 =", 3 > 5)
fmt.Println("!false =", !false)
fmt.Println("false && true =", false && true)
fmt.Println("false || true =", false || true)
```

```
a := 3 + 4i
aconj := cmplx.Conj(a)
fmt.Println("a := 3 + 4i, a*, |a|, real(a) =",
a, aconj, cmplx.Abs(a), real(a))
}
```

- Go program source code files must end with the extension *.go*. Each file can contain one or more Go functions and must have a package name at the beginning. The top-level package for a program should be named *main*.

- The entry point of the main program occurs through a function named *main*. This function has no arguments, but the *()* that would enclose any arguments must be included anyway.

- Go depends on the ability to load other packages to accomplish tasks not included in the basic language structure. This is done via the *import* statement.

- Comments in Go begin with // and end at the end of the line. The /* ... */ form used in C is also supported, but is less favored.

- Unlike Python, Go does not depend on indentation. However, indentation makes the code much more readable. Emacs comes with a Go mode if Go is installed. To activate it, put the line *(require 'go-mode-load)*' in your *.emacs* file. Hitting tab while editing a Go source file with Go mode will produce proper indentation. Continuation lines in Go work automagically, with no special syntax.

- Like C, Go uses curly braces to enclose blocks of code, such as the code in a function. Unlike C, semicolons are not normally used at the end of statements.

- The *fmt* package contains basic formatted input and output routines. Here we only invoke the *Println* function, which works the way the *print* statement does in Python3, except that strings must be enclosed in double quotes. Notice that all publicly accessible routines in a package must be capitalized and must be prefixed with the package name.

- Math operators work pretty much as they do in C.

1.6 Emacs Go mode

There is a mode for emacs that makes editing Go files much easier. Assuming that *go-mode.el* and *go-mode-autoloads.el* have been placed in the directory */usr/local/share/emacs/site-lisp/gomode*, add the following lines to your *~/.emacs* file:

```
;; Go mode
(add-to-list 'load-path "/usr/local/share/emacs/site-lisp/gomode/")
(require 'go-mode-autoloads)
```

The *go-mode* package for emacs can be downloaded from here. More information on it is given here. Note that the *go-mode* package used to be included in the Go distribution, but this is no longer the case – one needs to download it from Github.

Vi users are on their own!

1.7 Speed of execution

Go is a compiled language, so one would expect it to execute more rapidly than interpreted languages. In my experience with a few non-trivial applications, Go executables take roughly 1.5 to 3 times as long to execute as equivalent C code, with Google *go* tending to be about twice as fast as *gccgo*. Presumably the automatic garbage collection and the bounds checking are responsible for this slowdown. Well worth it in my opinion! However, speeds equal to that of C code can be obtained using *gccgo* if data are stored in static, preallocated arrays, with slices derived from these arrays. This has the disadvantage that changing the size of these arrays requires recompilation. The tradeoff may be worth it in speed-critical applications.

2 Language elements

This section provides the basics of the Go language. Not all language elements are included, just those that I find useful for scientific computation. Notably missing are discussions of the *interface* type and explicit use of pointers. (Pointers are used implicitly in Go to great effect.)

2.1 Elementary data types

Here is a list of elementary data types:

- *bool*: Booleans have two possible values, *true* or *false*. Unlike C, booleans have their own (unspecified) internal representation and are not interchangable with ints.
- *rune*: A rune is a Unicode representation of a character. ASCII is a subset of Unicode. Runes are represented internally by a 32 bit integer. A single rune is surrounded by single quotes in Go.
- *string*: A string is a sequence of zero or more runes (without the single quotes) delimited by double quotes. Go strings are not interchangable with C strings.
- *byte*: The byte is an unsigned 8 bit integer.
- *int*: Ints are either 32 bit or 64 bit integers, depending on the implementation. One can also specify integers of length 8, 16, 32, and 64 bits as well as unsigned versions, e.g., *uint32* – normal integers are signed.
- *float32, float64*: Go supports both single (float32) and double (float64) floating point numbers, but float64 is definitely preferred; all math library functions require and return float64. (This is consistent with other modern languages such as Python, which does not even support single precision floats.)
- *complex64, complex128*: Complex numbers are pairs of floats. Again, complex128 is preferred.

2.2 Variables, expressions, and assignment statements

As in C, all variables must be declared. For instance

```
var n int
var b, c bool
var x, y, z float64
var a float32 = 10.4
var i, j, k int = 1, 2, 3
var c complex128 = 4 + 3i
```

```
var q rune = 'x'
var xxx string = "this is a string"
```

A declaration can be anywhere in the function in which the variable is used, as long as it occurs before use. If a variable is not explicitly initialized in the declaration, it is implicitly initialized to zero.

The scoping rules of Go are similar to those in C. In general a variable can be referred to only within the function in which it is defined. If it is defined outside a function, it is accessible everywhere in the package unless a new variable with the same name is defined inside a function. In that case, the new variable takes precedence inside the function. Variables are not accessible outside the package in which they are defined.

Expressions and assignment statements are generally similar to those in C. Referring to the above variable declarations:

```
x = y + z
a = float32(x*z)
z = float64(i)*x
k = j + int(x)
c = complex(x,y)
x = real(c)
y = imag(c)
```

Note that types cannot be mixed and type conversions must be explicit. Conversion of floats to ints truncates toward zero. In converting to and from complex quantities, consistency between float types must be maintained.

Unlike C, variables can be defined on the fly in the first use of the variable using *:=*.

```
l := 3
r := 5.4
var a float32 = 45
abc := 21.7*a
```

Since the type of the variable is not explicitly specified, Go guesses at the type. For instance, in the above examples, *l* is given the type *int*, whereas *r* is given the type *float64*. However, *abc* is given the type *float32*, since *a* is defined to have this type. (Tricky! However, the default choices show the preference for 64 bit floats.)

Unlike C, Go has strings as a basic type. The only basic operation on strings is concatenation, using the + sign. For example, the following

```
var a, b string = "Hello", "world!"
c := a + " " + b
fmt.Println(c)
```

produces the result *Hello world!*.

2.3 Constants

It is generally desirable to define constants symbolically. The *const* construct is useful for this purpose:

```
const pi = 3.14159
const (
     a = 4.57
     b float64 = 5.23768987
     i, j, k int = 1, 2, 3
)
```

Constants can be defined anywhere, but if they are put near the beginning before any functions are defined, they are globally available everywhere in the file. The type specification is needed if the constant must have a definite type, but its use is optional. *Pi* is a globally defined constant available in all Go programs.

2.4 Functions

Functions contain the executable code in Go programs. They are defined as follows:

```
func function_name(input_list) (return list) {
....
}
```

The *input_list* and the *return_list* contain variables separated by commas. Each variable is followed by a type declaration, e.g.,

```
func myfunc(mystr string, myflt float64) (return1 int, return2 string) {
    ....
}
```

A *return* statement returns the current state of the specified return variables. Multiple return statements can exist in different branches of the code. The function name *main* is reserved for main or highest level routine in a Go program. Information on functions as methods are discussed in the section on structs.

Variables in the input list cannot be modified in the function in a way that affects the calling program. In this way Go is like C. However, also like C, if a variable in an input list is a pointer, the data to which the variable points can be modified. (More on pointers later.)

2.5 Branching

Two means of branching are supplied in Go, the *if* statement and the *switch* statement. Below is an example of the former:

```
package main
import "fmt"

func main() {
fmt.Println(factorial(12))
}

func factorial(n int) (fn int) {
if n > 1 {
return n*factorial(n - 1)
} else {
return 1
}
}
```

This function computes the factorial of a number (and by the way, demonstrates that functions can be called recursively in Go). It takes advantage of the fact that $n! = n(n-1)!$, evaluating the factorial of smaller and smaller integers until the factorial of 1 is reached, in which case it simply returns 1. The argument of the *if* statement must be a boolean expression. The *else* clause is optional and must be written after the closing *}* of the *if* statement, not on the next line (unlike C). Another option not used here is the *else if ...* clause, which pretty much works as you might expect.

The *switch* statement allows you to take various actions for various values of an expression. For example:

```
package main
import "fmt"

func main() {
mycolor := "yellow"

switch mycolor {
case "red":
fmt.Println("Stop!")
case "yellow":
fmt.Println("Go fast!")
case "green":
fmt.Println("Go!")
default:
fmt.Println("Panic!")
}
}
```

This program tests various possible stop light colors against the color you see and prints a course of action if the expression is equal to the argument of a particular *case* clause. The *default* clause covers all the other possibilities. If this clause isn't present, the flow of control falls through to the statement following the *switch* statement. If the expression matches a particular case, the statements in that *case* clause are executed before control passes to the next statement.

2.6 Arrays, slices, and pointers

The example below illustrates some properties of arrays. Cut and paste into a file and run using *go run file_ name*.

```
package main

import (
"fmt"
)
```

```
func main() {

// declare an array
var a [3]float64
a[0] = 3.5
a[1] = 7
fmt.Println(a)

// assigning an array
b := a
fmt.Println("b =", b)

// declare an array on the fly
c := [...]string{"first element", "second element", "third element"}
fmt.Println(c)

// slicing an array
fmt.Println(a[0])
fmt.Println(a[0:2])
fmt.Println(a[:2])
fmt.Println(a[1:])
fmt.Println(c[:2])
d := c[:2]
fmt.Println("d =", d)
c[0] = "blah!"
fmt.Println("d =", d)
}
```

Declaring an array with a *var* statement results in an array of the specified type. All arrays have their size fixed at compile time, as indicated by the number in square brackets. Array elements are initialized to zero and are indexed from zero. Array elements can be changed by assignment.

If an array is defined on the fly, then in addition, initial values for the array elements can be specified as shown in the *c :=* statement. The *...* indicates that the compiler determines the array size from the specified elements.

Sequential segments of an array can be defined the *slice* mechanism, much as in Python. For instance, an array index of the form *[1:3]* extracts a *slice*,

which is like a sub-array including (in this case) elements 1 and 2 of the original array. Recall that arrays are indexed from zero, so that element 0 and all elements after 2 are dropped. If the first element of the index is omitted, the slice starts from the beginning of the array, whereas if the last element is omitted, it ends at the end of the array.

Slicing doesn't copy data, it just sets up pointers to a segment of the original array. Thus, if an array element is changed, the corresponding element of the slice is also changed. If it is desired to make the slice independent of the original array, the *copy* command can be used:

```
number_of_elements := copy(slice_var, array_var[start:end])
```

Slices aren't just a mechanism to pick out parts of arrays; they are raised to the level of variables with values in Go. Slices may be allocated with a variable declaration

```
var a []float64
b := [...]float64{1,2,3,4}
a = b[1:]
```

which creates an empty slice to which a segment of an array can be assigned, or on the fly

```
b := []float64{1,2,3,4}
```

which creates a slice with 4 elements assigned the specified values. The 4 elements are actually held in an underlying, unnamed array. Notice that a slice definition differs from that of an array in that the // is empty.

A slice can also be declared using the *make* function,

```
c := make([]byte,100)
d := make([]float32, 200, 500)
```

which in the first case allocates space for 100 bytes initialized to zero, which can be accessed by indexing the variable *c* like an array. In the second case the slice initially has 200 *float32* elements initialized to zero, which can be expanded to a maximum of 500 elements without allocating more space. Slice variables always point to an underlying array. If they are expanded (say, via an *append*) to a greater size, a new, larger array must be allocated. Efficiency considerations suggest that the capacity of slices should generally be set to the maximum likely size of the slice when created.

Another way that arrays and slices differ is that arrays are passed in function calls by value, which copies the entire array, whereas slices are passed by reference, in which only pointer information is copied; the data remain in the original array, a much more efficient process.

Multidimensional arrays are simply arrays of arrays (two dimensions), arrays of arrays of arrays (three dimensions) etc. The same goes for slices. The following program shows how to use both:

```
package main
import (
"fmt"
)

func main() {
var a [3]([2]float64)
var b [3][2]float64
a[1][0] = 23.7
b[2][1] = 4e15
fmt.Println(a,b)

c := make([][]float32,3)
fmt.Println(c)

for i := 0; i < 3; i++ {
c[i] = make([]float32,2)
}
c[2][1] = 89.23
fmt.Println(c, len(c))
}
```

The variables *a* and *b* are both arrays with their usual fixed sizes. The two variables are equivalent, with the declaration of variable *b* being a shortcut of that for variable *a*. Variable *c* is a multidimensional slice. Note that the on-the-fly definition of *c* creates a slice of 3 empty *float32* slices. The subsequent loop is needed to allocate space for these sub-slices. These sub-slices need not be of the same length.

Sometimes one needs to determine the length or number of elements in a slice or array. The *len* function does the trick. For example *len(c)*. Note

that for a multidimensional slice, the length is that of the first dimension only, as the above sample program shows when it is run.

The built in *append* function allows new elements to be appended to an existing slice:

```
a := []int{1,2,3}
a = append(a,4,5)
```

If the underlying array supporting the slice isn't big enough, a new, larger array is automatically allocated.

The C language uses pointers with gay abandon, often with disastrous results. Go uses pointers with more discretion and their explicit use is typically less common. Nevertheless, their use is sometimes necessary. For instance, passing an array to a subroutine as an explicit argument is not desirable, as the whole array is passed, which means (1) that this is very inefficient if the array is large and (2) changes in the array inside the subroutine are not returned to the calling function. Mostly this is taken care of by passing a slice of the array, e.g.,

```
mysubroutine(myarray[:])
...
func mysubroutine(myslice []float64) {
...
```

rather than

```
mysubroutine(myarray)
...
func mysubroutine(myarray [arraysize]float64) {
...
```

This is both more efficient and allows changes made in the subroutine to appear in the calling function.

Occasionally, it is necessary for the subroutine to access the underlying array in a slice directly, e.g., when interfacing with a C function. This may be done with pointers as follows:

```
mysubroutine(&myarray[0])
...
func mysubroutine(myarray *float64) {
...
```

The ampersand in the subroutine call indicates that the argument to the subroutine is a pointer to the first element of the array. (This also works with slices, since slice elements are ordered in the same sequence as the underlying array elements.) The asterisk in the function definition indicates that "myarray" is a pointer to a float64 value, which in this case is the first array element.

Though valuable in special cases, explicit pointers should normally be avoided.

2.7 Looping

There is only one looping mechanism in Go, the *for* loop. This mechanism is more versatile than the C language *for* loop, incorporating some of what Python does as well. For example:

```
package main
import "fmt"

func main() {

// C-like loops
for i := 0; i < 5; i++ {
fmt.Println(i, i*i)
}

// like a while loop
x := 100.0
for x > 42 {
x = 0.8*x
fmt.Println("x =", x)
}

// more Pythonish in flavor
a := []int{1, 2, 3, 5, 8, 13}
for j, val := range a {
fmt.Println("index =", j, "-- value =",val)
}
```

```
// the forever loop
i := 0
for {
i = i + 1
fmt.Println("i =", i)
if i > 10 {
break
}
}
}
```

The first case is like a C *for* statement except that the parentheses are missing (and not needed). The second is like a C *while* statement. The third is more Python-like; the *for* statement loops over the elements of a slice, returning the index (starting at zero) and the value of the slice element for that index. If the index is not needed, just substitute an underscore _ in its place. If the value is not needed, just omit it. Finally, the last loop will run forever unless the *break* condition is satisfied.

2.8 Structures

A *struct* allows one to group together a collection of variables in a single entity. For example:

```
package main
import (
"fmt"
)
type Fields struct {
nx int
vx []float64
p []float64
}
func main() {
nx := 5
vars1 := Fields{}
vars1.nx = nx
vars1.vx = make([]float64, nx)
vars1.p = make([]float64, nx)
```

```
for ix := 0; ix < vars1.nx; ix++ {
vars1.vx[ix] = float64(ix)
vars1.p[ix] = vars1.vx[ix]*vars1.vx[ix]
}
fmt.Println(vars1)
}
```

This program defines an empty structure with the new type *Fields*. The integer *nx* is assigned a value of 0 and the following slices are empty. The first two statements in the function *main* define a new instance of the structure and an integer *nx* used to define the size of the slices in the structure. (Note that there is no name clash between this integer and the similarly named integer inside the structure.) The next statement gives *nx* a value and this value is used to assign zeroed slices of size *nx* in the following two statements. Note that these assignments could also be made in the data list of the structure declaration

```
vars1 := Fields{nx, make([]float64, nx), make([]float64, nx)}
```

The choice of which to use is a matter of esthetics. Following these declarations, a *for* loop is used to assign values to the slices, after which the results are printed. Cut and paste this code into a file and run it to see what it does. Initial values can also be assigned in the usual way inside the structure type declaration, but this may be less useful than doing it after an instance of the structure has been defined.

If a structure needs to be passed to a subroutine, it is sometimes valuable to pass a pointer to the structure, especially if the structure contains a large amount of data:

```
mysubroutine(&vars1)
...
func mysubroutine(x *Fields) {
...
x.nx = 2
y := x.p[2] + 1000.0
...
```

This is because structures are passed by value, which means that the component array data are also passed. Slices as components of a structure are passed as well, but since slices only contain pointers to array data rather

than the data itself, the size is small and changes in slice elements are seen in the calling function.

The same "&/*" notation is used for structures as for arrays and slices. Note that unlike C, referencing the elements of a structure has the same notation as referencing the elements of a structure pointer, as indicated above.

Note that a structure type with a capitalized name is global, so that other packages can see it.

2.9 Maps

Maps are like Python dictionaries. They consist of an unordered list of key-value pairs. For a given map, the keys and values must be all of the same type. For instance, a map with keys in the form of strings representing peoples' names and values representing their ages, one could proceed as follows:

```
package main
import "fmt"

func main() {
m := make(map[string]int)
fmt.Println(m)

m["George"] = 42
m["Sally"] = 23
m["Babe"] = 2

fmt.Println(m)

delete(m, "George")
fmt.Println(m)

element, ok := m["Sally"]
fmt.Println(ok, element)
element, ok = m["Anna"]
fmt.Println(ok, element)
}
```

The *make* function creates an empty map (unless values are assigned on the fly) and entries are made by assignment after creation. In this example the

keys are the names and the *values* are the numbers. Key-value pairs can be added by assignment and values can be retrieved by indexing with keys. An optional second argument resulting from indexing is a boolean variable indicating whether the map contains the specified key. Key-value pairs may be deleted using the *delete* function and the length of a map can be obtained using the *len* function.

Keys must be an orderable type such as strings or numbers. Values can be pretty much anything including structures, arrays, slices, or other maps.

2.10 Methods

Methods are special functions that act on derived (rather than primitive) types in the manner of object oriented programming. For instance:

```
package main

import (
"fmt"
"math"
)

type Vector []float64

func main() {
size := 5
vec := vmake(size)
vec.vadd(7)
vlength := vec.vlen()
fmt.Println(size, vec, vlength)
}

func vmake(size int) (v Vector) {
return make([]float64, size)
}

func (v Vector) vadd(a float64) {
for i := range v {
v[i] = v[i] + a
```

```
}
}

func (v Vector) vlen() (a float64) {
length := 0.0
for _, val := range v {
length = length + val*val
}
return math.Sqrt(length)
}
```

This program defines a *Vector* type, which is really just a slice of 64 bit floats. The function *vmake* is a function that creates a zeroed vector of the specified length. However, *vadd* and *vlen* are *methods* that apply only to the *Vector* type.

Note how methods are invoked in the main routine; *vec* is the name of a vector and *vec.vadd* invokes the first method, while *vec.vlen* invokes the second. The first method has no return value; instead, it adds *a* onto each element of the vector in place. The second method leaves the vector unchanged but returns the square root of the sum of the squares of the elements.

The method definitions differ from the definitions of ordinary functions in that a dummy vector is defined in the parenthetical expression between the *func* and the method name. This is a placeholder for the vector that is being acted on in the main function.

2.11 Scoping rules

The *scope* of a variable is the segment of a program in which the variable is visible. In Go, variables declared within a block of code as well as variables in the code block in which the original block is nested (the *nesting* block) are visible. However, variables defined in a nested block are not visible to the nesting block. A *block* is a segment of code delimited by curly braces *{}*, or at the uppermost level, code in the package itself. If a variable is defined in a block with the same name as a variable in the nesting block, the latter is hidden in favor of the local definition.

All functions and variables in packages are hidden from other packages that might be compiled into the same program except for names that begin

with an upper case letter. This includes variables defined as members of a structure. Capitalized functions and variables are therefore the only things in Go packages with *global* scope.

2.12 Concurrency and parallel processing in Go

Go is capable of running multiple threads (lightweight processes that share the same memory). These threads may execute on the same processor core (concurrency) or different cores in the same CPU (parallel processing). Two main tools are used, *goroutines* and *channels*. Goroutines are functions that run as a new thread, while channels pass data from one goroutine to another and to/from the main program. Synchronization of goroutines is enforced by channels; a goroutine expecting a channel message from another goroutine waits until the message is received before continuing.

Here is an example in which goroutines are used to create a table of sines in parallel on a computer with multiple cores or concurrenty on a single core machine:

```
// goroutine example -- parallelize the computation of a table of sin
// function values
package main

import (
"fmt"
"math"
)

func main() {
pi := 3.14159
nx := 32 // size of table -- must be a power of 2 for convenience
np := 4  // number of threads -- must be a power of 2 up to ntheta

// create a channel for communication between goroutines
// and the main program -- this is a buffered channel with
// buffer size equal to the number of threads
tomain := make(chan int, np)

// create the slices to contain the angle x and sin(x) values
```

```
x := make([]float64,nx)
sinx := make([]float64,nx)

// loop on threads, creating np goroutines to split up the work
for ip := 0; ip < np; ip++ {
go func (ip int) {

// determine the segment of the table generation
// to be done by this goroutine, based on the
// value of ip
n := nx/np
for i := ip*n; i < (ip + 1)*n; i++ {
x[i] = pi*float64(i)/float64(nx - 1)
sinx[i] = math.Sin(x[i])
}

// tell main program that we are done
fmt.Println("reporting done from goroutine",ip)
tomain <- ip
} (ip)
}

// wait until all goroutines report completion
for ip := 0; ip < np; ip++ {

// note that the order in which goroutines finish
// is not necessarily the same as the order in which
// they are started
goroutine_reporting := <-tomain
fmt.Println("goroutine",goroutine_reporting,"reporting")
}

// print the results
for i := 0; i < nx; i++ {
fmt.Println("x, sin(x):",x[i],sinx[i])
}
}
```

The goroutine is actually an example of an "anonymous function", i.e., a function that is defined in place and is executed once as it is reached. The stuff in parentheses after "func" defines the input parameters as in an ordinary function call. The stuff in parentheses at the end of the goroutine lists the arguments supplied by the calling program. Note that the usual scoping rules apply, with all variables defined in the calling routine available to the goroutine. So, information can be supplied to the goroutine by the main routine via this scoping rule or through explicit transmission via input parameters.

The variable *ip* is tricky. Since it takes on different values for the invocation of different goroutines, it should be passed in the goroutine parameter list. Otherwise its value will be unpredictable, with generally dire results.

There are two types of channels, buffered and unbuffered. An unbuffered channel would have zero in the second argument of *make(chan int, n)*. An unbuffered channel read, as in *<-tomain*, blocks until the first write to the channel, e.g., *tomain <- 1*. A buffered channel unblocks after n reads and can be reused with subsequent writes.

We have used channels of type *int*. However, channels can be defined to have any type and can be used to convey information in addition to acting as a synchronization mechanism. For instance, the above routine could have returned the segment of the sine function table that it had created. However, it is probably best not to do this, as (1) it is more work and (2) it doesn't take advantage of the access to variables defined in the calling routine, through which information can be returned. It may also be slower, though this isn't completely clear.

2.13 Calling C language functions from gccgo

The canonical way of linking C functions to Go is via the *cgo* directive of the *go* command. *Cgo* may be more trouble than it is worth when using *gccgo*, as this compiler has a simple way of implementing the link via the *extern* directive. For example, suppose one wanted to use the C version of the sine function:

```
package main

import (
"fmt"
```

```
)

func main() {
x := float64(1.2)
a := c_sin(x)
fmt.Println(x,a)
}

//extern sin
func c_sin(x float64) float64
```

Compile this in the usual way, but with the C language math libary included:

```
$ gccgo -o main main.go -lm
```

The last two lines of the calling Go program starting with *//extern...* tell Go what to expect from the C function. (The lack of space between *//* and *extern* is necessary!)

Slices and strings cannot be passed directly between Go and C, as they are represented by internal structures in Go that could change with updated versions of the compiler. However, arrays are easily passed. More properly, array *pointers* may be passed.

The problem with arrays (unlike slices) is that they are defined statically at compile time – shades of the bad old days of Fortran! Brute force is probably the best solution here; define the array as large as one could possibly imagine that it needs to be and reference what piece of it is needed by slice operations. (Memory is cheap!)

It is best to locate the array in the Go program rather than in the C function for a number of reasons. Here is an example of modifying a Go array in a C routine:

```
// This is the calling Go program
package main

import (
"fmt"
)
const bufsize int = 20
```

```
func main() {
var a [bufsize]float64
slicesize := 6
if slicesize > bufsize {
fmt.Println("buffer not big enough")
}

// put some stuff in a
for i := 0; i < slicesize; i++ {
a[i] = float64(slicesize - i)
}

// define a slice of a for convenience and print before
aslice := a[0:slicesize]
fmt.Println("a before:",aslice)

// pass a to and from C routine
c_carray(slicesize, &a)

// print after
fmt.Println("a after:",aslice)
}

//extern carray
func c_carray(asize int, a *[bufsize]float64)

============================================================

/* This is the called C function */
#include <stdlib.h>

void carray(int size, double *x) {
  int i;

  /* modify the array */
  for (i = 0; i < size; i++) x[i] += i*i;
}
```

Note that the construct *a *[...* tells Go to present the C function with a pointer ("*") to the Go array. C likes this, as arrays in C are really pointers to the actual array. Go refers to the C function by the C function name with "*c_*" prefixed. The argument list in the C function is the same as in the Go function, but translated into C language terminology.

Strings can be passed to a C function by first converting them to a null-terminated byte array. Here is an example of how to do this:

```
// This is the calling Go program
package main

const bufsize int = 20

func main() {
var a [bufsize]byte
mystring := "Hello!"

// give cstring a slice pointing to a -- result appears in a
cstring(a[0:bufsize],mystring)
c_strprint(&a)
}

func cstring(myslice []byte,mystring string) {
strlen := len(mystring)
if strlen + 1 > len(myslice) {
panic("string too long")
}

// convert string to byte slice
x := []byte(mystring)

// transfer bytes to myslice, which puts them in a
for i, mybyte := range x {
myslice[i] = mybyte
}

// null terminate for C
myslice[strlen] = 0
```

```
}

//extern strprint
func c_strprint(a *[bufsize]byte)

============================================================

/* This is the called C function */
#include <stdlib.h>
#include <stdio.h>

void strprint(char *x) {
  printf("%s\n",x);
}
```

Note that the string could be modified in the C function and returned to the Go program as in the previous example.

3 Common mistakes in writing Go programs

Go is a very safe language; it is virtually impossible to make it dump core. The language definition also catches many common errors that would pass unnoticed in C. The usual problem in Go programming is figuring out what an error message really indicates. Here are some common errors that at first appear mysterious.

1. Continuation lines: Go has no special syntax to indicate line breaks in long statements. So, if a break is made at a point where the code before the break point can be interpreted as a complete Go statement, error messages can be very mysterious. The best fix is either to never break long lines or to break after an operator or a comma.

2. Variable defined in an *if* statement: A variable defined in an *if* block is not visible in the block containing the *if* statement. If it is desired to extract values created in an *if* (or other) block, the variables containing the values should be defined in the the nesting block and modified in the nested block by an assignment statement.

3. Fussy type consistency: A slice is not an array, even if the element types are the same. *Float32*, *float64*, and *int* don't mix. Go will let you know if you flout these rules. By the way, this fussiness goes a long way toward keeping one out of trouble. Use explicit type conversions. It is best to do this even with literals: *float64(2.0)* rather than *2.0* for instance.

4. Define vs. assign: The Go compiler will let you know when you try to redefine a variable or assign to a variable that hasn't been defined.

5. "=" and "==": As in C, the former denotes assignment and the latter a logical equals test. Unlike C, Go won't let you inadvertently do an assignment in an *if* statement test clause.

6. "|" and "&" vs. "||" and "&&": The first two are for bit manipulation, the second two are for logical expressions, as in C. Unlike C, logical variables in Go have their own type, *boolean*, for which bit manipulation is not an allowed operation, thus avoiding confusion between the first and second pairs.

4 Standard packages

For information about these packages, visit the Go package web page. To use these packages, they must be imported with the specified name.

4.1 Errors

With the *errors* package one may create customized error messages. For example

```
package main
import (
  "errors"
  "fmt"
  "os"
)
func main()
  ...
  err := myread(readrequest, ...)
```

```
  if err != nil {
    fmt.Println(err)
    os.Exit(1)
  }
  ...
}

func myread(readrequest int, ...) (err error)
  ...
  READ STUFF
  ...
 if readsize < readrequest {
    err := errors.New("Read did not complete")
  } else {
    err := nil
  }
  return
}
```

The *READ STUFF* part does a file read that returns a *readsize* that may be less than the requested size *readrequest*. If it does, then the returned error is non-nil, containing the specified message.

4.2 Conversions

Sometimes numbers need to be converted between integers and floats and between integers of different length as well as floats of different length. There are four types of such conversions, *casts*, in which bits are rearranged to provide the same number (to the extent possible) in, say, integers and floats, *interpretations*, in which a particular pattern of bits is viewed, say, as an integer or a float, *external representations* of numbers longer than single bytes which can be either *big-endian* or *small-endian*, depending on the ordering of bytes, and *formatted conversion* between bit representations and human-readable strings.

4.2.1 Casts

Some examples of casts are:

```
package main

import (
"fmt"
)

func main() {
a := 4.3
b := -5.1
ia := int(a)
ib := int(b)
c := float64(ia)
fmt.Println("a,ia =",a,ia,"  b,ib =",b,ib,"  c =",c)
}
```

Notice that *int* truncates a float toward zero for both positive and negative floats. The function *int* produces integers of 32 or 64 bits, depending on the compiler. More specific integer conversion functions would be *byte*, *int16*, *int32*, and *int64*. There are also versions that produce unsigned integers, e.g., *uint16*. Conversion from integers to floats is done either with the *float32* or the *float64* functions, depending on whether 32 bit or 64 bit floats are desired. All of these functions can be used in integer to integer and float to float conversions as well.

4.2.2 Interpretations

Interpreting a given bit pattern as either an integer or a float can be done using functions from the *math* package:

```
package main

import (
"fmt"
"math"
)

func main() {
Since the type of the variable is not explicitly specified, Go guesses at the
ia := uint32(21)
```

```
fa := math.Float32frombits(ia)
fb := float32(21)
ib := math.Float32bits(fb)
fmt.Println("ia =",ia,"  fa =", fa,"  fb =",fb,"  ib =",ib)
}
```

Running this program shows that the number *21* has very different integer and float representations. Note that unsigned integers must be used as the input to *math.Float32frombits* and *math.Float32bits* returns unsigned integers. 64 bit versions of these functions also exist.

Similar alternate interpretations exist between strings and byte slices. For example

```
package main
import (
"fmt"
)

func main() {
ms := "Hello!"
bs := []byte(ms)
ms2 := string(bs)
fmt.Println(ms, bs, ms2)
}
```

yields

```
Hello! [72 101 108 108 111 33] Hello!
```

No bits change, simply the interpretation of the bits.

4.2.3 External representations

One must worry about byte order when dealing with external byte streams. Here is a routine that converts four external bytes into an unsigned integer assuming that the external bytes are in big-endian order:

```
func bytesToUint32(b []byte) uint32 {
i := uint32(b[0])<<24 | uint32(b[1])<<16 | uint32(b[2])<<8 |
uint32(b[3])
return i
}
```

For little-endian byte ordering, use *uint32(b[3])<<24*, etc. The pattern for uints of different lengths is an obvious extension of this routine.

To produce a big-endian stream bytes from an unsigned integer, use:

```
func uint32ToBytes(ui uint32) []byte {
b := make([]byte,4)
b[0] = byte(ui>>24)
b[1] = byte(ui>>16)
b[2] = byte(ui>>8)
b[3] = byte(ui)
return b
}
```

These tasks can be done with routines in the *bytes* and *encoding/binary* packages, but they involve a lot of extra boiler plate.

4.2.4 Formatted conversion

To convert between the internal binary representation of a number and the formatted ASCII (really UTF-8) representation, use the package strconv. The following commands convert integers and floats from binary to ASCII:

```
int_string := strconv.FormatInt(binary_int64, 10)
float_string := strconv.FormatFloat(binary_float64, fmt, digits, 64)
```

In the integer conversion, the binary form of the integer must be *int64*. It is safest to force this with a cast: *int64(24)*. The second number in the integer conversion is the base of the string representation. Change to *8* for octal and *16* for hexadecimal. In the float conversion, the float must be *float64*; use a cast as for the integer case if necessary. The *fmt* must be a single character and can be (among other things) *'f'*, *'e'*, or *'g'* as in C language formatted print statements. The *digits* argument specifies the number of significant digits in the ASCII representation.

To convert from ASCII to binary, use the following:

```
binary_int64, err := strconv.ParseInt(int_string, 0, 0)
binary_float64, err := strconv.ParseFloat(float_string, 64)
```

The second and third arguments in the integer conversion represent the base of the integer and the maximum bit size of the converted integer. The default

values, given by the zero arguments are generally satisfactory. (Regardless of the specified bit size, an *int64* is always returned.) The *64* in the float conversion is the maximum bit size allowed in the conversion. This could alternatively be set to *32*, but generally there is no point in doing this. Irrespective of this parameter, a *float64* is always returned. Note that the returned *err* argument is not optional. Generally, this is not equal to *nil* if the input string is malformed, and should probably be checked. If you are feeling lazy, it can be ignored by putting _ in its place, but this is not recommended except in casual programs where such errors are not critical.

Conversions between ASCII and binary can also be made using formatted input and output; see below.

4.3 Operating system services

Here is the documentation for the *os* package. This package contains basic routines for interacting with the operating system. Import the *os* package.

4.3.1 Command line arguments

To retrieve the command line arguments:

```
args := os.Args
```

args is a slice of strings containing the command line arguments, the first being the name of the program as in C. The package *flag* provides a convenient tool for processing flag-type command line arguments. Refer to this package for further information.

4.3.2 Exiting a program

There is no builtin function in Go to exit a program short of reaching the end. Here is a command using *os* that does the trick:

```
os.Exit(exitcode)
```

The *exitcode* is conventionally set to 0 for a normal exit and to some other integer for an abnormal exit.

4.3.3 Executing another program

To execute another program from within a Go program, load the *os/exec* package. To run a command *mycommand* with command line arguments *a1*, *a2*, and *a3*, construct the command using

```
xxx := exec.Command("mycommand","a1","a2","a3")
```

Then run the command with

```
err := xxx.Run()
```

This runs the command, returning no error if the command completes successfully. If it is desired to continue execution of the main program while *mycommand* is running, then start the execution of the command with

```
err := xxx.Start()
```

4.4 Input and output

4.4.1 File handling

The package *os* also provides low-level file handling:

```
package main

import (
"fmt"
"os"
)

func main() {

infile, err := os.Open("osfiles.go")
if err != nil {
panic("Open operation on file failed")
}

stuff := make([]byte, 500)
num, err := infile.Read(stuff)
fmt.Println(num, stuff, err)
```

```
infile.Close()

outfile, err := os.Create("osfiles.go.copy")
if err != nil {
panic("Create operation failed")
}

num, err = outfile.Write(stuff)
outfile.Close()
}
```

The *os.Open* function opens the named file for reading, returning a file pointer *infile* and and error code *err*. If this error code is non-nil, we quit the program with an error message, courtesy of the *Fatal* function in the *log* package. The *Read* method on *infile* returns the number of bytes returned *num* and an error message *err*, which we ignore here. If the read hits an end-of-file, the number of bytes returned is zero and the error message indicates an end-of-file condition. The *infile.Close()* does what one would expect.

Note that if reading from a pipe such as the standard input, *Read* may return fewer bytes than requested even if one is not at the end of the input. In this case, *Read* should be called as many times as necessary to get the rest of the input. This appears not to be an issue when reading from a file.

The io package has a routine that performs as many low-level *Read* operations as needed to get the desired number of bytes:

```
func ReadFull(r Reader, buf []byte) (n int, err error)
```

where the first argument is the file pointer (*infile* in the previous example) and the second argument is the buffer into which the read is to be placed, as in the low-level *Read*. On the other hand, if you want to avoid importing yet another package, here is a stand-alone routine that does the job:

```
// reliable read -- reads from pipes might not get entire desired block
// on the first try -- if err == nil, totalread should equal len(inbuff)
func myreader(fileptr *os.File, inbuff []byte) (totalread int, err error) {
nbytes := len(inbuff)
totalread = 0
for {
nread := 0
```

```
nread, err = fileptr.Read(inbuff[totalread:nbytes])
totalread += nread

// return on error, which generally indicates that
// the requested read size exceeds the remaining bytes
if err != nil {
return
}

// this shouldn't happen
if totalread > nbytes {
panic("read too many bytes!")
}

// quit trying when we have all we need
if totalread == nbytes {
return
}
}
}
```

The error message should be checked for end of file (*EOF*). The total bytes read should always equal the size of the supplied buffer unless an end of file is encountered before the requested number of bytes is read.

The *os.Create* operation works like the *os.Open* operation except it opens a file for writing. If the file already exists, the contents are trashed. The *outfile.Write* method writes the contents of the byte slice to the new file.

In the read and write methods the number of bytes read or written is equal to the length of the byte slice except that if there are fewer bytes remaining in a file on a read operation, only that number of bytes is returned, with zeros in the remainder of the slice. Normally the read and write methods are put in a loop so that files are read or written a chunk at a time.

The *os* package has three pre-opened files, *os.Stdin*, *os.Stdout*, and *os.Stderr*, which give access to the standard input, output, and error as expected. Reading from the standard input would be done via something like

```
stuff := make([]byte, 500)
num, err := os.Stdin.Read(stuff)
```

with similar write methods to standard output and standard error.

4.4.2 Formatted I/O

Here is the documentation. Import the *fmt* package. The best way to learn formatted I/O is via examples. Here is one for printing to the standard output:

```
package main

import (
"fmt"
)

func main() {

// printing to stdout
// unformatted printing -- no newline at end, no spaces
fmt.Print("this is a string -- a number follows: ",3.4," an int: ",42)

// let's put some newlines
fmt.Print("\n\n\n")

// if we want nice spacing and a newline at the end
fmt.Println("a string with no space at end:",42)

// we can include expressions
a := 4.5
fmt.Println("logical expression:",a > 10,"arithmetic:",a + 21)

// fussy control of formatting as in C
fmt.Printf("a float: %10.4f -- an integer: %4d\n",25.773,421)
}
```

The *Printf* command is very similar to that used in C. The formatting elements for floating point numbers include *%f* (ordinary output), *%e* (scientific notation), and *%g* (the lazy man's way of printing floating point). For integers use *%d* for decimal representation, *%o* for octal, and *%x* for hexadecimal. Use *%s* for strings.

Scanning a string for white-space-separated words (i.e., text separated by spaces, newlines, or tabs) is done as follows:

```
package main

import (
"fmt"
)

func main() {

// translate stuff in a string into binary values
instring := "two words 42.7 23"
w1 := ""
w2 := ""
a := float64(0)
b := int(0)
nwords, _ := fmt.Sscan(instring,&w1,&w2,&a,&b)
fmt.Println("#words:",nwords,"instring:",instring,"w1, w2:",w1,w2,"a:",a,"b:",
}
```

The words must match the types of variables listed after the input string in *Sscan*; otherwise the scan will terminate. *Sscan* returns the number of words successfully scanned. An error code is also returned, which we have lazily ignored. Note that pointers to the receiving variables (e.g., "*&w1*" rather than "*w1*") must be used, since otherwise *Sscan* can't assign values to them.

The function *Scan* does the same thing as *Sscan* except that the input string is omitted from the argument list is omitted and replaced by the standard input. *Scanln* is like *Scan* except that the takes a single line at a time. The number of white-space-separated words in the input must match exactly the number and type of output arguments, otherwise the function will fail. *Scanf* is the standard input reader equivalent to the standard output writer *Printf*.

Scanning from and printing to files is done with functions named *Fscan*, *Fprint*, etc. These functions have an extra argument indicating the file pointer for the file to be read or written that is obtained from the *os.Open* and *os.Create* calls. For example:

```
package main

import (
"fmt"
```

```
"os"
)

func main() {

// create a file and write some text to it
fptr, err := os.Create("boojum.txt")
if err != nil {
panic("error in file create")
}
fmt.Fprintln(fptr,"This goes to a file")
fmt.Fprintln(fptr,"This goes to the same file")
fptr.Close()

// read the file and print it out to standard error
// one word per line
inptr, err := os.Open("boojum.txt")
for {
a := ""
n, err := fmt.Fscan(inptr,&a)
if n != 1 {
break
}
if err != nil {
panic("error or eof in scan")
}
fmt.Fprintln(os.Stderr,a)
}
}
```

The other versions of *scan* and *print* operate similarly. Standard input, output, and and error have perpetually open file pointers *os.Stdin*, *os.Stdout*, and *os.Stderr*. The last is used in the above example to print the list of words from the created file.

Note that the *Scan* class of functions is limited for free format string input as they either return one whitespace-separated word at a time or the fields in a statically specified pattern. It is easier to use *os*-level input to read a byte slice, convert to a string (see section 4.2.2), and use the *strings* package

to parse the input. Here is a routine that returns a line at a time as a string with the newline omitted — if there is a carriage return, it is omitted as well:

```
func myline(fileptr *os.File) (s string, err error) {

// we will let inbuff grow as needed
inbuff := make([]byte, 0)
bytebuff := make([]byte, 1)
n := 0
// loop, reading a byte at a time
for {

// get a byte
n, err = fileptr.Read(bytebuff)

// take appropriate action
if n == 0 {
break
} else if n == 1 {
if bytebuff[0] == '\n' {
break
}
if bytebuff[0] == '\r' {
} else {
inbuff = append(inbuff,bytebuff[0])
}
} else {
panic("n neither 0 nor 1")
}
}

// convert to string
s = string(inbuff)

// done
return
}
```

4.5 Strings

The *strings* package has numerous routines for manipulating strings. Here is the documentation. A few of the more useful functions are listed below:

- *func Fields(s string) ([]string):* This splits the string *s* into substrings delineated by white space (spaces, tabs, or newlines). The substrings are returned in a slice of strings.
- *func Replace(s, old, new string, n int) (string):* This replaces the first *n* instances of the substring *old* in the string *s* with the substring *new*. The modified string is returned. If $n < 0$, then all instances are replaced.
- *func Split(s, sep string) ([]string):* This splits a string into substrings separated by *sep*. The substrings are returned in a slice of strings.
- *func ToLower(s string) string:* This returns string *s* with all upper case letters changed to lower case.
- *func ToUpper(s string) string:* This returns string *s* with all lower case letters changed to upper case.
- *func Index(s string, match string) int:* This returns the index of string *s* of the first occurrence of substring *match*. If there is no such substring, it returns *-1*.
- *func Trim(s string, cutset string) string:* This returns a string with characters included in *cutset* at the beginning and end of *s* removed.

Sometimes one needs to sort a slice of strings. This does it in place (a sorted string isn't returned):

- *func Strings(slist []string)*

This function is part of the *sort* package (documentation here).

4.6 Math

4.6.1 Standard math

Here is the documentation. To get the standard math package, use *import "math"*. The mathematics package contains pretty much what one would expect, including the usual absolute value, trig, log, and exponential functions

that occur in the C language math library, with one wrinkle: all functions operate exclusively on 64 bit floats. (Actually, the C library does the same, except that with C's type promotion, one doesn't notice!) So, if you wish to work with float32s, you must do something like

```
var a, b float32
a = 0.8
b = float32(math.Sin(float64(a)))
```

The message here is forget float32s and use float64s for math. The extra memory and computational time is negligible in most cases, and there are times when the increased accuracy and dynamic range are important.

4.6.2 Complex math

Here is the documentation. Use *import "math/cmplx"* to load *complex128* math functions. These include the standard trig, log, and exponential functions plus functions for managing complex numbers. In invoking these functions, only the prefix *cmplx* is needed.

4.6.3 Random numbers

Here is the documentation. Import *"math/rand"* for random number generators. Use the prefix *rand*.

5 Other packages

5.1 Candis

Candis is method for storing gridded numerical data in structured form. This section describes the Go interface to Candis. Here is the manual page for the Go interface, while here is the original paper (PDF) on this system. Most of the Candis package is written in the C language, though there is also an interface to Python. We explore how to use the Candis interface via two example programs, one to write a Candis file, the other to read Candis.

The *gocandis* package should be imported into Go programs that use the Candis interface. Here we expect this package to be in the form of an archived library file in */usr/local/lib*. The *gocandis* package is part of the overall Candis package and is installed as a part of installing Candis. The

gocandis library is assumed to be installed in */usr/local/lib* under the name *gocandis* for what follows.

Here is the Candis writer:

```
// create a simple candis file
package main

import (
"gocandis"
)

func main() {

// array size specification
nx := 6
ny := 4

// initialize the data
x := make([]float64,nx)
xsq := make([]float64,nx)
for ix := 0; ix < nx; ix++ {
x[ix] = float64(ix)*3
xsq[ix] = x[ix]*x[ix]
}
y := make([]float64,ny)
for iy := 0; iy < ny; iy++ {
y[iy] = float64(iy)*5 + 10
}
a := make([]float64,nx*ny)
for ix := 0; ix < nx; ix++ {
for iy := 0; iy < ny; iy++ {
a[iy + ny*ix] = x[ix]*y[iy]
}
}

// generate a candis structure with initial data
c := gocandis.NullCandis()
c.AddComment("Simple candis file")
```

```
c.AddParam("Answer","42")
c.AddDim("x",x)
c.AddDim("y",y)
c.AddVField("xsq",[]string{"x"},xsq)
c.AddVField("a",[]string{"x","y"},a)
ap1 := []float64{43}
c.AddVField("ap1",[]string{},ap1)

// write candis file
c.PutCandis("simple.cdf")

// write some additional slices
for i := 0; i < 2; i++ {

// update variable field data
ap1[0] = ap1[0] + 1
for ix := 0; ix < nx; ix++ {
xsq[ix] = xsq[ix]*2
}

// transfer the updated data to the candis structure
c.UpdateVField("ap1",ap1)
c.UpdateVField("xsq",xsq)

// write the variable slice
c.PutNextVSlice()
}

// close out
c.CloseCandis()
}
```

The first section of this program defines fields of zero, one, and two dimensions and their associated dimensional information to be stored in a Candis file. Multi-dimensional fields are actually stored in flat form in a one-dimensional slice within Candis. Functions exist within Candis (*Unflatten** and *Flatten**) to convert between unflattened and flattened forms of fields – see the *gocandis 3* manual page. Also see the manual page for information

on properly indexing multi-dimensional fields while in flat form. Note that a zero-dimensional field is not stored as a scalar, but as a one-dimensional slice with a single element.

After all fields are created, a Candis structure is created that incorporates the field data as well as information about the fields. This information takes the form of a section on comments, another on parameters, mostly numerical but stored as text, and information about the dimensions and characteristics of the fields themselves. The dimension information is stored in what is called the *static slice*, whereas field data are stored in one or more *variable slices*.

In the above program, three variable slices are created with different data values. In many cases a single variable slice is sufficient. All variable slices have the same collection of fields, but with different contents. The most common use of variable slices is to split up data from successive times into different slices. This is useful if the size of the data set is very large; it allows the data set to be processed one piece at a time.

After the header information and initial variable slice data are written to the named file by *PutCandis*, additional variable slices are created and written. Finally the output file is closed. Note that the file is written to the standard output if the specified file name is *"-"*.

The following Go program reads the Candis file created by the above program and prints out various pieces of information in the file.

```
// read.go -- read the candis file created by create.go
package main

import (
"fmt"
"gocandis"
)

func main() {

// open the candis file
d := gocandis.GetCandis("simple.cdf")

// get comments
comments := d.GetComments()
fmt.Println("Comments:", comments)
```

```
// get parameter names
pnames := d.GetParamNames()
fmt.Println("Parameter names:",pnames)
pvalue := d.GetParamValue("Answer")
fmt.Println("Answer =",pvalue)

// get dimension names
dnames := d.GetDimNames()
fmt.Println("Dimension names:",dnames)

// get variable field names
vfnames := d.GetVFieldNames()
fmt.Println("Variable field names:",vfnames)

// get dimension information
x := d.GetDimData("x")
nx := len(x)
y := d.GetDimData("y")
ny := len(y)
fmt.Println("First dimension:",nx,x)
fmt.Println("Second dimension:",ny,y)

// get field information
ap1dims := d.GetFieldDims("ap1")
ap1 := d.GetFieldData("ap1")
xsqdims := d.GetFieldDims("xsq")
xsq := d.GetFieldData("xsq")
adims := d.GetFieldDims("a")
a := d.GetFieldData("a")
fmt.Println("xsq",xsqdims,"a",adims,"ap1",ap1dims)

// infinite loop over variable slices
for {
fmt.Println("ap1",ap1)
fmt.Println("xsq",xsq)
for i := 0; i < nx*ny; i++ {
ix := i/ny
```

```
iy := i - ix*ny
fmt.Println("Field indices and value:",i,ix,iy,a[i])
}

// get next variable slice and break out if EOF
ssize := d.GetNextVSlice()
if ssize == 0 {
break
}
}

// close the input file
d.CloseCandis()
}
```

This program is largely self-explanatory. As with creating a Candis file, a file name of *"-"* is read from the standard input. The loop is set up to process the initial variable slice and then check for subsequent variable slices. If only a single variable slice is expected, the loop can be omitted.

The above programs may be cut and pasted into files named *create.go* and *cread.go* respectively. The following is a Makefile to compile the two Go programs using *gccgo*:

```
all : create.go cread.go
gccgo -o create create.go -I /usr/local/lib -l gocandis
gccgo -o cread cread.go -I /usr/local/lib -l gocandis

clean :
rm -f create cread simple.cdf *~
```

To compile the two programs, type make. To create the Candis file *simple.cdf*, simply type *create*. To read it, type *cread*. The Candis file may be examined with the usual array of Candis tools.

5.2 Gomatrix

Gomatrix is a package by John Asmuth and Ryanne Dolan for manipulating real (float64) matrices. (Sorry, no complex matrices!) Import *matrix*.

It is written totally in Go and is not as fast as the highly optimized LAPACK/BLAS packages for C. Nevertheless, it is useful for all but the most highly demanding applications. The source code may be found here and the documentation may be found here. A version of the *gomatrix* package is now included in Candis. The default installation location is */usr/local/lib*. Access the library with the *-lmatrix* flag during compilation of your program with *gccgo*.

Gomatrix matrices are stored as structures containing the matrix elements in a flat slice plus the number of rows and columns of the matrix plus a variable called *step*. In this slice, column number iterates most rapidly. This is consistent with a two-dimensional Candis field if the first dimension is the row and the second dimension is the column.

The variable *step* is the number of elements one must advance in the slice to get from one row to the next. Normally it is equal to the number of columns. However, if a submatrix is defined in-place, the elements are represented as a subslice of the original matrix's element slice. This subslice contains the full rows of the original matrix encompassing the submatrix, and therefore may contain elements of the original matrix that are not part of the submatrix. In this case *step* in the submatrix must equal its value in the original matrix in order for the elements of the submatrix to be accessed correctly. Since the subslice is just a pointer to the orig, the extra elements don't represent wasted storage. However, a side effect is that if the elements change in the submatrix, the original matrix is also affected and vice versa.

Here is an example that shows how to solve a set of linear equations:

```
// sample linear equation solver
package main

import (
"fmt"
"matrix"
)

func main() {

// make a matrix
as := make([]float64,9)
as[0] = 1
```

```
as[1] = 3
as[2] = 2
as[3] = 4
as[4] = 2
as[5] = 7
as[6] = -1
as[7] = 3
as[8] = -1
a := matrix.MakeDenseMatrix(as,3,3)

// find the determinant of the matrix
fmt.Println("Det(a)", a.Det())

// make a vector
bs := make([]float64,3)
bs[0] = 4
bs[1] = 7
bs[2] = 9
b := matrix.MakeDenseMatrix(bs,3,1)

// invert the matrix (ignore the error for now)
ainv, _ := a.Inverse()

//compute the solution and check it
x := matrix.Product(ainv,b)
c := matrix.Product(a,x)
d := matrix.Product(a,ainv)

// show the results
fmt.Println("----------------------------")
fmt.Println(a,"= a")
fmt.Println("----------------------------")
fmt.Println(b,"= b")
fmt.Println("----------------------------")
fmt.Println(ainv,"= ainv")
fmt.Println("----------------------------")
fmt.Println(d,"= a*ainv")
fmt.Println("----------------------------")
```

```
fmt.Println(x,"= x (solution)")
fmt.Println("----------------------------")
fmt.Println(c,"= c (a*x -- should equal b)")
fmt.Println("----------------------------")

// convert answer to an array
xs := x.Array()
fmt.Println("xs:",xs)
}
```

This solves the problem $a * x = b$ where a is the matrix of coefficents of the set of linear equations, b is a specified column vector and x is the column vector containing the solution. The *float64* slices *as* and *bs* are converted into matrices with the specified number of rows and columns. The matrix a is inverted and multiplied by b to get the answer x. The output of this program is:

```
Det(a) -3.9999999999999982
----------------------------
{ 1,  3,  2,
  4,  2,  7,
 -1,  3, -1} = a
----------------------------
{4,
 7,
 9} = b
----------------------------
{ 5.75, -2.25, -4.25,
  0.75, -0.25, -0.25,
  -3.5,   1.5,   2.5} = ainv
----------------------------
{1, 0, 0,
 0, 1, 0,
 0, 0, 1} = a*ainv
----------------------------
{-31,
  -1,
  19} = x (solution)
----------------------------
```

```
{4,
 7,
 9} = c (a*x -- should equal b)
---------------------------
xs: [-30.999999999999993 -0.9999999999999989 18.999999999999996]
```

As expected, $a * ainv$ equals the identity matrix and $a * x = b$.

Here is a program that finds the eigenvalues and eigenvectors of a symmetric matrix:

```
// sample linear equation solver
package main

import (
"fmt"
"matrix"
)

func main() {

// make a symmetric matrix
as := make([]float64,9)
as[0] = 1
as[1] = 3
as[2] = 2
as[3] = 3
as[4] = 4
as[5] = 6
as[6] = 2
as[7] = 6
as[8] = -1
a := matrix.MakeDenseMatrix(as,3,3)

// find the determinant of the matrix
fmt.Println("Det(a)", a.Det())

// compute the eigenvalues and eigenvectors
eigenvecs, eigenvals, err := a.Eigen()
if err != nil {
```

```
panic(err)
}

// compute the transpose of the eigenvectors and check orthonormality
eigenvect := eigenvecs.Transpose()
delta := matrix.Product(eigenvecs,eigenvect)

// rotate the original matrix to see if we get it in the PA frame
x := matrix.Product(a,eigenvecs)
arot := matrix.Product(eigenvect,x)

// show the results
fmt.Println("---------------------------------")
fmt.Println(a,"= a")
fmt.Println("---------------------------------")
fmt.Println(eigenvecs,"= eigenvecs")
fmt.Println("---------------------------------")
fmt.Println(eigenvals,"= eigenvals")
fmt.Println("---------------------------------")
fmt.Println(delta,"= delta")
fmt.Println("---------------------------------")
fmt.Println(arot,"= arot")
fmt.Println("---------------------------------")
}
```

Here is the output of this program:

```
Det(a) 25
---------------------------------
{ 1,  3,  2,
  3,  4,  6,
  2,  6, -1} = a
---------------------------------
{        0, -0.921012,  0.389534,
   -0.5547,  0.324112,  0.766328,
   0.83205,  0.216075,  0.510886} = eigenvecs
---------------------------------
{       -5,         0,         0,
         0, -0.524938,         0,
```

```
         0,         0,  9.524938} = eigenvals
---------------------------------
{ 1,  0, -0,
  0,  1,  0,
 -0,  0,  1} = delta
---------------------------------
{       -5,        -0,         0,
        -0, -0.524938,         0,
         0,         0,  9.524938} = arot
---------------------------------
```

The eigenvalues are represented as the matrix a in principal axis form, with the eigenvalues along the diagonal. The eigenvectors are presented as columns of a square matrix called *eigenvecs* in the above program. This matrix is also the orthogonal transformation matrix that rotates vectors and tensors from the principal axis reference frame back to the original frame. The transpose of this matrix, *eigenvect*, does the reverse, as demonstrated by the matrix *arot*. As expected, the product of these two matrices, *delta*, is the identity matrix.

Here is a list of useful matrix functions and methods:

```
// Here is a selection of useful functions and methods in gomatrix.
// Dense matrices are those represented by a slice encompassing all
// elements of the matrix.  Sparse matrices are represented by a list
// of non-zero elements.  Only dense matrices are documented here.
// See the link to documentation cited above to see all functions and
// methods.

// Dense Matrix functions ----------------------------------------

// Create a matrix from a flat slice of elements
func MakeDenseMatrix(elements []float64, rows, cols int) *DenseMatrix

// Create a matrix with all elements zeroed
func Zeros(rows, cols int) *DenseMatrix

// Create an identity matrix with the number of rows and columns
// equal to span
func Eye(span int) *DenseMatrix
```

```
// Multiply a matrix A by a constant f, returning a new matrix.
// leaving the original matrix unchanged
func Scaled(A Matrix, f float64) (B *DenseMatrix)

// Return the sum of two or more matrices, leaving the originals
// unchanged
func Sum(A, B, C, ... Matrix) (C *DenseMatrix)

// Return the matrix product of two or more matrices, leaving the
// originals unchanged
func Product(A, B, C, ... Matrix) (C *DenseMatrix)

// Dense Matrix methods ----------------------------------------

// Getting information from a matrix --------------------------

// Return the number of rows and columns of matrix A
func (A *DenseMatrix) GetSize() (rows, cols int)

// Return the element at row i and column j of matrix A
func (A *DenseMatrix) Get(i int, j int) (v float64)

// Return a copy of the matrix A
func (A *DenseMatrix) Copy() *DenseMatrix

// Return a flat slice containing the elements of matrix A
func (A *DenseMatrix) Array() []float64

// Return row i of matrix A as a matrix row vector -- this
// is a slice, not a copy, so changing elements in one
// changes elements in the other
func (A *DenseMatrix) GetRowVector(i int) *DenseMatrix

// Return column j of matrix A as a matrix column vector -- this
// is a slice, not a copy, so changing elements in one
// changes elements in the other
func (A *DenseMatrix) GetColVector(j int) *DenseMatrix
```

```
// Return a submatrix of matrix A starting at row i and column
// j of A with the specified number of rows and columns -- this
// is a slice, not a copy, so changing elements in one
// changes elements in the other
func (A *DenseMatrix) GetMatrix(i, j, rows, cols int) *DenseMatrix

// Return a copy of the diagonal of matrix A as a slice
func (A *DenseMatrix) DiagonalCopy() []float64

// Return the trace of a matrix
func (A *DenseMatrix) Trace() float64

// Modifying a matrix ----------------------------------------

// Set the value of the element of matrix A at row i, column j
func (A *DenseMatrix) Set(i int, j int, v float64)

// Fill row i of matrix A with the contents of buf
func (A *DenseMatrix) FillRow(i int, buf []float64)

// Fill column j of matrix A with the contents of buf
func (A *DenseMatrix) FillCol(j int, buf []float64)

// Fill the diagonal of matrix A with the contents of buf
func (A *DenseMatrix) FillDiagonal(buf []float64)

// Multiply matrix A by a scalar f, leaving the result in A
func (A *DenseMatrix) Scale(f float64)

// Add matrix B to matrix A, leaving the result in A -- the error
// condition is returned
func (A *DenseMatrix) AddDense(B *DenseMatrix) error

// Subtract matrix B from matrix A, leaving the result in A -- the
// error condition is returned
func (A *DenseMatrix) SubtractDense(B *DenseMatrix) error
```

```
// matrix operations -------------------------------------------

// Return the transpose of matrix A
func (A *DenseMatrix) Transpose() *DenseMatrix

// Return the inverse Ainv of matrix A and an error code
func (A *DenseMatrix) Inverse() (Ainv *DenseMatrix, err error)

// Return the eigenvectors V and eigenvalues D of matrix A as well as
// and error code -- the columns of V contain the eigenvectors --
// D is in the form of a matrix with the eigenvalues on the diagonal
func (A *DenseMatrix) Eigen() (V, D *DenseMatrix, err error)

// Return the solution x to a set of linear equations A*x = b --
// an error code is also returned
func (A *DenseMatrix) Solve(b Matrix) (x *DenseMatrix, err error)

// Return L in the Cholesky decomposition of a positive definite
// matrix A = L*transpose(L) -- L is a lower triangular matrix -- an
// error code is also returned
func (A *DenseMatrix) Cholesky() (L *DenseMatrix, err error)

// Perform a QR decomposition of matrix A = Q*R where Q and R are
// returned -- Q is an orthogonal matrix and R is upper triangular
func (A *DenseMatrix) QR() (Q, R *DenseMatrix)

// Return the lower triangular part L of matrix A
func (A *DenseMatrix) L() *DenseMatrix

// Return the upper triangular part U of matrix A
func (A *DenseMatrix) U() *DenseMatrix

// Perform an LU decomposition of matrix A = P*L*U where L is lower
// triangular, U is upper triangular, and P is the pivot matrix,
// returning L, U, and P
func (A *DenseMatrix) LU() (L, U *DenseMatrix, P *PivotMatrix)

// Perform a singular value decomposition of matrix A =
```

```
// U*Sigma*transpose(V), returning U, Sigma, and V -- U and V are
// orthogonal matrices and Sigma is diagonal -- an error code is also
// returned
func (A *DenseMatrix) SVD() (U, Sigma, V *DenseMatrix, err error)
```

5.3 Fast Fourier transform

I have extracted the fast Fourier transform (FFT) sub-package (merging with the *dsputils* sub-package on which it depends) from the go-dsp signal processing package of Matt Jibson. The FFT package is documented here and the *dsputils* package is documented here. The combined FFT/dsputils package has been incorporated into candis. The library is called *fft* and it normally lives in */usr/local/lib*.

The discrete Fourier transform is defined

$$X_k = \sum_{n=0}^{N-1} x_n \exp(-2\pi ikn/N)$$

and the corresponding inverse transform is defined

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \exp(2\pi ikn/N).$$

Note that the FFT has an odd notion of index ordering. Speaking in terms of space and wavenumber, the forward transform, x_n starts at the zero x at $n = 0$, reaches the most positive value of x at $n = N/2 - 1$, jumps back to the most negative value at $n = N/2$, and reaches the negative value just before zero at $n = N - 1$. The same thing happens with wavenumber.

A sample one-dimensional forward and inverse transform is illustrated in the following program:

```
// make forward and inverse transformations in 1-D
package main

import (
"fmt"
"math"
"fft"
)
```

```
func main() {

// construct a real function
nx := 16
x := make([]float64,nx)
for m := range x {
xval := float64(m - nx/2)
x[reorder(m, nx)] = math.Exp(-0.2*xval*xval)
}

// do the forward transform
y := fft.FFTReal(x)

// do the inverse transform
xp := fft.IFFT(y)

// print results
fmt.Println("  n   m  np        x[n]       x[m]           y[m]                x
for n := range x {
m := reorder(n, nx)
np := reorder(m, nx)
fmt.Printf("%3d %3d %3d: %10.6f %10.6f %10.6f %10.6f\n",n,m,np,
x[n],x[m],y[m],real(xp[n]))
}
}

func reorder(n, N int) (m int) {
ncenter := N/2
if n >= ncenter {
m = n - ncenter
} else {
m = n + ncenter
}
return
}
```

This program takes the FFT of a centered Gaussian curve, resulting in a

centered Gaussian in wavenumber space as well. The Gaussian function produced by the initial loop is reordered by the function *reorder* to accommodate the FFT routine. The inverse of the results are also computed. The results are then printed in columns by the program.

The results of this program are shown below:

```
 n   m  np        x[n]       x[m]          y[m]                 xp[n]
 0   8   0:   1.000000   0.000003 (  0.000032 +0.000000i)   1.000000
 1   9   1:   0.818731   0.000055 (  0.000317 -0.000000i)   0.818731
 2  10   2:   0.449329   0.000747 (  0.003837 -0.000000i)   0.449329
 3  11   3:   0.165299   0.006738 (  0.032001 -0.000000i)   0.165299
 4  12   4:   0.040762   0.040762 (  0.181376 +0.000000i)   0.040762
 5  13   5:   0.006738   0.165299 (  0.699210 -0.000000i)   0.006738
 6  14   6:   0.000747   0.449329 (  1.833120 +0.000000i)   0.000747
 7  15   7:   0.000055   0.818731 (  3.268461 +0.000000i)   0.000055
 8   0   8:   0.000003   1.000000 (  3.963324 +0.000000i)   0.000003
 9   1   9:   0.000055   0.818731 (  3.268461 +0.000000i)   0.000055
10   2  10:   0.000747   0.449329 (  1.833120 +0.000000i)   0.000747
11   3  11:   0.006738   0.165299 (  0.699210 +0.000000i)   0.006738
12   4  12:   0.040762   0.040762 (  0.181376 +0.000000i)   0.040762
13   5  13:   0.165299   0.006738 (  0.032001 -0.000000i)   0.165299
14   6  14:   0.449329   0.000747 (  0.003837 -0.000000i)   0.449329
15   7  15:   0.818731   0.000055 (  0.000317 -0.000000i)   0.818731
```

The column n shows the index of slices input and output by the forward and inverse transforms. The column m shows the permuted index while np shows that two applications of the permutation yield the original index. The column $x[n]$ shows the input slice in unpermuted form, whereas $x[m]$ shows it in the permuted, human-readable form. $y[m]$ shows the Fourier transform of x in permuted form and $xp[n]$ shows the results of the inverse transform in unpermuted form.

The FFT package uses the Cooley-Tukey algorithm for N equal to a power of two and the Bluestein algorithm otherwise. The latter is slower than the former, but still takes of order $N \log N$ calculations. This package also performs multi-dimensional transforms by doing one-dimensional transforms successively on the different dimensions. Two-dimensional transforms are singled out as a special case, as they are used so frequently. This FFT package includes the "$1/N$" factor in the inverse transform, unlike some packages.

Forward transforms of real (*float64*) fields are also included as a special case of the more general complex (*complex128*) transforms.

Here is a selection of useful functions in the *fft* package:

```
// Some useful functions in gofft.

// Make a one-dimensional complex to complex forward transform
func FFT(x []complex128) []complex128

// Make a one-dimensional real to complex forward transform
func FFTReal(x []float64) []complex128

// Make a one-dimensional complex to complex inverse transform
// Note that a real to complex inverse transform exists, but a
// complex to real inverse transform does not -- the former is
// not very useful, so is not documented here
func IFFT(x []complex128) []complex128

// Make a two-dimensional complex to complex forward transform --
// note that the input and results are represented by unflattened
// two-dimensional slices -- the gocandis package has routines
// to flatten and unflatten 2, 3, and 4 dimensional slices
func FFT2(x [][]complex128) [][]complex128

// Make a two-dimensional real to complex forward transform
func FFT2Real(x [][]float64) [][]complex128

// Make a two-dimensional complex to complex inverse transform
func IFFT2(x [][]complex128) [][]complex128

// Make convolution of 2 one-dimensional complex slices --
// this uses FFT and IFFT
func Convolve(x, y []complex128) []complex128
```

Similar functions exist for higher dimensional transforms. See the online documentation.

5.4 Sio, a simple input/output package

As it stands, the input/output packages *io* and *bufio* are perhaps overly complex. *Sio* is a simple package based on the input/output routines in the base level *os* package. Documentation for the package, follows:

- File handling:
  - func Create(filename string) (fileptr *os.File, err error)
    - Create a new file "filename" for writing. The file pointer "fileptr" is returned along with the error flag "err". The "os" package file handling routines are used to accomplish this. In the special case in which the file name is "stdout" or "stderr", no new file pointer is returned. Instead, the pre-defined standard output or standard error file pointers are returned.
  - func Open(filename string) (fileptr *os.File, err error)
    - As in Create except an existing file is opened for reading. If the file name is "stdin", the pre-defined standard input pointer is returned.
  - func Append(filename string) (fileptr *os.File, err error)
    - As in Append except an existing file is opened for appending. If the file doesn't exist, a new file with the specified name is created for writing. The filename should not point to a pipe, e.g., "stdout" or "stderr".
  - func Close(fileptr *os.File) (err error)
    - This routine closes the file represented by the file pointer "fileptr".
- Binary reading and writing:
  - func Get(fileptr *os.File, nbytes int) (inbuff []byte, err error)
    - This routine reads up to "nbytes" bytes from the previously opened file associated with "fileptr". Less than "nbytes" are read if an end of file is encountered. The results are returned in a byte slice whose length equals that of the number of bytes read. An error is raised on an end of file.
  - func Put(fileptr *os.File, b []byte) (err error)

        * This routine writes the bytes in byte slice "b" to the open file represented by "fileptr". An error is returned if something goes wrong.

- Text reading and writing:
    - func GetLine(fileptr *os.File) (s string, err error)
        * This routine returns a line of text from the previously opened file. The read is terminated when it encounters a newline character. This is the normal line separator for Linux. If a carriage return is encountered (as occurs before the newline in Windows text files), it is silently discarded. The string containing the text does not include the newline. If an end of file is encountered before a newline, an error is returned along with the text received up to that point.
    - func PutLine(fileptr *os.File, s string) (err error)
        * This routine writes a line of text represented by string "s" (without a terminating newline) to the previously opened file. An error is returned if a problem is encountered.

A program exercising all of the *sio* functions except *append* is shown below, illustrating how the package functions are used.

```
// Sample program exercising sio package main
import (
"sio"
)
func main() {

// open needed files -- possible error
// conditions are ignored here, but should
// be checked for high reliability programs
stdin, _ := sio.Open("stdin")
stdout, _ := sio.Create("stdout")
ofbin, _ := sio.Create("outf.bin")

// read each line of standard input -- err
// indicates an end of file
```

```
for {
s, err := sio.GetLine(stdin)
if err != nil {
break
}
// write each line to standard output
sio.PutLine(stdout,s)

// also convert each line to a string and
// do a binary write to a file
sio.Put(ofbin,[]byte(s))
}
sio.Close(ofbin)

// read the binary file -- the lesser of the file
// length and maxbytes will go into t -- repeated
// Gets could be used to read a bigger file or
// maxbytes could be increased
maxbytes := 10000
ifbin, _ := sio.Open("outf.bin")
t, _ := sio.Get(ifbin, maxbytes)
sio.Close(ifbin)

// reconvert the catted binary lines to a single
// string and write to stdout
sio.PutLine(stdout,string(t))
}
```

5.5 Gompi, a gccgo wrapper for C language MPI calls

MPI (Message Passing Interface) is the standard mechanism for implementing parallel, especially massively parallel programming. There are a few wrappers of the C language MPI functions for the go compiler, but there appear to be none for the gccgo compiler. The Candis distribution (see above) now contains a minimal but functional set of wrappers for MPI. A man page is included in the Candis distribution.

A working example of use of these wrappers to solve the 2-D heat equation in parallel is presented here. (Note that the algorithm used is simple

relaxation, so the code is not particularly efficient. However, it illustrates a possible way to write such code.)

Here is the program:

```
// This is a heat conduction model for testing speed across different
// languages -- static array version using MPI parallelism via the
// gompi package.

package main

import (
  "fmt"
  "gocandis"
  "math"
  "gompi"
)

// these constants have to be specified in order to allow gccgo to
// do bounds checking in compile time rather than run time -- this
// speeds things up by a factor of order 2
const (
  nx = 102
  ny = 802
  nt = 20000
  dx = 1.0
  dy = 1.0
)

type fields struct {
  chi [nx*ny]float64
  chistar [nx*ny]float64
}

func main() {

  // start MPI
  comm, nprocs, rank := gompi.Init()
  fmt.Println("comm, nprocs, rank:",comm,nprocs,rank)
```

```
// initialize global fields in all processes
pi := 3.14159
mx := (nx - 2)*nprocs + 2
my := ny
lambdax := dx*float64(mx - 1)
lambday := dy*float64(ny - 1)
x := make([]float64,mx)
y := make([]float64,my)
for ix := 0; ix < mx; ix++ {
  x[ix] = dx*float64(ix)
}
for iy := 0; iy < my; iy++ {
  y[iy] = dy*float64(iy)
}
chi := make([]float64,mx*my)
for ix := 0; ix < mx; ix++ {
  for iy := 0; iy < my; iy++ {
    phasex := 6*pi*x[ix]/lambdax
    phasey := pi*y[iy]/lambday
    i := I(my,ix,iy)
    chi[i] = math.Sin(phasex)*math.Sin(phasey)
  }
}

// initialize local fields for each MPI process -- the
// global fields are chopped up to make the local fields
// with overlap to handle inter-process boundaries
v := fields{}
ix1 := (nx - 2)*rank
ix2 := ix1 + nx
for ix := ix1; ix < ix2; ix++ {
  for iy := 0; iy < my; iy++ {
    v.chi[I(ny,ix - ix1,iy)] = chi[I(my,ix,iy)]
    v.chistar[I(ny,ix - ix1,iy)] = chi[I(my,ix,iy)]
  }
}
```

```
  // do evolution
  timestep(nx,ny,nt,rank,nprocs,comm,&v)

  // gather chi fields from worker processes (including root) and merge
  chirecv := make([]float64,nx*ny)
  for xrank := 0; xrank < nprocs; xrank++ {
    copy_float64(v.chi[:],chirecv,rank,xrank,0,comm)
    if rank == 0 {
      ix1 = (nx - 2)*xrank
      ix2 = ix1 + nx
      for ix := ix1; ix < ix2; ix++ {
        for iy := 0; iy < my; iy++ {
          chi[I(my,ix,iy)] = chirecv[I(ny,ix - ix1,iy)]
        }
      }
    }
  }

  // send chi to a candis file if root process
  if rank == 0 {
    c := gocandis.NullCandis()
    c.AddComment("Goheat result")
    c.AddDim("x",x)
    c.AddDim("y",y)
    c.AddVField("chi",[]string{"x","y"},chi)
    c.PutCandis("gotest.cdf")
    c.CloseCandis()
  }

  // finish MPI
  gompi.Finalize()
}

func timestep(nx, ny, nt, rank, nprocs int,
  comm gompi.MPI_Comm, v *fields) {

  // time loop
  for it := 0; it < nt; it++ {
```

```
    // space loops -- do the relaxation
    for ix := 1; ix < nx - 1; ix++ {
      for iy := 1; iy < ny - 1; iy++ {
        v.chistar[I(ny,ix,iy)] =
          0.25*(v.chi[I(ny,ix + 1,iy)] +
          v.chi[I(ny,ix - 1,iy)] +
          v.chi[I(ny,ix,iy + 1)] +
          v.chi[I(ny,ix,iy - 1)])
      }
    }

    // set up MPI transfer of x boundary columns in
    // parallel with writeback -- use non-blocking send
    // and receive
    sl, sr, rr, rl := btransfer(rank,nprocs,comm,v)

    // writeback while boundary conditions are being transferred
    for ix := 1; ix < nx - 1; ix++ {
      for iy := 1; iy < ny - 1; iy++ {
        i := I(ny,ix,iy)
        v.chi[i] = v.chistar[i]
      }
    }

    // wait for boundary transfers to finish
    bwait(sl,sr,rr,rl)
  }
}

// copy the contents of a buffer for irank to a buffer for orank
// works if irank == orank since non-blocking send/recv functions used
// however the function itself blocks -- this is used for gathering
// the results at the end of the computation
func copy_float64(ibuff, obuff []float64, rank, irank, orank int,
comm gompi.MPI_Comm) {
  var sresult, rresult *gompi.MPI_Request
  if rank == irank {
```

```
    sresult = gompi.Isend_float64(ibuff,orank,comm)
  }
  if rank == orank {
    rresult = gompi.Irecv_float64(obuff,irank,comm)
  }
  if rank == irank {
    gompi.Wait(sresult)
  }
  if rank == orank {
    gompi.Wait(rresult)
  }
}

// do non-blocking transfers between neighboring local blocks to
// satisfy boundary conditions between blocks
func btransfer(rank, nprocs int, comm gompi.MPI_Comm,
  v *fields) (sl, sr, rr, rl *gompi.MPI_Request) {
  lrank := rank - 1
  if lrank < 0 {lrank += nprocs}
  rrank := rank + 1
  if rrank >= nprocs {rrank -= nprocs}
  sl = gompi.Isend_float64(v.chistar[I(ny,1,0):I(ny,2,0)],
    lrank,comm)
  sr = gompi.Isend_float64(v.chistar[I(ny,nx - 2,0):I(ny,nx - 1,0)],
    rrank,comm)
  rr = gompi.Irecv_float64(v.chi[I(ny,nx - 1,0):I(ny,nx,0)],
    rrank,comm)
  rl = gompi.Irecv_float64(v.chi[I(ny,0,0):I(ny,1,0)],
    lrank,comm)
  return
}

// wait for the above non-blocking transfers to complete
func bwait(sl, sr, rr, rl *gompi.MPI_Request) {
  gompi.Wait(sl)
  gompi.Wait(sr)
  gompi.Wait(rr)
  gompi.Wait(rl)
```

```
}

// flatten 2-D array references to a one-dimensional array value
func I(ny, ix, iy int) (ival int) {
  ival = ix*ny + iy
  return
}
```

As is typical of MPI, this program is rather long compared to equivalent serial code. However, it is mostly self-explanatory, given the comments. The local arrays for each process are statically declared. This helps gccgo do most of the bounds checking needed in the time loop in the compile phase. The resulting code is as fast (within 10%) as the equivalent code in C using the gcc compiler.