G Go interface

Go! Go! Go!
— Punters at the race track.

Compared to other programming languages with interfaces to the `mathcw` library, the Go language is relatively recent. It was not even available when most of the software library, and the book *The Mathematical Function Computation Handbook*, were written.

This appendix would have appeared in the book at the location suggested by its page numbers, but was necessarily written after book publication.

G.1 Go history

Go language development began in 2007 at Google, Inc., but the first public release from that team was in 2012. In December 2010, the GNU Compiler Collection (`gcc`) team added support for Go with the `gccgo-4.6` compiler. The language is well described in a recent book, *The Go Programming Language*, by Alan A. A. Donovan and Brian W. Kernighan.[1] There are also extensive online resources for Go, notably, in *The Go Programming Language Specification*[2] and via the `go doc name` command.

As that book describes, the heritage of Go can be traced to several older programming languages, including C. Unlike C, Go is strongly typed, supports low-cost microthreading through `go` keywords prefixed to function calls, has function calls prefixed with the `defer` keyword that run between function returns and resumption of execution in the caller, allows multiple return values from functions, offers automatic memory management with run-time garbage collection, lacks a macro preprocessor, has no comma operator for evaluation synchronization, and has no equivalent of the `volatile` type qualifier of C.

Unlike most other programming languages, Go always initializes all program variables: zero for numerics, and empty for lists and strings. Explicit initializers can be given if needed.

The `go` and `gccgo` compilers have been ported to most current platforms, from mobile devices to supercomputers, and the language designers guarantee upward compatibility. Go programs can therefore be written to run

[1]Addison-Wesley (2016), xvii + 380 pages, ISBN-10 0-13-419044-0, ISBN-13 978-0-13-419044-0, LCCN QA76.73.G63 D66 2016.
[2]Available at `https://golang.org/ref/spec`, along with many other resources at the site.

everywhere without source code changes, and with the expectation that the code will remain acceptable, and with the same meaning, to future versions of the language and compilers.

G.2 Go language summary

Each Go source file declares a package name to which the following source code belongs, but only functions and variables that begin with an uppercase letter are exported, allowing them to be visible in code in other packages.

Comments in Go are like those in C99, and do not nest: `//` begins a comment that extends to the end of the current line, and `/* ... */` brackets a possibly multiline comment.

Statements in Go are normally on separate lines, and do not require terminating semicolons, as C does. Semicolons are needed in Go only when several short statements appear on a single source line.

Signed and unsigned integer types in Go match current hardware, and are like those in many other modern programming languages, including C: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64`. Go also has *platform-dependent* generic integer types `int` and `uint`: they correspond to 32-bit integers on machines with a 32-bit address space, and to 64-bit integers when the address size is 64 bits.

Go has a Boolean type, `bool`, with values `true` and `false`. Variables of that type occupy one byte of storage, even though only one bit is needed. Unlike C, `bool` values in Go cannot be coerced to integer values by typecasts.

For floating-point arithmetic, Go provides only two types: `float32` and `float64`. Its mathematical function support is largely modeled on the `double` functions of the C library.

Type declarations in Go are reversed from the practice in C. In Go, they consist of the `var` keyword, followed by a comma-separated list of variables, followed by a data type, as in these examples:

```
var x, y, z float64         /* Go-style scalars */
var u [10]int               // Go-style vector
var v [25]int               // Go-style vector
```

Equivalent declarations in C look like this:

```
double x, y, z;             /* C-style scalars */
int u[10], v[25];           /* C-style vectors */
```

In Go, strings are always stored in the UTF-8 encoding of the Unicode character set. Each Unicode code point, which humans would call a *character* or a *glyph*, is called a *rune* in Go, and requires from one to four bytes. Because the ASCII character set occupies the first 128 positions in Unicode, *all* existing computer files in ASCII are already UTF-8 encoded. Unlike C, strings in Go are not terminated with a `NUL` character (`'\0'`); any Unicode code point, including `NUL`, is valid in a Go string, which is an immutable object with an address and a length.

The UTF-8 encoding algorithm guarantees that no multibyte code point contains a `NUL` byte, so unless the Go programmer has intentionally inserted such a byte into a string, premature termination of strings passed to C code cannot happen.

Go permits any Unicode code point classed as a letter or digit to be used in identifier names, although such practices are expected to be uncommon in software developed in parts of the world that require only the traditional unaccented Latin letters in their language alphabets.

Arrays and strings in Go are indexed from 0, and substrings can be selected with a following `[i:j]` selector, which means elements or runes from positions `i` to `j - 1`. Either, or both, of the limits may be omitted in taking a substring of a string `s`, in which case `0` and `len(s)`, respectively, are assumed. Thus, `s[3:7]`, `s[3:]`, `s[:4]`, and `s[:]` are all valid, even when `s` is replaced by a quoted string constant. It is a fatal compile-time, or run-time, error to index a string or array outside its bounds.

The string concatenation operator in Go is the plus sign, and it can be used for both constant and variable strings. Thus, `"ab" + "cd"` is a new string with the value `"abcd"`.

Here is a traditional example of what a simple Go program looks like:

```
% cat hello.go
// This is a simple program that prints a greeting.
package main
```

```
import "fmt"

/* This is a comment */
func main() {
        fmt.Println("Hello, World ... this is the Go language")
}
```

It can be compiled and run on UNIX-like systems with either of these commands:

```
% go run hello
Hello, World ... this is the Go language

% gccgo -g hello.go && ./a.out
Hello, World ... this is the Go language
```

The `package` statement is mandatory, and its name must be `main` for any file that contains a top-level main program.

The `import` statement is analogous to the preprocessor directive `#include` in C. Header files in C are almost always text files, but imported packages in Go are compiled binary files. The `import` statement may specify a single quoted string, or else a parenthesized list of strings separated by newlines or semicolons. However, unlike C, Go requires that each imported package be *used* in the source code. It does so to simplify dependency checking for the `go` build system, which in simple cases can replace the UNIX `make` tool.

The `import` statement can have an alternative form where an alias is prefixed to a package name:

```
import f "fmt"
import m "math"
...
f.Printf("m.Sqrt(2) = %.16g\n", m.Sqrt(2))
```

However such aliasing is rarely needed, or desirable, because standard Go package names tend to be short.

The position of the open brace in a compound statement is significant in Go: it *must* be on the same line as the statement to which it belongs. In practice, that means that matching braces do not align vertically, making careful code formatting and indenting advisable. The command `go fmt` can be used to prettyprint source code files into a standard form.

Go has a powerful string formatting package that enhances the offerings of C's `printf()` family, and the function call `fmt.Println()` illustrates external package identification, and the capitalization of exported names.

Unlike C, Go main programs do not return an explicit value to the operating system: the return value available to a subsequent shell command is normally zero. You can return a nonzero value with a final statement `os.Exit(49)` provided that `os` appears in an `import` statement. Curiously, with the `go run` command, that value is reported in an output line `exit status 49` and a return code of 1. With the `gccgo` compiler, no status line is printed, and the return code is 49. However, this alternative approach of first building, then running, the program makes the behavior consistent:

```
% go build hello-with-exit

% ./hello-with-exit ; echo $status
Hello, World ... this is the Go language
49
```

G.3 Numbers in Go

In C, C++, C#, Fortran, Java, and many other languages, the size or precision of numeric constants is specified by a suffix on the constant. Thus, in C, we might define symbolic names like this:

```
const float      PI_32    = 3.14159265F;
const double     PI_64    = 3.1415926535897932;
const __float80  PI_80    = 3.14159265358979323846W;
const __float128 PI_128   = 3.14159265358979323846264338327950288Q;

const uint8_t  UINT8_MAX  = 255U;
const uint16_t UINT16_MAX = 65536U;
const uint32_t UINT32_MAX = 4294967295UL;
const uint64_t UINT64_MAX = 18446744073709551615ULL;
```

Go lacks numeric type suffixes, and infers precision from the constants themselves, except that integer constants are treated as type `int` unless cast to a different size.

Here is a sample program and its output that illustrates the differences:

```
% cat numbers.go
package main

import  "fmt"

func main() {
        PI_32     := 3.141_592_65
        PI_64     := 3.141_592_653_589_793_2
        PI_XX     := 3.141_592_653_589_793_238_462_643_383_279_502_88

        ui8_max   := uint8(255)
        ui16_max  := uint16(65_535)
        ui32_max  := uint32(4_294_967_295)
        ui64_max  := uint64(18_446_744_073_709_551_615)

        fmt.Printf("ui8_max    is %20d (%#x)\n", ui8_max,  ui8_max)
        fmt.Printf("ui16_max   is %20d (%#x)\n", ui16_max, ui16_max)
        fmt.Printf("ui32_max   is %20d (%#x)\n", ui32_max, ui32_max)
        fmt.Printf("ui64_max   is %20d (%#x)\n", ui64_max, ui64_max)

        fmt.Printf("PI_32  is %.8g\n", PI_32)
        fmt.Printf("PI_64  is %.16g\n", PI_64)
        fmt.Printf("PI_XX  is %.25g\n", PI_XX)

        fmt.Printf("PI_32  is %.g (no length field)\n", PI_32)
        fmt.Printf("PI_64  is %.g (no length field)\n", PI_64)
        fmt.Printf("PI_XX  is %.g (no length field)\n", PI_XX)
}

% go run numbers
ui8_max    is                  255 (0xff)
ui16_max   is                65535 (0xffff)
ui32_max   is           4294967295 (0xffffffff)
ui64_max   is 18446744073709551615 (0xffffffffffffffff)
PI_32  is 3.1415927
PI_64  is 3.141592653589793
PI_XX  is 3.141592653589793115997963
PI_32  is 3 (no length field)
PI_64  is 3 (no length field)
PI_XX  is 3 (no length field)
```

Notice that although `PI_XX` is assigned a long numeric constant, its precision is limited by its default 64-bit type, `float64`, and the extra digits printed do not match the original constant.

Go permits digit-separating underscores in numbers, although its output formatting provides no options to produce them. Unlike many languages, Go allows the most negative 32- and 64-bit signed integer values to be written in source code: `-2_147_483_648` and `-9_223_372_036_854_775_808`.

The Go `:=` operator used in our test program is a short declaration that declares a variable with the type of the expression on the right-hand side, and initializes it with the value of that expression. It is illegal to use that operator more than once for the same variable in the current scope in the function.

For the integer variables, if strong typing were not required, then we could have omitted the type casts, except for that for `ui64_max`, because the value is too big for a 64-bit signed integer. The cast for `ui32_max` would still be needed, however, on a system where `int` is a 32-bit size.

Alternatively, we could replace the short declarations with code like this:

```
const PI_32    = 3.141_592_74
const PI_64    = 3.141_592_653_589_7931
const PI_XX    = 3.141_592_653_589_793_238_462_643_383_279_502_884_197_169_399_375_105
const ui8_max  = uint8(255)
const ui16_max = uint16(65_535)
const ui32_max = uint32(4_294_967_295)
const ui64_max = uint64(18_446_744_073_709_551_615)
```

That is better practice, because it prevents reassignment with different values, and might offer better opportunities for compiler optimizations.

The Go Programming Language Specification says this about the evaluation of constant numeric expressions:

> ***Constant expressions are always evaluated exactly; intermediate values and the constants themselves may require precision significantly larger than supported by any predeclared type in the language.***

G.4 Simple interface from Go to C

Go has a pseudo-package named `C` that is handled specially by the `go` compilers from both Google and from `gcc` builds; at the time of writing this, it is not supported at all by the `gccgo` compiler. Here is a short example of a small test program and its execution, followed by commentary:

```
% cat go-to-c-1.go
// Package go-to-c-1 demonstrates how to call the C library sqrt()
// function from Go.
package main

import (
        "fmt"
        "math"
)

// #cgo LDFLAGS: -lm
// #include <math.h>
// double alt_sqrt(double x) { return (sqrt(x)); }
import "C"

func main() {
        fmt.Printf("Go says that     sqrt(2) is %.16g\n", math.Sqrt(2))
        fmt.Printf("C  says that     sqrt(2) is %.16g\n", C.sqrt(C.double(2)))
        fmt.Printf("C  says that alt_sqrt(2) is %.16g\n", C.alt_sqrt(C.double(2)))

        fmt.Printf("C  says that     sqrt(2) is %.16g\n", C.sqrt(2))
}

% go run go-to-c-1
Go says that     sqrt(2) is 1.414213562373095
C  says that     sqrt(2) is 1.414213562373095
C  says that alt_sqrt(2) is 1.414213562373095
C  says that     sqrt(2) is 1.414213562373095
```

We need the `math` package to access the Go math library repertoire, and its square-root function must be invoked as `math.Sqrt()`.

The comment block that precedes the `import "C"` statement contains C code in either comment style, with *no* blank lines between the comment and the `import` keyword. That C code must provide prototypes for any functions needed from that language, most easily done via standard header files. However, for this short example, we could have replaced the `#include` directive with the prototype `double sqrt(double);`.

When additional libraries, or compiler options, are needed for the C functions, they must be specified in the special comments following the `#cgo` directive as `variable : value(s)` settings, most commonly with `CFLAGS` for compiler flags, and `LDFLAGS` for loader/linker flags. The `go` compiler imposes strong restrictions on acceptable values for such variables, but they do not affect us here.

In Go code, each reference to a C constant, variable, or function must be identified with a language prefix and for values, a C type qualifier, unless the function prototypes allow the argument types to be inferred by the `go` compiler. Thus, we wrote the last output statement using the shorter form, `C.sqrt(2)`.

G.5 The Go interface to the mathcw library

The preceding section showed that the simple case of passing scalar arguments from Go to C is reasonably easy to accomplish. However, many of the functions in the mathcw library have character string, pointer, and array arguments, and we have not yet shown how they are handled.

Because Go is strongly typed, the size of an array is part of its type: you cannot pass a 4-element vector to a function that expects a vector of a different fixed size. When the sizes match, a *copy* is passed to the function, so any changes to the array that are made inside the function are lost on return to the caller.

Arrays are therefore avoided in most Go programs, and replaced by *slices*. A *slice* is a special structure that contains a pointer to the contiguous array data, plus a current length, and a maximum capacity. Slices are first-class objects that can be assigned values, passed as arguments, returned from functions, and dynamically resized.

Here is a test program and its output to illustrate some of the features of slices:

```
% cat slice.go
package main

import  "fmt"

func main() {
        v := []float64 { 3, 14, 159, 2653, 5897, 93238, 462643, 3832795 }
        w := make([]float64, 3, 15)
        x := w

        fmt.Printf("v      = %v\n", v)
        fmt.Printf("len(v) = %v\n", len(v))
        fmt.Printf("cap(v) = %v\n\n", cap(v))

        fmt.Printf("w      = %v\n", w)
        fmt.Printf("len(w) = %v\n", len(w))
        fmt.Printf("cap(w) = %v\n\n", cap(w))

        fmt.Printf("x      = %v\n", x)
        fmt.Printf("len(x) = %v\n", len(x))
        fmt.Printf("cap(x) = %v\n\n", cap(x))

        w = v

        fmt.Println("After slice assignment, w = v, we have:\n")

        fmt.Printf("w      = %v\n", w)
        fmt.Printf("len(w) = %v\n", len(w))
        fmt.Printf("cap(w) = %v\n\n", cap(w))

        fmt.Println("Grow an array by appending values:\n")

        x = append(x, 100)
        x = append(x, 200)
        x = append(x, 300)

        fmt.Printf("x      = %v (after 3 appends)\n", x)
        fmt.Printf("len(x) = %v\n", len(x))
        fmt.Printf("cap(x) = %v\n\n", cap(x))

        fmt.Println("Do more appends\n")

        for i := 0; i < 15; i++ { x = append(x, float64(i)) }

        fmt.Printf("x      = %v (after 15 more appends)\n", x)
        fmt.Printf("len(x) = %v\n", len(x))
        fmt.Printf("cap(x) = %v\n\n", cap(x))
}
```

```
% go run slice
    v      = [3 14 159 2653 5897 93238 462643 3.832795e+06]
    len(v) = 8
    cap(v) = 8

    w      = [0 0 0]
    len(w) = 3
    cap(w) = 15

    x      = [0 0 0]
    len(x) = 3
    cap(x) = 15

    After slice assignment, w = v, we have:

    w      = [3 14 159 2653 5897 93238 462643 3.832795e+06]
    len(w) = 8
    cap(w) = 8
```

```
        Grow an array by appending values:

        x      = [0 0 0 100 200 300] (after 3 appends)
        len(x) = 6
        cap(x) = 15

        Do more appends:

        x      = [0 0 0 100 200 300 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14] (after 15 more appends)
        len(x) = 21
        cap(x) = 30
```

Notice that no loop over elements is required to assign one slice to another, and slice assignment creates a *new* slice, rather than overwriting a previous value. Even when there is capacity to do so, you cannot assign a slice element with an index greater than or equal to the length. However, the `append()` function can grow the slice, and the last set of operations shows that the slice is automatically reallocated with a larger capacity as needed.

We printed entire slices with the `%v` format item, a powerful feature of the `fmt` package. An advanced feature of the Go language is the concept of *reflection*: it allows writing of generic code that can determine types and layout of function arguments, and produce suitably tailored output. Such code is needed to support `%v` formatting.

Slices are what we need to pass vector data between Go and C functions. Here is a wrapper function for a member of the `mathcw` library that does that:

```
import "unsafe"
...
// void fsplit(double result[/* 2 */], double x);
import "C"
...
func Fsplit(v []float64, x float64) {
        C.fsplit((*C.double)(unsafe.Pointer(&v[0])), C.double(x))
}
```

The first argument to the Go function `Fsplit()` is a slice of type `float64`, a composite structure. The C function `fsplit()` expects a pointer to a vector of `double` elements (only two are needed here), and that pointer is computed from the address of the first element, `&v[0]`, coerced by `unsafe.Pointer()` from a pointer to `float64` to a pointer to an arbitrary memory location, then further coerced by the typecast `(*C.double)` to the type expected by the C function. It is important to note that *no numeric data conversion is required*: the two languages just have different names for the same storage format and layout. The double casting is needed to subvert the strong typing in Go. The package name `unsafe` is intentionally descriptive: it warns that the protection offered by strong typing has been overridden, and things might go wrong. The parentheses around the call to `unsafe.Pointer()` are mandatory.

Every running Go program has a garbage collector thread that recovers allocated memory that can no longer be reached. The collector can also do storage compaction that *moves* stack variables. That could change the address of the data for `v[]`, invalidating references made to it inside the C function. That issue caused considerable discussion on Go developer lists in 2015, but in later compiler releases, it appears that the problem is resolved by wrapping the call to the C function in another function that temporarily suspends garbage collection until the C function returns.

The last interlanguage issue that we must deal with is the passing of UTF-8 character strings from Go to C where the expectation is that each byte holds exactly one character, and that the string is NUL-terminated. Here, an explicit data conversion is required, and it is done as in this wrapper function for another member of the `mathcw` library:

```
// double (nan)(const char *);
import "C"
```

```
...
func Nan (s string) float64 {
    cs := C.CString(s)
    defer C.free(unsafe.Pointer(cs))
    return (float64(C.nan(cs)))
}
```

The short declaration of `cs` dynamically creates a new string suitable for use in C with a copy of the data in the Go string `s`, and that new string is in the C memory space, the heap, managed by `malloc()` and `free()`. That puts it safely out of reach of the Go garbage collector. However, we only need it for the duration of the call to `C.nan()`, after which it can be freed. We neatly avoid creation of a temporary variable to hold the function return value by using the `defer` keyword on the call to `C.free()`.

Tests of passing Go strings with Unicode characters to C show that the non-ASCII runes appear as bytes with values in 128...255 (a consequence of the UTF-8 encoding design), and any embedded `NUL` runes are preserved in the C string, effectively terminating it in normal string operations. The string received by the C function has the same UTF-8 byte sequence as in the Go caller.

G.6 Examples of the Go `MathCW` package

In this section, we show how the `MathCW` package is used in Go programs. Because the library location is embedded in the compiled interface, nothing additional needs to be specified by the user: the usual `go build` and `go run` commands just need the name(s) of the user's Go file(s).

Most of our examples use functions that are available in both the native Go `math` package, and in the `MathCW` package, so that computed results from the Go and C languages can be compared.

G.6.1 Floating-point access to fraction and exponent

The first test program shows results in both decimal and binary to allow verification that the operations provided by `Frexp()` and `Ldexp()` are *exact*: the exponents change, but the significand bits *must not*.

The Go math library offers similar functions to those in the C library, although the calling sequences sometimes differ, as they do in this example, where `Frexp()` receives only a single argument, but returns a pair of values that correspond to the fraction and exponent.

```
% cat frexp.go
// Package frexp demonstrates frexp() and ldexp() from the native
// Go math library, as well as from the MathCW library.
package main

import "fmt"
import "math"
import "MathCW"

func main() {
    var e32             int32
    var nexp            int
    var f32, x32        float32
    var f64, frac, x64  float64

    fmt.Println("Demonstration of 32-bit frexp() and ldexp()\n")

    x32 = -0x1.fadecap-32

    fmt.Printf("                        x32 = %#x\n", x32)
    frac, nexp = math.Frexp(float64(x32))
    f32 = MathCW.Frexpf(x32, &e32)

    fmt.Printf("  math.Frexp (float64(x32)) = (%#x, %d)\n", frac, nexp)
    fmt.Printf("MathCW.Frexpf(x32, &e32)    = (%#x, %d)\n", f32, e32)

    fmt.Println()

    fmt.Printf("  math.Ldexp (%#x, %d) = %#x\n", frac, nexp, math.Ldexp (frac, nexp))
    fmt.Printf("MathCW.Ldexpf(%#x, %d) = %#x\n", f32, e32, MathCW.Ldexpf(f32, e32))

    fmt.Println()

    fmt.Println("Demonstration of 64-bit frexp() and ldexp()\n")

    x64 = -0x1.beadcafefeedep-64

    fmt.Printf("                      x64 = %#x\n", x64)

    frac, nexp = math.Frexp(x64)
    f64 = MathCW.Frexp(x64, &e32)

    fmt.Printf("  math.Frexp(x64)         = (%#x, %d)\n", frac, nexp)
    fmt.Printf("MathCW.Frexp(x64, &e32)   = (%#x, %d)\n", f64, e32)

    fmt.Println()

    fmt.Printf("  math.Ldexp(%#x, %d) = %#x\n", frac, nexp,   math.Ldexp(frac, nexp))
    fmt.Printf("MathCW.Ldexp(%#x, %d) = %#x\n", f64,  e32,  MathCW.Ldexp(f64, e32))
}

% go run frexp
Demonstration of 32-bit frexp() and ldexp()

                        x32 = -0x1.fadecap-32
  math.Frexp (float64(x32)) = (-0x1.fadecap-01, -31)
MathCW.Frexpf(x32, &e32)    = (-0x1.fadecap-01, -31)

  math.Ldexp (-0x1.fadecap-01, -31) = -0x1.fadecap-32
MathCW.Ldexpf(-0x1.fadecap-01, -31) = -0x1.fadecap-32

Demonstration of 64-bit frexp() and ldexp()

                      x64 = -0x1.beadcafefeedep-64
  math.Frexp(x64)         = (-0x1.beadcafefeedep-01, -63)
MathCW.Frexp(x64, &e32)   = (-0x1.beadcafefeedep-01, -63)

  math.Ldexp(-0x1.beadcafefeedep-01, -63) = -0x1.beadcafefeedep-64
MathCW.Ldexp(-0x1.beadcafefeedep-01, -63) = -0x1.beadcafefeedep-64
```

G.6.2 Computing trigonometric functions

Our second example computes the sine and cosine of a number with both mathcw and Go libraries:

```
% cat sincos.go
    package main

    import "fmt"
    import "math"
    import "MathCW"

    func main() {
        var c, s, x float64

        fmt.Println("Demonstration of 64-bit sincos()\n")

        x = 3.0 * (math.Pi / 8.0)
        fmt.Printf("                       x = %g\n", x)
        fmt.Printf("                         = %#x\n", x)
        fmt.Printf("                         = (3/8) * Pi\n\n")

        fmt.Println("Expect sin((3/8) * Pi)   = 0.923_879_532_511_286_756_128_...")
        fmt.Println("                         = 0x1.d906_bcf3_28d4_628a_fcc2_0463_...p-1\n")

        fmt.Println("Expect cos((3/8) * Pi)   = 0.382_683_432_365_089_771_728_...")
        fmt.Println("                         = 0x1.87de_2a6a_ea96_2d1a_6245_854b_...p-2\n")

        s, c = math.Sincos(x)
        fmt.Printf("  math.sincos(x)         = (%.17f, %.17f)\n", s, c)
        fmt.Printf("                         = (%#x, %#x)\n\n",   s, c)

        MathCW.Sincos(x, &s, &c)
        fmt.Printf("MathCW.Sincos(x, ss, cc) = (%.17f, %.17f)\n", s, c)
        fmt.Printf("                         = (%#x, %#x)\n",     s, c)
    }

    % go run sincos
    Demonstration of 64-bit sincos()

                           x = 1.1780972450961724
                             = 0x1.2d97c7f3321d2p+00
                             = (3/8) * Pi

    Expect sin((3/8) * Pi)   = 0.923_879_532_511_286_756_128_...
                             = 0x1.d906_bcf3_28d4_628a_fcc2_0463_...p-1

    Expect cos((3/8) * Pi)   = 0.382_683_432_365_089_771_728_...
                             = 0x1.87de_2a6a_ea96_2d1a_6245_854b_...p-2

      math.sincos(x)         = (0.92387953251128674, 0.38268343236508984)
                             = (0x1.d906bcf328d46p-01, 0x1.87de2a6aea964p-02)

    MathCW.Sincos(x, s, c)   = (0.92387953251128674, 0.38268343236508984)
                             = (0x1.d906bcf328d46p-01, 0x1.87de2a6aea964p-02)
```

G.6.3 Decoding floating-point fields

Our third example does not interface to the mathcw library at all, but is nevertheless useful in working with floating-point numbers.

The C library provides functions for decomposing and reconstructing floating-point values from their sign, exponent, and significand fields, with `signbit()`, `frexp()`, and `ldexp()`. Go has similar functions `math.Signbit()`, `math.Frexp()`, and `math.Ldexp()`, although the argument conventions for the first two differ from their companions in C.

Sometimes, however, it is useful for debugging and analysis purposes to be able to access the three fields and display them in various number bases. The package fmtfloat included with the mathcw software provides a set of string-returning functions to do that. A companion `testfmtfloat.go` file exercises those functions with important special values in 32-bit and 64-bit floating-point arithmetic.

We can use the `go doc` command to display the brief package documentation, along with prototypes for all of its public constants and functions:

```
% go doc fmtfloat
package fmtfloat // import "fmtfloat"

Package fmtfloat provides functions for decoding and formatting fields of
floating-point numbers. They each return a dynamic string that can be used
for printing such numbers in a form that displays their encoding in storage.

const Bias32 int64 = 0x7f
const Bias64 int64 = 0x3ff
func Bitfields32(x float32) string
func Bitfields64(x float64) string
func Decfields32(x float32) string
func Decfields64(x float64) string
func Hexfields32(x float32) string
func Hexfields64(x float64) string
func Octfields32(x float32) string
func Octfields64(x float64) string
```

The two-sentence paragraph was extracted from leading commands immediately preceding the `package` statement, but the remainder of the output was automatically produced by `go doc` from the source code, without any need for the Go programmer to duplicate that information.

We can use `go doc` to extract documentation of a particular function in a package:

```
% go doc fmtfloat Hexfields32
package fmtfloat // import "fmtfloat"

func Hexfields32(x float32) string
    Hexfields32 decomposes a float32 value into its three fields in hexadecimal.
```

We can dig even further and ask for the function source code as well:

```
% go doc -src fmtfloat Hexfields32
package fmtfloat // import "fmtfloat"

// Hexfields32 decomposes a float32 value into its three fields in hexadecimal.
    func Hexfields32(x float32) string {
            var h int64
            b := fmt.Sprintf("%.032b", math.Float32bits(x))
            e, _ := strconv.ParseInt(b[1:9], 2, 32)
            s, _ := strconv.ParseInt(b[9:], 2, 32)
```

```
                if e == 0 {
                        h = 0
                } else {
                        h = 1
                }
                return (fmt.Sprintf("sign : %s  exponent : %#02x (2-to-(%+04d))  \
    significand : 0x%d.%06xp0",
                        b[0:1], e, e-Bias32+1-h, h, s))
        }
```

The displayed code differs somewhat from that in the file `fmtfloat.go`: it has been automatically prettyprinted into standard Go code layout, with one long line wrapped at a backslash-newline to fit the page.

You can use the same commands to extract documentation and source code for *any* standard Go package: the system fetches needed data from the Web if it cannot be found locally. Here is a short example:

```
% go doc math Signbit
package math // import "math"

func Signbit(x float64) bool
    Signbit reports whether x is negative or negative zero.
```

The Go documentation system provides a simple solution to a common problem in all programming languages and software projects, and it is well worthwhile for Go programmers to make use of. We showed only text output here, but the system can also produce attractive HTML pages for display in Web browsers.

G.6.4 Accurate vector summation

Our fourth, and last, example uses the mathcw function `vsum()` to compute a sum of vector elements, with an error estimate.

We choose the vector carefully so that its sum cannot be represented exactly in a 64-bit floating-point value, but is easily computed by hand. Recall that the sum of the first n integers is $(n(n+1))/2$, so we can predict that the computed sum with $n = 10$ should be close to $(10 \times 11)/2 = 55$, plus a minor perturbation from the small elements, each with only a single nonzero bit.

```
% cat vsum.go
// Package vsum demonstrates vector summation with
// error accumulation, using the MathCW library.
package main

import "fmt"
import "fmtfloat"
import "math"
import "MathCW"

func main() {
    var err, sum float64
    v := []float64 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                    0x1p-45, 0x1p-46, 0x1p-47, 0x0p-48,
                    0x1p-49, 0x1p-50, 0x1p-51, 0x1p-52,
                    0x1p-53, 0x1p-54, 0x1p-55, 0x1p-56,
                    0x1p-57, 0x1p-58, 0x1p-59, 0x1p-60 }

    sum = MathCW.Vsum(&err, int32(len(v)), v)
```

```
        fmt.Printf("len(v)     = %d\n", len(v))
        fmt.Printf("v          = %#.3v\n", v)
        fmt.Printf("           = %#x\n", v)
        fmt.Printf("sum        = %g  = %#x\n", sum, sum)
        fmt.Printf("err        = %g  = %#x\n", err, err)

        fmt.Println()

        fmt.Printf("sum bits   = %#.064b\n", math.Float64bits(sum))
        fmt.Printf("err bits   = %#.064b\n", math.Float64bits(err))

        fmt.Println()

        fmt.Printf("sum bit fields = %s\n", fmtfloat.Bitfields64(sum))
        fmt.Printf("err bit fields = %s\n", fmtfloat.Bitfields64(err))

        fmt.Println()

        fmt.Printf("sum dec fields = %s\n", fmtfloat.Decfields64(sum))
        fmt.Printf("err dec fields = %s\n", fmtfloat.Decfields64(err))

        fmt.Println()

        fmt.Printf("sum oct fields = %s\n", fmtfloat.Octfields64(sum))
        fmt.Printf("err oct fields = %s\n", fmtfloat.Octfields64(err))

        fmt.Println()

        fmt.Printf("sum hex fields = %s\n", fmtfloat.Hexfields64(sum))
        fmt.Printf("err hex fields = %s\n", fmtfloat.Hexfields64(err))
    }

    % go run vsum
    len(v)     = 27
    v          = []float64{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2.84e-14, 1.42e-14,        \
                            7.11e-15, 0, 1.78e-15, 8.88e-16, 4.44e-16, 2.22e-16,         \
                            1.11e-16, 5.55e-17, 2.78e-17, 1.39e-17, 6.94e-18,            \
                            3.47e-18, 1.73e-18, 8.67e-19}
               = [0x0.000p+00 0x1.000p+00 0x1.000p+01 0x1.800p+01 0x1.000p+02           \
                  0x1.400p+02 0x1.800p+02 0x1.c00p+02 0x1.000p+03 0x1.200p+03           \
                  0x1.400p+03 0x1.000p-45 0x1.000p-46 0x1.000p-47 0x0.000p+00           \
                  0x1.000p-49 0x1.000p-50 0x1.000p-51 0x1.000p-52 0x1.000p-53           \
                  0x1.000p-54 0x1.000p-55 0x1.000p-56 0x1.000p-57 0x1.000p-58           \
                  0x1.000p-59 0x1.000p-60]
    sum        = 55.00000000000005  = 0x1.b800000000007p+05
    err        = 3.5518463170625125e-15  = 0x1.ffep-49

    sum bits   = 0b0100000001001011100000000000000000000000000000000000000000000111
    err bits   = 0b0011110011101111111111100000000000000000000000000000000000000000

    sum bit fields = sign : 0  exponent : 0b10000000100 (2-to-(+0005))                  \
            significand : 0b1.1011100000000000000000000000000000000000000000000111p0
    err bit fields = sign : 0  exponent : 0b01111001110 (2-to-(-0049))                  \
            significand : 0b1.11111111111000000000000000000000000000000000000000p0

    sum dec fields = sign : 0  exponent : 1028 (2-to-(+0005))                            \
            significand : 1.7187500000000016
    err dec fields = sign : 0  exponent : 0974 (2-to-(-0049))                            \
            significand : 1.99951171875

    sum oct fields = sign : 0  exponent : 0o02004 (2-to-(+0005))                         \
            significand : 0o1.5600000000000000034p0
    err oct fields = sign : 0  exponent : 0o01716 (2-to-(-0049))                         \
            significand : 0o1.7776000000000000000p0

    sum hex fields = sign : 0  exponent : 0x404 (2-to-(+0005))                           \
            significand : 0x1.b800000000007p0
    err hex fields = sign : 0  exponent : 0x3ce (2-to-(-0049))                           \
            significand : 0x1.ffe0000000000p0
```

G.7 Calling Go functions from C

The Go build system permits preparation of source files that can be compiled into a shared library usable by C programs. As a short example, we show a Go file that makes three math library functions available for use in a C program, prefixing names with Go_ to avoid collisions with normal C library functions, permitting both to be used in the same program, and following the go doc command conventions:

```
% cat gomath32.go
// Package gomath32 supplies an interface from Go to C for some
// mathematical functions.  Building the package creates both a shared
// library, libgomath32.so, and a C/C++ header file, libgomath32.h.
//
// Usage:
//    go build -buildmode c-shared -o libgomath32.so gomath32.go
package main

import "C"
import "math"

// Interface from Go to C for expf().
//
//export Go_expf
func Go_expf(x float32) float32 { return (float32(math.Exp(float64(x)))) }

// Interface from Go to C for logf().
//
//export Go_logf
func Go_logf(x float32) float32 { return (float32(math.Log(float64(x)))) }

// Interface from Go to C for tanf().
//
//export Go_tanf
func Go_tanf(x float32) float32 { return (float32(math.Tan(float64(x)))) }

func main() { }
```

The leading comments show how the shared library is created. The `import "C"` statement is required, even though there are no visible references to that package. The `//export` comments that immediately precede function definitions make their names visible in the shared library.

Because the Go math library offers functions only in the longer floating-point type, our interface wrappers have to convert library function results to the shorter type for the wrapper return value.

The C test program for those functions looks like this:

```
% cat mathtest.c
/*
** Usage:
**    cc mathtest.c -L. -Wl,-rpath,. -lgomath32 -lm && ./a.out
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PRINTF  (void)printf

extern float Go_expf(float);
extern float Go_logf(float);
extern float Go_tanf(float);

int
main(void)
{
    float x, y;

    x = 0.5F;

    y = expf(x);
    PRINTF("C  expf(%.3f) = %12.9g = %.6a\n", x, y, y);

    y = Go_expf(x);
    PRINTF("Go expf(%.3f) = %12.9g = %.6a\n", x, y, y);

    PRINTF("\n");

    y = logf(x);
    PRINTF("C  logf(%.3f) = %12.9g = %.6a\n", x, y, y);

    y = Go_logf(x);
    PRINTF("Go logf(%.3f) = %12.9g = %.6a\n", x, y, y);

    PRINTF("\n");

    y = tanf(x);
    PRINTF("C  tanf(%.3f) = %12.9g = %.6a\n", x, y, y);

    y = Go_tanf(x);
    PRINTF("Go tanf(%.3f) = %12.9g = %.6a\n", x, y, y);

    return (EXIT_SUCCESS);
    }
```

Because the program is short, we supplied our own prototypes for the Go functions. We could instead have used the generated header file `libgomath32.h` that provides type definitions to map from Go to C or C++, and function prototypes for the complete interface.

The leading comments show how the two languages are combined into an executable program, and run, producing this output:

```
C  expf(0.500) =   1.64872122 = 0x1.a61298p+0
Go expf(0.500) =   1.64872122 = 0x1.a61298p+0

C  logf(0.500) = -0.693147182 = -0x1.62e430p-1
Go logf(0.500) = -0.693147182 = -0x1.62e430p-1

C  tanf(0.500) =  0.546302497 = 0x1.17b4f6p-1
Go tanf(0.500) =  0.546302497 = 0x1.17b4f6p-1
```

G.8 Limitations of Go and its math library

At the beginning of this appendix, we briefly mentioned some of the features of Go that should be of interest to programmers of traditional languages like C and Fortran.

There are many powerful features of Go that make programming convenient, faster, and less error prone, notably enhanced output formatting, dynamic strings and garbage collection, string concatenation, and array assignments. Once the syntactical differences were mastered, this author found coding in Go a pleasant experience, with fewer logic errors than would be likely in C.

However, there are glaring deficiencies of Go as well:

- The absence of 80-bit and 128-bit floating-point numbers is disappointing, given that one, or both, have been available in common hardware, and other programming languages, for more than four decades. Even if many users of floating-point arithmetic concentrate primarily on the 32-bit and 64-bit formats, it is often valuable for enhanced accuracy in internal routines to have access to wider formats.

- The Go `math` package addresses mainly 64-bit computation, yet there are several critical 32-bit functions that cannot be correctly computed by coercions between the two sizes.

- The Go library exposes only one kind of NaN, hiding the distinction between quiet and signaling NaNs, and giving no support for NaN payloads.

- There is no access to advanced features of IEEE 754 arithmetic, such as precision control, rounding mode selection, choice between abrupt versus gradual underflow, and exception handling.

- Although division by zero is perfectly well defined in IEEE 754 arithmetic, the Google and `gcc` Go compilers all refuse to compile code with such divisions. Instead, a minor rewrite is required:

  ```
  // Fails to compile: error: division by zero
  // func Inf64() float64 { return (1.0 / 0.0) }

  func Inf64() float64 { var x float64; return (1.0 / x) }
  ```

In testing Go code written in support of the interface to the `mathcw` library, the author found two missing features in GNU `gccgo` and `go` that are available in Google `go`: digit-separating underscores in numbers, and hexadecimal floating-point constants. While the former are strongly desirable for readability, the lack of the latter is a huge deficiency for accurate numerical programming. Once again, the programmer is faced with an inability to specify numbers in code and data that are correct to the last bit.

G.9 Summary

We have now covered all of the tools in Go that are needed to communicate with functions written in C in the `mathcw` library. That library was carefully designed to minimize the number of argument types needed for interlanguage communication. In particular, it does not use any data structures other than scalars and one-dimensional arrays. With additional effort, it is possible to pass `struct` values between Go and C, but we do not require that ability here.

We noted that the Go `math` library exploits multiple-value returns, such as for `math.Frexp()`, `math.Lgamma()`, `math.Modf()`, and `math.Sincos()`. Doing so avoids the need for pointer arguments, which often have some peril. However, we intentionally chose *not* to use such features in the `mathcw` interface, because it is essential that the library be usable from many programming languages, and that its UNIX manual page documentation can be used for all such languages, without having to document changes in calling sequences and return values.

The file `mathcw.go` available in the MathCW distribution is generated completely automatically by an awk program that reads the carefully formatted C header file, `mathcw.h`, so that the package author has confidence that the Go interface is free of human-caused errors. The file `mathcw.go` contains more than 550 Go functions that can access *all* functions for real arithmetic in the `mathcw` library that have data type counterparts in both languages. That library has more than 5100 functions, but all those without interface wrappers require C types that are unavailable in Go.