Crossplatform usage of compiled C library with golang

> *The project domain is solving linear systems of equations. Currently implemented only matrix method.*

Run demo on darwin arm64(requires installed docker, golang, gcc):

```
./test.sh;
```

The example output located in the end of report.

Library

The basic library is implemented in C in `linear_system.c` and binary compiled to platforms:

- `darwin-arm64`
- `linux-arm64`
- `linux-amd64`

Library binaries compiled to `./bin` directory in format `./bin/${platform}_liblinear_system.so` and created each time when `test.sh` script is running.

Compiler requires custom flags to build library, which will be used as a dynamically linked library later:

```
gcc -o ./bin/${platform}_liblinear_system.so -fpic -shared ./linear_system.c
```

Usage by golang

The library is dynamically linked in `main.go` file, using `cgo` directives. Cgo enables the creation of Go packages that call C code by importing a pseudo-package "C".

The Go code can then refer to types such as `C.size_t`, variables such as `C.stdout`, or functions such as `C.putchar`.

The following `cgo` directives(preamble) are used to connect library to package:

```
// #cgo CFLAGS: -I.
// #cgo LDFLAGS: -L. -llinear_system
// #include "linear_system.h"
// #include <stdlib.h>
// #include <stdio.h>
import "C"
```

> *Note: `cgo` can only create dynamically linked binaries of golang code, statically linked binaries are not supported.*

Calling C functions

As mentioned above, all C functions and structs are available in the package. Here is an example of calling C function to create vector and store pointer to it in golang variable:

C

```
Vector* make_Vector(number *a, int length) {
  Vector *vector = (Vector*)malloc(sizeof(Vector));
  vector->a = a;
  vector->length = length;
  return vector;
}
```

Golang

```
func MakeVector(a []float64) (*C.Vector, func()) {
    // ...
    cVec := C.make_Vector(&cArr[0], C.int(length))
}
```

Memory management and function calling.

Garbage collection

Fundamentally, golang is garbage collected language and doesn't require manual memory management, C - doesn't have garbage collection and requires to manually allocate and free memory. It means that memory management should be taken carefully at shim-code.

As a rule of thumb it's better never pass to `C` functions `go` 's reference types([]C.int, etc.), but instead do malloc-operations to create an arrays. Some functions in Cgo already make this:

```
// predefined function
func C.CString(string) *C.char
```

To clean up the memory, we call the `C.free` function. Example:

```
func MakeVector(a []float64) (*C.Vector, func()) {
  // ...
    cVec := C.make_Vector(&cArr[0], C.int(length))
    return cVec, func() {
        C.free(unsafe.Pointer(cVec))
    }
}
```

Pinning mechanism

Go is a garbage collected language, and the garbage collector needs to know the location of every pointer to Go memory. Because of this, there are restrictions on passing pointers between Go and C.

All Go pointers passed to C must point to pinned Go memory. Go pointers passed as function arguments to C functions have the memory they point to implicitly pinned for the duration of the call. Go memory reachable from these function arguments must be pinned as long as the C code has access to it.

Go values created by calling new, by taking the address of a composite literal, or by taking the address of a local variable may also have their memory pinned using `runtime.Pinner` . This type may be used to manage the duration of the memory's pinned status, potentially beyond the duration of a C function call.

Memory may be pinned more than once and must be unpinned exactly the same number of times it has been pinned.

Example:

```
func MakeVector(a []float64) (*C.Vector, func()) {
    length := len(a)
    cArr := make([]C.float, length)
    pinner.Pin(&cArr)
    for i, v := range a {
        cArr[i] = C.float(v)
    }
    cVec := C.make_Vector(&cArr[0], C.int(length))
    return cVec, func() {
        C.free(unsafe.Pointer(cVec))
        pinner.Unpin()
    }
}
```

So there are two approaches to pass an array to C code:

1. Use `runtime.Pinner` to pin the array and pass it to C code.
2. Use `malloc` to allocate an array and pass it to C code.

Iterating over C array

Golang doesn't have pointer arithmetics, so to make any iteration over C array, it's necessary to use `unsafe` package. Example:

```
/*
The problem is that pointer in C can be treated as pointer to struct and as an
array.
So the matrix have **Vector attribute, which should be treated as an array of
*Vector.
*/
func ParseMatrix(cMat *C.Matrix) [][]float64 {
    rows := int(cMat.rows)
    a := make([][]float64, rows)

    vectorArrayPtr := uintptr(unsafe.Pointer((*cMat).row_vectors))
    pointerSize := uintptr(unsafe.Sizeof((*cMat).row_vectors))
    for i := 0; i < rows; i++ {
        vector := (**C.Vector)(unsafe.Pointer(
            uintptr(vectorArrayPtr + pointerSize*uintptr(i)),
        ))
        a[i] = ParseVector(*vector)
    }
    return a
}
```

Test output

```
----PLATFORMS----
darwin_arm64: arm64-darwin
linux_arm64: arm64-linux
linux_amd64: amd64-linux


-----COMPILING LIBRARY-----
Building for arm64-darwin...
Building for arm64-linux using Docker...
Building for amd64-linux using Docker...
----COMPILATION SUCCESSFUL----

----EXECUTION FOR DARWIN ARM64----
Running main for darwin arm64...
Hello from golang, platform: darwin; arch: arm64
Hello from C, dude!
0.000000 1.000000 3.000000
1.000000 1.000000 1.000000
4.000000 2.000000 1.000000
9.000000 3.000000 1.000000
Matrix is not singular
[matrix_inverse] matrix:
1.000000 1.000000 1.000000
4.000000 2.000000 1.000000
9.000000 3.000000 1.000000
[matrix_inverse] adjugate:
-1.000000 2.000000 -1.000000
5.000000 -8.000000 3.000000
-6.000000 6.000000 -2.000000
[matrix_inverse] det: -2.000000
[matrix_inverse] result:
0.500000 -1.000000 0.500000
-2.500000 4.000000 -1.500000
3.000000 -3.000000 1.000000
SolveMatrix:  [0.5 -0.5 0]

----EXECUTION FOR LINUX ARM64----
Unable to find image 'golang:1.22' locally
1.22: Pulling from library/golang
Digest: sha256:147f428a24c6b80b8afbdaec7f245b9e7ac342601e3aeaffb321a103b7c6b3f4
Status: Downloaded newer image for golang:1.22
Hello from golang, platform: linux; arch: arm64
Hello from C, dude!
0.000000 1.000000 3.000000
1.000000 1.000000 1.000000
4.000000 2.000000 1.000000
9.000000 3.000000 1.000000
Matrix is not singular
```

```
[matrix_inverse] matrix:
1.000000 1.000000 1.000000
4.000000 2.000000 1.000000
9.000000 3.000000 1.000000
[matrix_inverse] adjugate:
-1.000000 2.000000 -1.000000
5.000000 -8.000000 3.000000
-6.000000 6.000000 -2.000000
[matrix_inverse] det: -2.000000
[matrix_inverse] result:
0.500000 -1.000000 0.500000
-2.500000 4.000000 -1.500000
3.000000 -3.000000 1.000000
SolveMatrix:  [0.5 -0.5 0]
Done

----EXECUTION FOR LINUX AMD64----
Unable to find image 'golang:1.22' locally
1.22: Pulling from library/golang
Digest: sha256:147f428a24c6b80b8afbdaec7f245b9e7ac342601e3aeaffb321a103b7c6b3f4
Status: Downloaded newer image for golang:1.22
Hello from golang, platform: linux; arch: amd64
Hello from C, dude!
0.000000 1.000000 3.000000
1.000000 1.000000 1.000000
4.000000 2.000000 1.000000
9.000000 3.000000 1.000000
Matrix is not singular
[matrix_inverse] matrix:
1.000000 1.000000 1.000000
4.000000 2.000000 1.000000
9.000000 3.000000 1.000000
[matrix_inverse] adjugate:
-1.000000 2.000000 -1.000000
5.000000 -8.000000 3.000000
-6.000000 6.000000 -2.000000
[matrix_inverse] det: -2.000000
[matrix_inverse] result:
0.500000 -1.000000 0.500000
-2.500000 4.000000 -1.500000
3.000000 -3.000000 1.000000
SolveMatrix:  [0.5 -0.5 0]
Done
```