Practical 02

By the end of this set of problems, you will be familiar with:

- Arrays
- Kinds
- `function`
- `subroutine`

All `programs` *must* contain `implicit none`. While not strictly necessary, it will eliminate an entire class of bugs.

It is a very good idea to always use the following compiler flags for all these problems: `-Wall -Wextra -fcheck=all -g`. Try to ensure you have no warnings. You may also want to use `-std=f2008` or `-std=f2018`, depending on the version of `gfortran` you are using. This will ensure you stick to standard Fortran.

You should use the lecture materials for help/inspiration, but please don't copy and paste! There is some value to be had in typing up the programs yourself.

There are very likely too many exercises here for the time you have available. It's ok if you don't get through them all!

There may be several ways to solve each problem. If you have time, you might like to try different approaches.

Use a separate file and program for each exercise.

Reversed Range

1. Write a program that creates an array filled with the numbers 1 to 10 and prints it to screen.
2. Read two integers in the range $[1, 10]$ from the user and print out the array within that range. Hint: you can use either a loop or a slice. Which is shorter?
3. Using the two numbers from step 2, reverse the numbers in the array within that range, and print the entire array. Hint: it may be useful to use a stride!

Sample output:

```
$ gfortran reversed_range.f90 -o reversed_range
$ ./reversed_range
   1   2   3   4   5   6   7   8   9  10
 Enter lower and upper bounds
2 6
   1   6   5   4   3   2   7   8   9  10
```

Further

1. What happens if the user gives numbers outside the range $[1, 10]$? What are some ways of handling this?
2. What happens if the user gives the two numbers in the opposite order than you're expecting? What are some ways of handling this?
3. Make the size of the array an input parameter. Hint: this requires the use of `allocatable`.

Factorial

The factorial of n is

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n.$$

1. Write a function that computes $n!$ by using a `do` loop
2. Write a `recursive` function that computes $n!$ by calling itself with $(n-1)$.

3. Write a function that can compute $n!$ on every element of an array using either of the above methods

Cross product

The cross product of two three dimensional vectors is given by:

$$a \times b = (a_2 \cdot b_3 - a_3 \cdot b_2)\hat{\mathbf{i}} + (a_3 \cdot b_1 - a_1 \cdot b_3)\hat{\mathbf{j}} + (a_1 \cdot b_2 - a_2 \cdot b_1)\hat{\mathbf{k}}$$

1. Write a subroutine that returns the cross product of two arrays via an out argument.
2. Rewrite your subroutine as a function that returns the cross product in the result.

Matrix-vector multiplication

The formula for matrix multiplication

$$\mathbf{A} = \mathbf{BC},$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices with elements a_{ij}, b_{ij}, c_{ij}, respectively, is

$$a_{ij} = \sum_{k=1}^{n} b_{ik} c_{jk}$$

1. Write a function that returns the result of multiplying a 3x3 matrix with a 3-element vector.
2. Write a `logical function` that returns `.true.` if the result of `matmul` is identical to your result if you used `integer`s, or sufficiently close (given some tolerance) if you used `real`s, and `.false.` otherwise.

You can check your implementations here by using the intrinsic `matmul`. Don't use it in your implementations though!

Further

1. Write a function that returns the result of multiplying an $n \times m$ matrix with a vector of length m.
2. Can you enforce or check that the size of the vector is compatible with the matrix?
3. Write a function that returns the result of multiplying an $n \times m$ matrix with an $m \times k$ matrix. What can you do if the two matrices are incompatible sizes?

Mean and standard deviation

The mean, $\bar{x}$, and standard deviation, σ, of a set x of n numbers are given by

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x(i)$$

and

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x(i) - \bar{x})^2}.$$

1. Write a subroutine that takes an array and returns its mean and standard deviation via out arguments.
2. Given a `real` scalar or array `r`, `call random_number(r)` will fill `r` with a random number between 0 and 1. You can use this to verify your subroutine is working correctly: the mean of a uniformly distributed set of numbers between $[0, 1)$ is 0.5, and the standard deviation is about 0.29.

Euler Integration

The Euler update formula for some function $f(y,t)$ is

$$y^{n+1} = y^n + (\Delta t) f(y^n, t^n),$$

where $y^n = y(n\Delta t)$ is an approximation to $y(t)$, Δt is the timestep, and t^n is the current time. Use this formula to solve the ODE

$$f(y,t) = \frac{dy}{dt} = \sin^2(t), y(0) = 0.$$

The exact solution at $t = \pi/2$ is $y = \pi/4$.

1. Write a program that has two functions: one that returns $f(t) = \sin^2(t)$, and one that returns y^{n+1} given y^n, Δt and t^n. The second function should call the first. Use the default `real` for floating point variables.
2. Read an integer N from the user, and then take N timesteps from 0 to $\pi/2$. Compute the error. How does it vary as you increase N?
3. Change the `real` variables to kind `real64`. Now what is the error as you increase N? Make the kind a parameter so that you can change between `real64` and `real32` and see the difference.

Calculating π

It is possible to use the `random_number` routine from above to calculate π. The method is as follows:

1. Generate a random point in a 2D surface between $[-1, 1]$. You will need to generate two numbers here, one for x and one for y.
2. Calculate the distance from the origin $(0, 0)$. (Hint: the intrinsic `hypot` may be useful here).
3. Repeat steps 1. and 2. for a total of N points.
4. Determine how many points are less than a distance of 1 from the origin. Using the ratio of the number of these points to the total N, along with $A = \pi r^2$, calculate π.
5. Repeat the above steps for varying N until the answer has converged in N.
6. How does this change when using `real32` compared to `real64`?