Wednesday 160/2D Class

Office hours this evening in 2082, from 6:30-8:30pm.
⇒ catch up on Session 3/4 or Assignment 2 (or 1)

New handouts: 313 from Landau/Pace, derivative_test.cpp,
  GSL_eigsystems, eigenv_test.cpp
  From before: handout on pointers, pointer_test.cpp

Gameplan:
  • Brief discussion of empirical error analysis (bonus on Assignment #3)
  • Overview of pointers and structures
  • Session 5: numerical derivatives: algorithm error and
    [more, details?], Richardson extrapolation
    • practice with pointers
    • first crack at linear algebra (eigenvalues)

Empirical error analysis (see 313 handout)
  • So far, we've analyzed approximation errors by examining
    relative errors with respect to a known, exact error.
  • But what if an exact answer is not known?
  • Compare approximate answers.
  • Suppose with N points or steps, your method predicts
    $\text{Approx}(N)$, and the exact answer is $\text{Exact}$. If $N$ is small
    enough that round-off errors are small, then

      $\text{Approx}(N) \approx \text{Exact} + \frac{\alpha}{N^\beta}$

    (for a power law error). Then calculate for a larger $N$, say $2N$:

    $\text{Approx}(2N) \approx \text{Exact} + \frac{\alpha}{(2N)^\beta}$

    $\left(\frac{(N - \text{Approx}(2N)}{\text{Exact}}\right) \approx \frac{\alpha}{(2N)^\beta}$

    and a log-log plot reveals the power law.
It doesn't have to be precisely $SN$, e.g. for Milne's rule integration, compare $N$ and $SN+3$ (or $SN-i$) ⇒ $A_N = h(n+1)$ for some $n_i$. Run $SN+3$ just as well (for some other $n$).

- See the error plot examples.
- Interesting feature: The round off error fluctuates much less than when comparing to the exact solution.

Pointers and structures ⇒ time for a bit of C/C++ "review".

Let's start with structures.

- Suppose you are doing calculations involving a quadratic equation. It is specified by the three coefficients, $a$, $b$, $c$.

$$ax^2 + bx + c = 0$$

It is very useful to keep these grouped together. We can do this in C/C++ with a structure:

```c
struct coefficients {
    double a;
    double b;
    double c;
};
```

Then we can make an instance:

```c
struct coefficients my_coefficients;
```

An equivalent construction is

```c
typed struct {
    double a;
    double b;
    double c;
};
```

Note we're the structure's name is `coefficients`, not `coefficients`.

If you can have doubles, floats, ints, our structure, whatever.
Then we'd make an instance with
quad-parameters my-coefficients;

How do we get the a, b, or c parts? With the "dot" notation
my-coefficients.a = 3.0;
alpha = my-coefficients.b;

Very useful concept, but even better if you can package together functions with the data as well. This leads to the idea of a class in C++, (later!)

Another example: Complex numbers

typedef struct
{
    double real,
    double imag;
} complex_number;

We can define
complex_number z;
x = z.real
y = z.imag

We'd like to know how Z1\times Z2 works
That's why we need a class, so that
Z1\times Z2 \rightarrow Z1.real \times Z2.real - Z1.imag \times Z2.imag

-see examples & structures at beginning of pointer_test.c
Pointers: see handout (pages on handout) on pointer test app.

When you define a `double` named `alpha` by:

```c
double alpha = 5.3;
```

The value 5.3 in binary floating point is stored in 32 bits (4 bytes) in a definite area of the computer memory. Memory locations 0x1000 in that location:

```
5.3 -> 0x1000, 1
0x1001, 0
0x1002, 1
```

If the program wants to do something with `alpha`, we can refer either to its value (at the moment), which is 5.3, or the starting address where it is stored, along with the knowledge that it is stored as a `double`.

A pointer to `alpha` has the address of `alpha`:

```
 allegedly Web addresses (URLs) are displayed as
 "http://www.physics.diu-state.edu/hcc/ calculus physics/index.html"
```

- `/logo.jpg` is a JPEG picture
- `/sessions.tar.gz` is a tarball

`alpha` is a `float`, not a `double`, structure, even a function. `alpha_ptr` is an address in memory, but we also need to know what is stored there. Examples:

```
double alpha, gamma; // ordinary doubles
double *beta_ptr;  // * in a definition beta_ptr is
// a pointer. The_ptr is just an memory

alpha = 3.1;
```

beta_ptr = &alpha; // & means "address of"

gamma = *beta_ptr + 5.1; // * is the "dereferencing" operator.

What is gamma? (5.1)
Two tricky extensions: pointers to functions and void pointers.
You'll see both in today's examples.

In derivative_test.cpp, we want to define forward_diff
with the approximation
\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]

```c
double forward_diff ( double x, double h )
{ return (f(x+h)-f(x))/h );
}
```

This is like our integration routines. It works only if
our function is named f. But what if it is named
`fuct`? Or we have several functions, all with the
same name?

⇒ pass the function. E.g. `double func1(double x)`
How do we do that? Pass the address of where the function
definition is stored ⇒ pass a pointer to the function:

```c
double forward_diff ( double x, double h, double (*f)(double x) )
{ return (f(x+h)-f(x))/h );
}
```

And call it with:
```c
diff_func = forward_diff(x, h, &func);
```

See derivative_test for more examples.
If function has a parameter like \( \alpha \) or \( \beta \), then it is defined:

\[
\text{double}
\]
\[
\text{func (double x)}
\]
\[
\text{return exp(-alpha*x)};
\]

Now we change the values of \( \alpha \) and \( \beta \) in a function.

We are passing:

- we want to allow for one, two, whatever numbers,
- and types of parameters

\( \Rightarrow \) void pointers

- points to an address, but don't specify whether it
  - is a double or int or whatever until used

\[
\text{double alpha, beta, i;}
\]
\[
\text{void *params_ptr;}
\]

\[
\text{params_ptr = &alpha;}
\]
\[
\text{\"i\" = 5;}
\]
\[
\text{\"n\" = 5;}
\]

but then we can't just say in the function:

\[
\text{beta = *params_ptr;}
\]

\( \Rightarrow \) type cast

\[
\text{beta = *(double *)params_ptr;}
\]

Now: TRY OUT EXAMPLES!

**COSL eigenystems, Hilbert matrix**

\[
\begin{bmatrix}
\frac{2}{n+1}
\end{bmatrix}
\]

- obvious place to use package \( \Rightarrow \) eigenvalues, eigenvectors

**Mathematica**

- \( A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \) \( \Rightarrow \) \( \{3.75, 3.75\} \)

\( \Rightarrow \) Table[\( \frac{1}{n}, \frac{1}{n+1}\) for \( n = 1, 2, 3 \) \( \Rightarrow \) Table[\( \frac{1}{n-1}, \frac{1}{n}\) for \( n = 1, 2, 3 \)]