# Q3 - Passive Networks

# Background

## Representing simplified passive networks

Passive networks are composed of combinations of capacitors, inductors, and resistors. In general a passive network can contain any combination of wiring, but a useful restricted case is to build two terminal (single input + single output) circuits that only consist of parallel and serial compositions of other two-terminal circuits. In this restricted approach, a passive network can consist of:

- A primitive component: resistor, capacitor, or inductor.
- Serial composition of networks.
- Parallel composition of networks. The input and outputs terminals of the sub-networks are connected, so that the parallel composition is still a two terminal network.

In order to represent such networks as in code we use the following structure:

```
struct Network
{
    char type;
    float value;
    vector<Network> parts;
};
```

The meaning of the members is:

- `type` : A single character specifying the network type. For primitive components this will be one of `R`, `L`, or `C`. For serial composition this is `&` (ampersand), and for parallel composition it is `|` (pipe).
- `value` : For primitive components this contains the non-negative value of the component, given as Ohms, Henries, or Farads. For non-primitive networks it is 0.
- `parts` : For non-primitive networks this is a non-empty list of networks to combine in serial or parallel. For primitive components the parts list is empty.

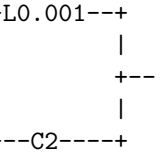
Examples of creating primitive components in code are:

```
Network r_1 =   { 'R', 1,     {} }; // A 1 Ohm resistor
Network c_2 =   { 'C', 2,     {} }; // A 2 Farad capacitor
Network l_1mH = { 'L', 0.001, {} }; // A 1 milli-Henry inductor
```

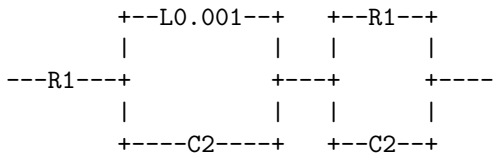
These could then be combined in a number of ways:

```
Network lc =  { '|', 0,  { l_1mH, c_2 } };  // Inductor and capacitor in parallel
Network rc =  { '|', 0,  { r_1, c_2 } };    // Resistor and capacitor in parallel
Network lcr = { '&', 0,  { r_1, lc, cr } }; // Series connection of multiple circuits
```

The `lcr` variable represents the following network:




while the final `lcr` variable represents the following network:




## Network IO

Networks are read and written as text using the following format:

- Primitive components: the letter `R`, `C`, or `L`, followed by the value of the component as a decimal value.
- Non-primitive networks: non-primitive networks begin with `(`, then contain one or more network parts separated by `&` or `|`, and finish with `)`.

The numeric value of primitives can be given with or without an exponent, but should be accurate to at least four significant digits. The default output format used by `ostream` for printing a `float` will always meet this requirement.

If a non-primitive network only contains one part, then it should be read as a degenerate series connection with only one component.

Examples of IO are:

- `R1` : A 1 Ohm resistor
- `C2.0000` : A 2 Farad capacitor
- `L0.001` : A 1mH inductor
- `(R1|C2)` : A 1 ohm and 2 Farad capacitor in parallel
- `( R1 & C1e-3 )` : A 1 ohm and 1mF capacitor in series
- `(R1|R2|R3|R4)` : Four resistors connected in parallel, with resistance from 1 to 4 Ohms.
- `(C1&C2&C3&C4)` : Four capacitors connected in series, with capacitance from 1 to 4 Farads.
- `((R1&C1)|(C1&L1))` : A serially connected resistor and capacitor, in parallel with a serially connected capacitor and inductor.
- `(R1)` : A single resistor connected in a degenerate series with nothing.
- `(R1&(L0.001|C2)&(R1|C2))` : The `lcr` example given above.

## Canonical network form

Not all networks can be expressed as combinations of parallel and serial components, but even for those that can be, there is no single unique representation. For example, the networks `(R2|R3)`, and `(R3|R2)` have a different structure (both in text and when converted to a `Network`), but represent the "same" circuit in an electrical sense.

*Canonicalisation* is the process of trying to restrict the set of possible representations, so that if two networks describe the same circuit structure, then they also have the same network representation. Two basic approaches to canonicalisation for these circuits are:

1. *Flattening parts vectors.* The two non-primitive networks `(C1&(C2&L3))` and `((C1&C2)&L3)` can be "flattened" into the circuit `(C1&C2&L3)`. Similarly `(C1|(C2|L3))` can be flattened to `(C1|C2|L3)`. However, the network `(C1|(C2&L3))` cannot be flattened. Degenerate non-primitives such as `(C1)` can also be flattened to `C1`.
2. *Sort the parts vector*: In the first example of `(R2|R3)` versus `(R3|R2)`, if we decide that smaller values come before larger values, then `(R2|R3)` is the only possible order. Similarly, if we decide that capacitors must come before resistors, then `(R2&C3)` would be canonicalised to `(C3&R2)`.

These two approaches can be applied to any network, by recursively canonicalising, flattening, and then sorting the parts. Pseudo-code for this is:

```
function make_canonical(c)
begin
    if is_primitive(c) begin
        return c
    end
    parts=c.parts   # Make a copy of the parts list so we can modify it
    for i in length(parts) begin
        parts[i] = make_canonical(parts[i])  # Recursively canonicalise each part
    end
    parts = flatten(c.type, parts)   # Flatten any parts with the same type
    parts = sort(parts)              # Sort the final set of parts
    return Network( c.type , parts )
end
```

This process should result in a functionally equivalent network which still contains the same primitives, but for which there is only one representation. For simplicity, the pseudo-code does not handle flattening of degenerate series.

This canonicalisation requires a sort order, so that we can define a less-than operator on `Network`. We define this as first sorting by the type of the network, then by the value or parts list. The full sort order is then:

1. By type: `&` < `C` < `L` < `R` < `|`. Note that this follows the standard character ordering for ASCII, which is also the ordering for `char`.
2. By value: for primitive components of the same type, lower values should appear before larger values.
3. By parts: for non-primitive networks of the same type the lexicographical order of the two vectors of parts should be used; lexicographical ordering is the same as the default ordering for `vector`.

Examples of ordering primitives are: `C0.5` < `C1` < `C2` < `R1e-3` < `R1` < `L0.3` < `L10`. Any serial composition (`&`) will compare less than any primitive, while any primitive will compare less than any parallel composition (`|`). For parallel compositions, we have `(C1|C2)` < `(R1|C2)` because `C1` < `R1`, and `(C2|L0.1)` < `(C2|R3)` because `L0.1` < `R3`.

Combining the sort order with the canonicalisation allows us to find a unique representation for any given network.

For example, for the input `((C1|L3)|(R2&R1)|(L1|C1))`:

1. `((C1|L3)|(R2&R1)|(L1|C1))` : Flattened inner (no effect)
2. `((C1|L3)|(R1&R2)|(C1|L1))` : Sorted inner
3. `(C1|L3|(R1&R2)|C1|L1))` : Flattened outer
4. `((R1&R2)|C1|C1|L1|L3)` : Sorted outer

Other examples are:

| Original | Canonical |
|---|---|
| `( R1.3 & C1.2 & L2 & L0.5)` | `( C1.2 & L0.5 & L2 & R1.3)` |
| `((C1.2 & L2) & L0.5 & (R1.3))` | `( C1.2 & L0.5 & L2 & R1.3)` |
| `(((R4)\|R2)\|R1)` | `(R1\|R2\|R4)` |
| `((R4\|R2\|(C3&C1))\|R1)` | `( (C1&C3) \| R1 \| R2 \| R4)` |
| `((R4&R2&(C3\|C1))&R1)` | `( R1 & R2 & R4 & (C1\|C3))` |

*Note: spaces are irrelevant in the printed representation.*

### Complex impedance

The complex impedance of a network is defined in terms of a frequency omega, and follows the standard rules for impedance:

- `Z( omega, Rv ) = v`
- `Z( omega, Cv ) = -j/(omega*v)`
- `Z( omega, Lv ) = j*omega*v`
- `Z( omega, (p1&p2&...&pn) ) = Z(omega,p1) + Z(omega,p2) + ... + Z(omega,pn)`
- `Z( omega, (p1|p2|...|pn) ) = 1 / ( 1/Z(omega,p1) + 1/Z(omega,p2) + ... + 1/Z(omega,pn) )`

Examples of impedances of particular networks are:

| Network | omega=0.5 | omega=1 | omega=2 |
|---|---|---|---|
| R1 | 1 | 1 | 1 |
| C1 | -2j | -j | -0.5j |
| L1 | 0.5j | j | 2j |
| (R1&C1) | 1 -2j | 1 -j | 1 -0.5j |
| (C1&C2) | -1.5j | 0 | 1.5j |

### Transfer functions

Given two networks `n1` and `n2`, we can consider them as a potential divider:

```
Vin -----+
         n1
         +----- Vout
         n2
         |
        Gnd
```

The transfer function for this potential divider can be expressed as:

```
H(omega, n1, n2) = Vout / Vin
                 = Z(omega,n2) / ( Z(omega,n1) + Z(omega,n2))
```

For example, given the network `n1=R2` and `n2=C3`, we would end up with the following:

| omega | Z(omega,n1) | Z(omega,n2) | H = Vout / Vin |
|---|---|---|---|
| 1 | 2 | 0 - 0.33333i | 0.027027 - 0.162162i |
| 2 | 2 | 0 - 0.16667i | 0.006897 - 0.082759i |
| 4 | 2 | 0 - 0.08333i | 0.001733 - 0.041594i |

## Tasks

### T1 - Network operations (35%)

#### T1.1 - is_primitive function (5%)

Implement the function `is_primitive`, which returns true if the argument is a primitive component, and false otherwise.

### T1.2 - Component list (10%)

Write a program called `print_component_list.cpp`. This program should read a single `Network` from `stdin`, and print out the primitive components that it contains to `stdout`. Each component should be printed on a separate line.

No specific order is required, as long as each component gets printed once for each time it appears in the input network.

Component values do not need to exactly match the input text, as long as the values are correct to four significant digits.

Example usage:

```
$ ./build_print_component_list.sh
$ echo "(R1 | R2.4 | ( C1.99999999 & L2) | L2 )" | ./print_component_list
R1
R2.40000
C2
L2.00
L2
```

*Note: The varying numbers of output digits are to make the point that it is the numeric value that matters in Network IO, not the textual representation. No sensible code would print `L2.00` and `L2` for the same component, but it is valid.*

### T1.3 - Network comparison (10%)

Declare and define an implementation of the less-than operator for `Network`. The declaration should be in `network.hpp`, and the definition in `network_ops.cpp`

### T1.4 - Canonicalise (10%)

Implement the function `canonicalise`.

## T2 - Network IO (20%)

The program `test_network_io.cpp` can be used to check whether input and output are working, and may be useful in this question.

### T2.1 - Writing of networks (10%)

Complete the implementation of the `<<` operator for `Network`.

### T2.2 - Create test-cases (10%)

Complete the function `create_test_networks` so that it returns at least ten `Network` instances that can be used to test that IO operates correctly.

You should try to anticipate corner cases that might not work when writing circuits.

*Hint: you may find this function useful for other test purposes. Note: you are allowed to have (lots) more than ten test-cases if you wish.*

## T3 - Impedance (45%)

### T3.1 - Calculating impedance (10%)

Complete the function `impedance` so that it can calculate impedance for all five network types.

### T3.2 - Transfer function (10%)

Complete the function `transfer_function` which takes three parameters:

1. A Network `n1`.
2. A network `n2`.
3. A vector of floats `omega`.

The return value should be a complex vector containing the transfer function for the potential divider at each value of omega.

### T3.3 - Logarithmic spacing function (10%)

Implement the function `make_log10_space(a,b,n)`.

Given the range `[a,b]`, the function should return `n` points, equally spaced logarithmically (base-10). The first point should be `a`, and the last point should be `b`.

| Input | Output |
|---|---|
| `a=1, b=10, n=2` | `[1, 10]` |
| `a=1, b=100, n=3` | `[1, 10, 100]` |
| `a=0.01, b=100, n=3` | `[0.01, 10, 100]` |
| `a=1, b=10, n=5` | `[1, 1.7783, 3.1623, 5.6234, 10]` |
| `a=0.1, b=10, n=5` | `[0.1, 0.31623, 1, 3.1623, 10]` |

The above values are rounded to an arbitrary number of digits for display - the function does not need to perform any rounding. Any reasonable calculation order using `float` or `double` expressions and functions is acceptable.

### T3.4 - Transfer function program (15%)

Create a program `print_transfer_function.cpp` which reads two networks, and then prints out the properties of the transfer function at logarithmically spaced points.

The program should take three optional command-line arguments:

1. `a` (default = 1)
2. `b` (default = 100)
3. `n` (default = 10)

The program should read two networks `n1` and `n2` from `stdin`, and then calculate the transfer function of `n1` and `n2` when treated as a potential divider at `n` logarithmically (base-10) spaced values of `omega` in the inclusive range `[a,b]`

The printed output should consist of three columns seperated by commas, with one row per value of `omega`.

1. `Omega` : frequency

2. `abs(H)` : Absolute value of the transfer function
3. `arg(H)` : Phase of the transfer function in radians

White-space may be inserted within a line if desired.

Example output:

```
$ echo "C1.5 (L0.5&R1)" | ./print_transfer_function 0.1 10 9
   0.1,       0.149623,  1.47076
   0.177828,  0.264601,  1.39276
   0.316228,  0.461972,  1.25378
   0.562341,  0.770448,  1.00931
   1,         1.10282,   0.628796
   1.77828,   1.18999,   0.251803
   3.16228,   1.10282,   0.0664762
   5.62341,   1.03879,   0.0138287
  10,         1.01298,   0.0025974
```

*Note: These results did not come from a C++ program, and do not reflect the exact numeric values or whitespace formatting required.*