Exploring Arduino Analog Input Resistance

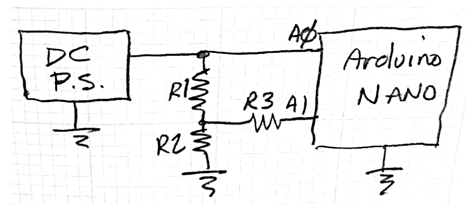
16 Feb 2023

B. Rasnow

Objective: Better understand the DC input resistance of the Arduino Nano analog inputs. My goal is not to quantify the inputs like a Thevenin equivalent circuit[1], but much more simply, to determine ranges of values of series resistors (R3 in Fig. 1) that might protect the Arduino from excessive voltage while not introducing errors in the values that are measured.

Methods:

1) Documentation – google it.
2) Measure it – measure analog voltages through voltage dividers and series resistors. The circuit in Fig. 1 was used. The DC power supply (LM317) was set to ~2.5V and not adjusted during the experiment (to ensure not exceeding a safe 5V). A0 was sometimes switched to ground and the power supply Vout. The Arduino was programmed with the following code, modified from File→ Examples → AnalogReadSerial (Fig. 1)




```
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Serial.print(analogRead(A0));
  Serial.print("\t");
  Serial.println(analogRead(A1));
  delay(200);
}
```

Figure 1. Schematic to measure Arduino input impedance and source code.

Results:

1) Documentation: Searching "arduino analog input impedance", I found a rich discussion at: https://forum.arduino.cc/index.php?topic=65134.0[2]: "The ATmega328P datasheet has this info way in the back: Table 28-16 (page 328). The analog input resistance is claimed to be 100 Mohms. During an actual sample, the input resistance is temporarily a lot lower as the sampling capacitor is charged up so it is recommended that whatever you connect to the A/D have an output impedance of 10k or less for best accuracy." And "the sample/hold cap has only 6us in which to charge and settle in normal operation, fortunately its very small (14pF)". They recommend adding an external 0.1uf cap when measuring DC voltages.
2) Measurements:
    a) With R1=R2=R3 = 10k and Vin ~2.5V generated expected output with no evidence of errors (Fig. 2). Mean and standard deviation computed in Matlab were 626.1±.4 and 312.6±.5 – A1 was ½ of A0.
    b) Changing R3 to 1MΩ changed A0 and A1 to 626.0±.3 and 402.1±2.4
    c) Leaving R3 as in b) but grounding A0 → 0±0 and 227.8±4.9
    d) Leaving R3 as in b) and c), and measuring R3's voltage on both A0 and A1 in parallel → A0 = 311.8±1.2 and A1 = 310.5±1.1

```
…
626     313
626     312
626     313
626     313
626     312
…
```

Figure 2. Arduino output in Serial Monitor was pasted into Matlab.

Discussion: A few other experiments were done, e.g., changing R1 and R2 to 1MΩ, but the above are adequate to understand the underlying phenomena. The Arduino's DC input resistance is high enough that putting 10kΩ in series doesn't cause errors. We expected A1 = ½ A0 and that's what we measured within 1 ADU (analog-to-digital unit) and within the 1% error rating of the voltage divider resistors. This might protect the input circuits from excess currents if the input voltage exceeds 5V. This is a recipe to protect the Arduino – always put a 10kΩ resistor in series with Ain pins.

1) d) tells us that A0 and A1 differ by 1.3 ADU even though they're measuring the same voltage – so we should consider channel-to-channel variability of ~1-2ADU as the accuracy limits of this device.
2) A 1MΩ series resistor causes significant errors – 402-313 = 89ADU and 313-228=85ADU, about 87/313 = ±27% error. Why is the error sometimes high and sometimes low (by about the same amount)? I think the answer involves the internal 14pF sample and hold capacitor. The error depends on how much the voltage changes between the previous and next measurement. E.g., if A0 and A1 are at the same voltage, then the capacitance effect is null (result d). But as the voltage differences increase, then the high external resistance doesn't allow the C to charge or discharge before the next reading. If the prior voltage reading was higher, then residual charge on C will make the next reading too high, and conversely, if the prior voltage was lower, then the S/H capacitor won't have time to charge up to the new voltage and the next reading will be low. That's consistent with our results b) and c).
3) Theoretically, assuming the quotes we found are true, 10kΩ in series with 14pF has a time constant[3] of τ = 14e-12*1e4 = .14µsec, so the 6µsec between samples is 6/.14 = 42τ – far more than enough time to settle (~1% error in 5τ). R3 = 1MΩ is 100 times larger, so the 6µsec = .42τ , and we'd expect the capacitor to charge to ~1-exp(-.42) ~ 34% of the right answer. We got about twice that much (27% error = 63% of the right answer), I think being within a factor of 2 is pretty reassuring.

<u>Conclusions</u>: Input resistance, and more generally, input impedance[4] (a complex value that includes capacitive resistance), is an important characteristic of a circuit. The high input resistance of the Arduino's analog inputs allows us to add a 10kΩ resistor in series without causing voltage errors for DC inputs. The series resistor will limit current and protect the Arduino from overvoltages.

Capacitors add significant complexity because their effects depend on their history – how much charge is stored in them from the past. The more resistance in a circuit, the slower the capacitors charge and discharge to equilibrium at DC. With time-varying voltages (AC), there is no static equilibrium, so it's even more challenging. We'll have to wrap our heads around this complicated stuff – and we'll find that complex numbers and Fourier transforms are going to help us *simplify* these characteristic behaviors!

---

[1] https://en.wikipedia.org/wiki/Thévenin's_theorem
[2] https://forum.arduino.cc/index.php?topic=65134.0
[3] https://en.wikipedia.org/wiki/RC_time_constant
[4] https://en.wikipedia.org/wiki/Electrical_impedance

Appendix: Raw data and analyses

```
>> data = [... % R1=R2=R3=10k
626    313
627    313
627    313
626    313
% about 60 similar lines deleted
626    312
626    313
626    312
626    312
];
>> mean(data)
ans =
  626.0811  312.5946
>> std(data)
ans =
    0.4303    0.4943
>> data = [... % R1 = R2 = 10k, R3 = 1M
626    404
626    404
626    404
% about 60 lines deleted
626    402
626    403
626    402
];
>> mean(data), std(data)
ans =
  625.9848  402.1364
ans =
    0.3278    2.3657
>> data = [... % R1 = R2 = 10k, R3 = 1M, A0 grounded
0      228
0      228
0      229
% about 70 lines deleted
0      229
0      228
0      229
0      229
];
>> mean(data), std(data)
ans =
         0  227.7681
ans =
         0    4.8964
>>
>>
>> data = [... % Same R's, A0 = A1
312    310
% ~150 lines deleted
314    313
310    309
310    309
311    310
];
>> mean(data), std(data)
ans =
  311.7838  310.4775
ans =
    1.1937    1.1371
```