Reliable Line Tracking

William Dubel

January 5, 2004

Introduction:

A common task for a robot is to follow a predetermined path. One popular way to mark a path is with a high contrast line, such as black electrical tape on a light colored surface. When deciding on the type of line to create, consider the type of sensor that will detect it, as well as how easy it is to apply. In describing reliable line tracking, most of the work is in the sensor arrangement and programming. For this paper, lines created with electrical tape will be considered, but you should be able to adapt your system to most high contrast lines.

The first step is to determine what types of paths your line will describe. Will you have curvy or angular turns? Will the lines intersect? Will the line always be available? I will attempt to help you design systems that can handle these common types of paths. I've divided this paper into two parts, one for the electronics and one for the programming. You can read the parts in any order, and jump back and forth if you like. If you are someone who likes to think things through on your own, you may want to take a moment to consider what may cause a particular configuration to fail before I provide my own reasons. You can skip the electronics part if you have found or already built line-tracking electronics.

Part I: Electronics

You can probably assume your line has some turns in it, and even if it doesn't, your robot will never be so lucky as to line up perfectly on the line and it will need to make adjustments along the way. We need a sensor that can tell us when we are no longer aligned with the line, either because the line has moved away from the robot (a turn), or the robot has moved away from the line. To your robot, it's all the same. You can use an infrared (IR) light emitting diode (LED) and a phototransistor to measure how much light is reflected from the surface, to detect the presence of a line.

Phototransistors are similar in function to a transistor, where the base (or gate) of the transistor is controlled by light. The more light into the sensor (which usually looks like a standard LED), the more current that can flow through the two leads.

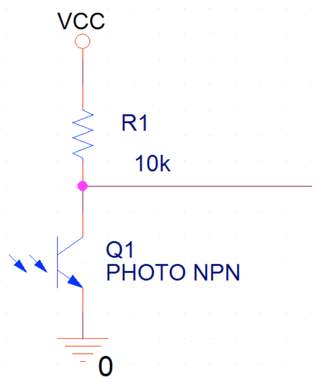

Fig. 1: A phototransistor in an [inverse] "light to voltage" circuit.

The phototransistor is paired with an infrared LED. The LED emits light against the surface, and the phototransistor turns on if enough of the light is reflected back. The output voltage will range between Vcc and 0 volts, [inversely] depending on how much light is reflected. There are many components available that pair these two devices, but you can also make your own. I suggest purchasing the paired component, because they are usually more reliable and smaller than a homemade version. Since the output of the phototransistor circuit is a voltage, you'll need to decide on a threshold voltage that signifies the presence of a line. If you are reading this value from an A/D converter, you'll program that value into your microcontroller – probably a value between 0 and 255. You may determine that anything above 127 is a black line, and anything else must be floor.

I prefer to use an analog comparator (such as the LM311) to determine if the voltage is greater than your threshold or not. The threshold voltage can be created with a potentiometer used as a voltage divider. That way, the threshold can be adjusted at any time, and you can save the use of an A/D converter and some programming. The output from the comparator will be logic 1 if there's a black line, or logic 0 if not. You can reverse the inputs on the comparator to get the opposite effect, to produce logic 1 when enough light is reflected and logic 0 otherwise.

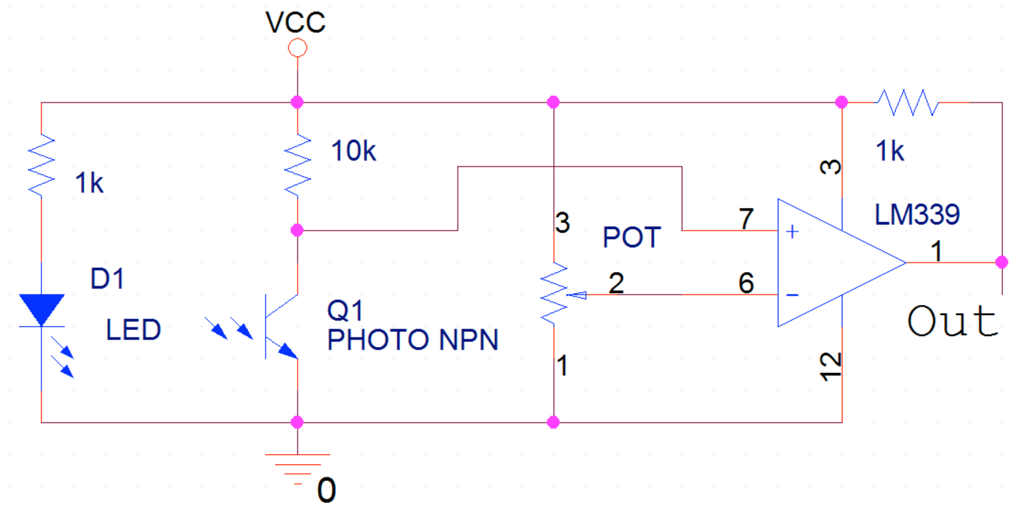

Fig. 2: Circuit outputs logic high when a black line is detected.

The complete circuit for a single-pair line-detecting module is shown in figure 2. I call it a single-pair module to distinguish it from line-tracking modules with additional pairs of IR LED's and phototransistors described later. While the comparator based output requires a few extra components, it greatly simplifies programming. I use an LM339 comparator because a single package contains four comparators, which will be useful for additional sensor pairs. Also note that most comparators require a pull-up resistor for proper use. Finally, I considered the use of hysteresis on the comparator, but after testing this circuit I don't see the need. Now we a have module that can determine the presence of a line. The pot used to adjust the threshold voltage is shown on the module as well. This is not the only circuit that will work, and you will find that different designs are available as well. Alternately, by correctly biasing the phototransistor, you can use a Schmitt triggered buffer (or inverter) in place of the comparator and potentiometer circuit above. It's a simpler design, but lacks the ability to adjust the threshold which can be useful on some surfaces.

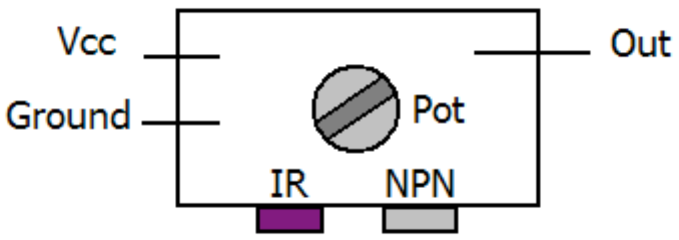

Fig. 3: Line detection Module

The only problem with using this single-pair module to track a line is that once the robot moves off the line, it doesn't know which direction to turn to get back to the line. It could try turning a little bit to the left, checking for a line, and if unsuccessful, it could turn to the right until it finds the line, but this isn't very efficient. It would be nice to know the direction the line disappeared to. One way to do this is by using two phototransistor pairs. Notice if you are building the two of them in a module you can share the comparator chip and possibly the pot output voltage. You will only need to add the IR-phototransistor pair, as well as the necessary resistors. I like to use separate pots to help compensate for any component variances.

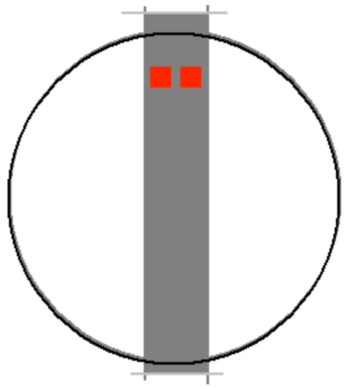

Fig. 4: Circular robot on dark line. Red squares represent the Detector Modules.

Now with the two pairs of phototransistors, we can arrange them close together on the robot so that when the robot is centered on the line, both pairs detect the line, as shown in figure 4. As the robot moves off of the line, one of these pairs will invariably lose the line before the other. Now the robot can determine which direction to go to re-center on the line. I'll mention a few things here that may be obvious:

First, you'll want the line-tracking module positioned as far forward as possible. That way, as long as the module is centered on the line, the robot is already pointing (and moving) in the correct direction. Compare this to having the module in the center of its drive wheel axis. You could now have the robot pointing in any direction, even though the module is centered on the line. In general, the closer the line detection is to the center of rotation for the robot, the less stable your robot will be when tracking a line. It will “wobble” back and forth much more (in attempts to re-center), and if it’s completely centered on the axis, it will be practically useless. Anything behind the axis of rotation for your robot would be unstable.

Second, the line-tracking module will need to be as close as practically possible to the surface of the line. You want to get the maximum amount of reflection from your IR LED to your phototransistor, without picking up the glare from many types of surfaces.

Avoiding glare brings us to the last note. You may get better results from having your sensor module angled. This helps avoid some problems with glare, where the phototransistor picks up light from the IR whether the surface is light or dark. I’ve found that a 15 to 30 degree angle is most reliable, but it will depend on your phototransistor pair. Some are designed to work perpendicular to the surface.

What can we do with our current line-tracker? What won’t it do? Given the right programming, the two-pair line-tracker described so far can track straight lines, curvy lines, and lines with angled turns of less than 90 degrees. It won’t do anything else reliably, but that may be all you need. It won’t properly follow lines with breaks unless you want the robot to turn around completely and follow the same line in the opposite direction – which may indeed be what you want it to do. It also can’t properly follow angles of 90 degrees or greater. The reason for both of these limitations is that when the line-tracking module reaches the end of the line or a 90-degree (or greater) turn, three things can happen, with about the same certainty:

The sensor pairs may both run off of the line at the same time – or at least as perceived by your processor. Now your processor will not know if the line has ended, or if it has just made a sharp turn. What action should it take? Continue forward? Or turn until it finds the line again. Which way should it turn to find the line?

The right pair may exit the line before the left pair. This could happen at the end of the line, if the robot exited at a slight angle, or if the tape wasn’t cut perpendicular at the end. Either way, it would be perceived as a turn to the left. Consider the following turn:

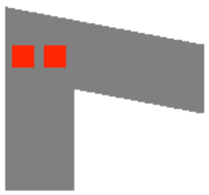
Fig. 5: A turn which would confuse our two pair logic.

The right sensor exits first, and the processor decides that the line must have moved to the left of the sensors. This is incorrect of course, and the robot would turn around and follow the line backwards now.

Finally, the left pair could exit before the right pair, and the robot assumes a similar fate. Clearly, our two pair line-tracking system is inadequate for lines of more than just curves.

Enter the three pair line-tracking module: The three pair module has an extra IR-phototransistor pair, and possibly a pot, to provide three logic outputs to your processor. The sensor pairs are usually arranged as follows, shown as it would be centered on a line:

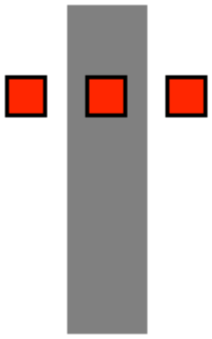

Fig. 6: The three-pair line-tracking module, centered on a line.

This arrangement is the most commonly found commercially, except maybe for the single-pair sensor. The sensors are spaced so that at least one sensor will detect the line at any given time.

The line-tracking module now has the capability to handle end-of-line and sharp turn situations. At the end of the line, all of the sensors will lose the line. At a sharp turn, the right or left sensor will have to detect the line, at least by the time that the center sensor loses the line. Because this style of line-tracker, given the right programming of course, can follow almost any line, it's probably the most popular style of line-trackers. If your robot doesn't need to handle intersections, you can jump ahead to the programming.

But wait, I want my robot to handle intersections! You may be thinking right now of ideal conditions where all intersections are exactly perpendicular, and that the robot will always be aligned on its line when it reaches the intersection. Too bad – nothing is ever ideal for a robot! You may be able to use the three-pair module for perfectly drawn intersections and your robot may work most of the time. However, since there are angles that the robot could approach the intersection and be confused, it is not a reliable system. Consider the following 'T' intersection:

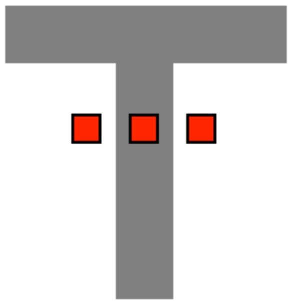

Fig. 6: Three-pair module approaching an intersection.

The module approaches an ideal intersection. One outer sensor pair will probably detect the presence of the intersecting line before the other. Let's assume the right sensor pair detects the line first. The robot will mistakenly begin a right turn, believing that the line must be a 90 degree turn to the right. The left sensor will now detect the line, and the module will detect a line on all three sensors and realize (programmed correctly) that it is now on an intersection, and the processor can take the correct action. But this won't always work. If the robot (including its line-tracking module) is off track before entering an intersection, it will not be clear if the robot has arrived at a turn or at an intersection. Since you may want the robot to go a particular direction at an intersection or know that it has reached an intersection at all, this distinction is important. You will need an additional sensor pair.

The four-pair line-tracking module is really a hybrid of the two-pair and three pair modules. It has four outputs to more accurately indicate the presence of a line and possible intersections. Recall that the problem with the three-pair module arises when a robot not completely aligned on its line approaches an intersection. It can't determine if it has reached a turn or an intersection, and can only assume a default action to take. The four-pair sensor has two center sensor pairs to maintain a centered position on the line at all times, as well as two outer sensor pairs that detect a sharp turn or an intersection.

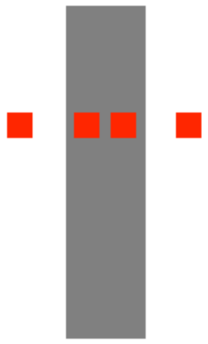
Fig. 7: Four-pair sensor arrangement.

The outer sensor pairs only have significance when the center sensor pairs are aligned on the line. A complete four-pair module produced from the schematic in figure 2 requires only one LM339 comparator, though I'd recommend 4 separate pots, and prudent use of decoupling capacitors.

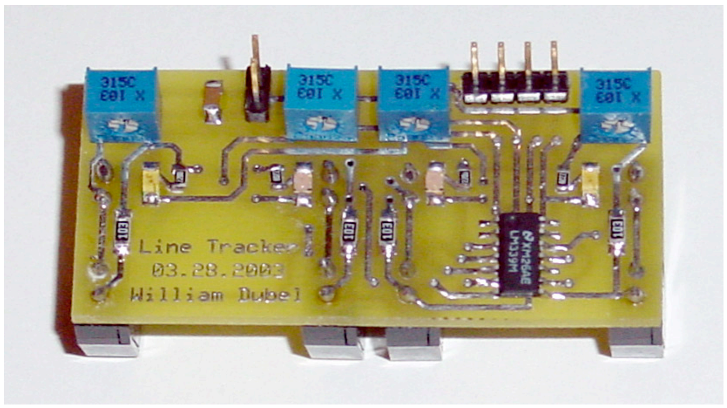

Fig. 8: A four-pair line-tracking module.

A completed board using dimensions appropriate for general-purpose electrical tape would be about 2 ¼" wide. The design shown in figure 8 provides indication of a tracked line visually with LEDs, as well as through a header with standard TTL outputs suitable for a microcontroller. Power to the board is provided through a 2-pin header at 5 volts. The four blue squares are the threshold voltage adjustment pots, and from the image you can see they are all set to the same value. The sensor pairs are combined IR LED and phototransistors, part number OPB745, available at Digikey, among other suppliers. They are one of the larger sensor pairs I've seen, but they work very well. However, don't hesitate to use your favorite sensor pair on your board. Most of the consideration goes to the spacing of the center sensor pairs, to accommodate ¾" electrical tape. You should design your board so that it works with the thickness of the line you intend to use.

To make use of the new information provided by the additional sensor, priority is given to the two center sensor pairs. They must be centered on the line before the processor can correctly discriminate a sharp turn and an intersection. You can also make use of the outer sensors to initially find a line, and also to assist if the line-tracking module has veered completely off of its center sensors, without detecting which of the center sensors exited first. But to correctly discriminate turns and perpendicular intersections, it's important that both center sensors are aligned with the line.

I'll leave you with the four-pair line-tracker, but there are situations where you may want additional information from your line-tracker. You may want the ability to turn exactly 90 degrees off of the line or you may want to handle intersections that are not perpendicular. There are hardware configurations to solve these problems that can make programming your robot to track during these situations possible, or at least easier. However, you will probably find that the two-pair sensor works just fine for most tracks, in which case even the four-pair line-tracker already too much information. Before you begin redesigning your hardware to provide additional information about your environment, make sure your programming is already making the best use of the information you already have!

Part II: Programming

You have good line-tracking hardware, but what about the programming? The programming is always more difficult than electronics to explain. Unlike electronics where the primitives are always the same, programming constructs vary between languages. Depending on your programming background, you may be very familiar with interrupts – or never used them. Even given the same language, popular features may or may not be available on your compiler. This is typical in a microcontroller environment. For example, most microcontrollers have a C compiler available, although recursion is usually not available (due to limited stack space). Because of differences not only in programming languages, but also in the capabilities of different microcontrollers, programmers will usually settle on not only a particular language, but for a particular microcontroller environment. At a compromise of not providing the most efficient solution in order to satisfy the masses, I have created my examples out of a common denomination of the C language usually available with most microcontroller environments. Needless to say, you'll just have to bear with my examples and do your best to apply them usefully in your environment. My primary goal is to introduce the basic logic of line tracking, and a practical way to implement it in C.

Most microcontroller environments provide some way to communicate directly with its I/O pins. Usually the pins are addressed at the byte level as a group, and some mnemonic is defined in a header file to make programming easier. For my examples, we'll define whatever that group (sometimes called a port) of bits lies on as `LT_PORT`. The bits are the logic values read from the line-tracker. In the case of a two sensor pair line-tracker, there would be 2 bits. For example, assume that the line tracker is attached to I/O pins 0 and 1 of PORTB of your microcontroller. The following definition would suffice:

```
#define     LT_PORT     PORTB & 0x03
```

I have masked the `PORTB` with hex value 3, so that I'm only looking at the two least significant bits of that port. On some microcontrollers, those pins would be called PB0 and PB1, although I'm sure if you look in the header files, you'd find a similar mnemonic for the whole byte.

```
// another way to do the same thing - microcontroller dependent
#define     LT_PORT     (PB1<<1)|PB0
```

Don't forget to set the direction of those bits so that they are inputs. Now we can make choices based on the value of `LT_PORT`. For my implementation, 0 means a black line present (or nothing detected), and logic 1 means surface detected. For example, if tracking a line were the sole task of your robot, the following code will work just fine:

```
unsigned char trackToLine()
{
      static unsigned char prevDir = 0;

      if (LT_PORT!=prevDir)
      {
            switch (LT_PORT)  // using a two-pair line-tracker
            {
                  case 0x00:  // centered on the line
                  {
                        updateMotors(FORWARD, FORWARD);
                        break;
                  }
                  case 0x01:  // line to the left
                  {
                        updateMotors(REVERSE, FORWARD);
                        break;
                  }
                  case 0x10:  // line to the right
                  {
                        updateMotors(FORWARD, REVERSE);
                        break;
```

```
                }
                case 0x11:  // no line
                {
                      updateMotors(FORWARD, FORWARD);
                      break;
                }

          }
      }
      if (LT_PORT==0xF && prevDir!=0xF)   // lost the line,
          prevDir = LT_PORT;              // continue prevDir

      return LT_PORT;
}

int main()
{
      init();      // your initialization routine

      while (1)    // infinite while loop
      {
            trackToLine();
      }
}
```

Just to review a few things in the code above in case you haven't used C in a while: I have defined two functions, `trackToLine()` and `main()`. When your microntroller starts (resets), it will execute `main()`, and anything `main()` calls, but nothing else. Once it finishes `main()`, it will run it again, over and over if you let it. Many first time microcontroller users do this without realizing it, and their code still works to their satisfaction.

Inside a while loop in my `main()`function, I call `trackToLine()`. The function `trackToLine()` contains the logic to update the motors depending on the current state of `LT_PORT`, and possibly the previous state if the line was lost. The previous state is maintained by the static variable `prevDir`. I use `unsigned char` because it's all you need. Register space on microcontrollers is always at a premium. Keeping track of the previous direction is important because the robot may run completely off the line faster than its motors can react. If this were to happen without some memory of the previous direction, the (stateless) choice of action would be based only on the current reading of the line-tracker, which would probably be to just move forward. This is what most otherwise decent line-tracking programs are missing – causing the robot to apparently run off the line without regard to the direction the line disappeared to. Notice that the code only updates the motors when there has been a change in the current state of `LT_PORT`.

`FORWARD`, `STOP`, and `REVERSE` are constants defined as different speeds for the drive motors. These constants might be used by your function to update your drive motor speeds, which I am representing with the call to `updateMotors()`. Your implementation of `updateMotors()` will depend on how your microcontroller communicates with your motors.

Looking closer at the heart of the `trackToLine()` routine, you'll find that the switch statement updates the motors depending on the value of `LT_PORT`. In this code, the switch routine handles four scenarios, one case for each possible with a two-pair module. With a four-pair module there are sixteen possibilities of sensor outputs, and your switch statement will need to be expanded accordingly. Recall that the motivation for a four-pair line-tracking module was the ability to detect perpendicular intersections in your lines. As mentioned in the electronics part, the center pairs must be centered on the line before you can correctly discriminate an intersection and a sharp turn. Consider the hex value 0xC, or binary 1100 on `LT_PORT`. You could interpret that as a 90-degree turn to the left, but it could also be an intersection. Since the module's center pairs are not centered on the line, you cannot be sure. With both the center pairs aligned, the chance of misinterpreting an intersection is very low. The following is a switch statement for a four-pair line-tracker:

```
switch (LT_PORT)  //priority to the center sensor pairs
{
      case 0xF:         //          1 11 1
      {
            updateMotors(M_FORWARD, M_REVERSE);
            break;
      }
      case 0xE:         //          1 11 0
      {
            updateMotors(M_FORWARD, M_REVERSE);
            break;
      }
      case 0xD:         //          1 10 1
      {
            updateMotors(M_FORWARD, S_REVERSE);
            break;
      }
      case 0xC:         //          1 10 0
      {
            updateMotors(M_FORWARD, S_REVERSE);
            break;
      }
      case 0xB:         //          1 01 1
      {
            updateMotors(S_REVERSE, M_FORWARD);
            break;
      }
      case 0xA:         //          1 01 0
      {
            updateMotors(S_REVERSE, M_FORWARD);
            break;
      }
      case 0x9:         //          1 00 1
      {
            updateMotors(MF_FORWARD, MF_FORWARD);
            break;
      }
      case 0x8:         //          1 00 0
      {
            updateMotors(M_FORWARD, S_REVERSE);
            break;
      }
      case 0x7:       //          0 11 1
      {
            updateMotors(M_REVERSE, M_FORWARD);
            break;
      }
      case 0x6:       //          0 11 0      //not sure what to do
      {
            updateMotors(S_FORWARD, S_REVERSE);
            break;
      }
      case 0x5:       //          0 10 1
      {
            updateMotors(M_FORWARD, S_REVERSE);
            break;
      }
      case 0x4:       //          0 10 0
      {
            updateMotors(M_FORWARD, S_REVERSE);
            break;
      }
      case 0x3:       //          0 01 1
      {
            updateMotors(S_REVERSE, M_FORWARD);
            break;
      }
      case 0x2:       //          0 01 0
      {
            updateMotors(S_REVERSE, M_FORWARD);
            break;
      }
      case 0x1:       //          0 00 1
      {
            //maybe stop is better
            updateMotors(S_REVERSE, M_FORWARD);
            break;
      }
      case 0x0:       //          0 00 0
      {
            updateMotors(MF_FORWARD, MF_FORWARD);
            break;
      }
}
```

I added additional relative motor speeds to make the line tracking a little smoother. Of course, the relative speed differences will depend on your implementation. In my constants, S is for slow, M is for medium, and F is for fast. So MF_FORWARD would be a constant that would move a motor medium-fast forward.

If you'd like to be doing other things at the same time such as obstacle avoidance, things become less trivial. You don't want your robot to run into someone just because the line it was following went underneath someone's shoe. By coding the while loop outside of the `trackToLine()` procedure, you can decide what procedures can be called in turn with `trackToLine()`, as well as a condition that will break out of the line tracking while loop. For example, a routine that tracks a line until an obstacle is reached (as determined by an IR sensor), that also updates a blinking LED, may look like this:

```
while (IR <= 80)  // while nothing in front of the IR sensor
{
      trackToLine();
      updateLED;  // function to toggle the LED output after 10 calls
}
// your function to stop your drive motors
updateMotors(STOP, STOP);
// now do something to get around the obstacle
```

Since `trackToLine()` returns the value of the `LT_PORT`, you can use the call to `trackToLine()` as part of a condition to exit your while loop. For example, with a four-pair line-tracking module, you may want to stop on an intersection:

```
// as long as we are not on an intersection
while (trackToLine() != 0x0F);

// your function to stop your drive motors
updateMotors(STOP, STOP);
```

Of course, if you have timers and interrupts available in your environment, you'll definitely be able do more things at once with your microcontroller. This is just to get you started; by now you probably have a few ideas of your own to test!